
HLA Reference Manual

1	HLA Overview	1
1.1.1	What is a "High Level Assembler"?	1
1.1.2	What is an "Assembler"	4
1.1.3	Is HLA a True Assembly Language?	4
1.1.4	HLA Design Goals	5
1.1.5	How to Learn Assembly Programming Using HLA	7
1.1.6	Legal Notice	7
1.1.7	Teaching Assembly Language using HLA	8
2	The Quick Guide to HLA	25
2.2.1	Overview	25
2.2.2	Running HLA	25
2.2.3	HLA Language Elements	26
2.2.3.1	Comments	26
2.2.3.2	Special Symbols	26
2.2.3.3	Reserved Words	27
2.2.3.4	External Symbols and Assembler Reserved Words	27
2.2.3.5	HLA Identifiers	27
2.2.3.6	External Identifiers	27
2.2.4	Data Types in HLA	27
2.2.4.1	Native (Primitive) Data Types in HLA	27
2.2.4.2	Composite Data Types	28
2.2.4.3	Array Data Types	28
2.2.4.4	Record Data Types	28
2.2.5	Literal Constants	29
2.2.5.1	Numeric Constants	29
2.2.5.1.1	Decimal Constants	29
2.2.5.1.2	Hexadecimal Constants	29
2.2.5.1.3	Binary Constants	29
2.2.5.1.4	Real (Floating Point) Constants	29
2.2.5.1.5	Boolean Constants	29
2.2.5.1.6	Character Constants	29
2.2.5.1.7	String Constants	30
2.2.5.1.8	Pointer Constants	30
2.2.5.1.9	Structured Constants	30
2.2.6	Constant Expressions in HLA	30
2.2.7	Program Structure	31
2.2.8	Procedure Declarations	31
2.2.8.1	Declarations	32
2.2.8.2	Type Section	32
2.2.8.3	Const Section	33
2.2.8.4	Static Section	33
2.2.8.4.1	The @NOSTORAGE Option	33

2.2.8.4.2	The EXTERNAL Option	33
2.2.8.5	Macros	34
2.2.9	The #Include Directive	35
2.2.10	The Conditional Compilation Statements (#if)	35
2.2.11	The 80x86 Instruction Set in HLA	36
2.2.11.1	Zero Operand Instructions (Null Operand Instructions).....	36
2.2.11.2	General Arithmetic and Logical Instructions	36
2.2.11.3	The XCHG Instruction	37
2.2.11.4	The CMP Instruction	37
2.2.11.5	The Multiply Instructions	37
2.2.11.6	The Divide Instructions	38
2.2.11.7	Single Operand Arithmetic and Logical Instructions	38
2.2.11.8	Shift and Rotate Instructions	38
2.2.11.9	The Double Precision Shift Instructions.....	38
2.2.11.10	The Lea Instruction.....	39
2.2.11.11	The Sign and Zero Extension Instructions.....	39
2.2.11.12	The Push and Pop Instructions	39
2.2.11.13	Procedure Calls.....	39
2.2.11.14	The Ret Instruction	40
2.2.11.15	The Jmp Instructions.....	40
2.2.11.16	The Conditional Jump Instructions.....	40
2.2.11.17	The Conditional Set Instructions	40
2.2.11.18	The Input and Output Instructions.....	41
2.2.11.19	The Interrupt Instruction.....	41
2.2.11.20	Bound Instruction	41
2.2.11.21	The Enter Instruction	41
2.2.11.22	CMPXCHG Instruction	41
2.2.11.23	The XADD Instruction	42
2.2.11.24	BSF and BSR Instructions.....	42
2.2.11.25	The BSWAP Instruction	42
2.2.11.26	Bit Test Instructions.....	42
2.2.11.27	Floating Point Instructions.....	42
2.2.11.28	MMX and SSE Instructions.....	42
2.2.12	Memory Addressing Modes in HLA.....	42
2.2.13	Type Coercion in HLA.....	44
3	Installing HLA	45
3.3.1	Installing HLA Under Windows	45
3.3.1.1	New Easy Installation:.....	45
3.3.1.2	Manual Installation under Windows.....	45
3.3.1.2.1	What You've Just Done	46
3.3.1.2.2	Running HLA.....	49
3.3.1.3	Standard Configurations Under Windows.....	52
3.3.2	Installing HLA Under Linux, Mac OSX, or FreeBSD (*NIX).....	54
3.3.2.1	Standard Configurations under Linux/FreeBSD/Mac OSX	57
3.3.3	Non-Standard Configurations under Windows and Linux.....	57
3.3.4	Customizing HLA	57

3.3.4.1	Changing the Location of HLA	58
3.3.4.2	Setting Auxiliary Paths	59
3.3.4.3	Setting the Default Back-End Assembler	59
4	Using HLA with the HIDE Integrated Development Environment	1
4.4.1	The HLA Integrated Development Environment (HIDE)	1
4.4.1.1	Description	1
4.4.1.2	Operation	1
4.4.1.3	First Execution	1
4.4.1.4	The Windows	1
4.4.1.4.1	Editor	2
4.4.1.4.2	Output	2
4.4.1.4.3	Tool Bar	2
4.4.1.4.4	Tab Bar	2
4.4.1.4.5	Status Bar	2
4.4.1.4.6	Panel	2
4.4.1.4.7	Project Panel	3
4.4.1.4.8	Properties	4
4.4.1.5	Compiling Simple Programs	4
4.4.1.6	Menus	4
4.4.1.6.1	Edit	4
4.4.1.6.2	View	5
4.4.1.6.3	Project	6
4.4.1.6.4	Make	6
4.4.1.6.5	Tools	7
4.4.1.6.6	Options	9
4.4.1.6.7	HIDE Settings	10
4.4.1.6.8	SetPaths	12
4.4.1.6.9	User	13
4.4.1.6.10	Help	14
4.4.1.7	HIDE Macros	14
4.4.1.8	Project Manager	14
4.4.1.9	Auto Completion	17
4.4.1.10	CommandLine Tools	18
4.4.1.10.1	kMake	18
4.4.1.11	Project File Format	18
4.4.1.12	Licences	22
4.4.1.12.1	HIDE	22
4.4.1.12.2	PellesC	23
4.4.1.12.3	HLA	23
4.4.2	The RadASM/HLA Integrated Development Environment	24
4.4.2.1	Integrated Development Environments	24
4.4.2.2	HLA Project Organization	24
4.4.2.3	Using Makefiles	25
4.4.2.4	Installing RadASM	31
4.4.2.5	Running RadASM	31
4.4.2.6	The RadASM Project Management Window	32

4.4.2.7	Compiling and Executing an Existing RadASM Project.....	38
4.4.2.8	Creating a New Project in RadASM.....	41
4.4.2.9	Working With RadASM Projects.....	48
4.4.2.10	Build Options with RadASM/HLA.....	50
4.4.2.11	Editing HLA Source Files Within RadASM.....	55
4.4.2.12	Managing Complex Projects with RadASM.....	59
4.4.2.13	Project Maintenance with Batch Files.....	60
4.4.2.14	Project Maintenance with Make Files.....	61
4.4.2.15	RadASM Menus.....	64
4.4.2.15.1	The RadASM File Menu.....	64
4.4.2.15.2	Edit Menu Items.....	67
4.4.2.15.3	The View Menu.....	67
4.4.2.15.4	Format Menu.....	68
4.4.2.15.5	The Project Menu.....	68
4.4.2.15.6	Make Menu.....	72
4.4.2.15.7	The Tools Menu.....	72
4.4.2.15.8	The Window Menu.....	72
4.4.2.15.9	The Option Menu.....	72
4.4.2.16	Customizing RadASM.....	74
4.4.2.16.1	The RADASM.INI Initialization File.....	74
4.4.2.16.2	The HLA.INI Initialization File.....	77
5	HLA Internal Operation.....	84
6	Using the HLA Command-Line Compiler.....	86
7	HLA v2.x Language Reference Manual.....	93
7.7.1	HLA Language Elements.....	93
7.7.2	Comments.....	93
7.7.3	Special Symbols.....	93
7.7.4	Reserved Words.....	93
7.7.5	External Symbols and Assembler Reserved Words.....	100
7.7.6	HLA Identifiers.....	100
7.7.7	External Identifiers.....	100
7.7.8	HLA Literal Constants.....	101
8	HLA Data Types.....	102
8.8.1	Data Types in HLA.....	102
8.8.2	Native (Primitive) Data Types in HLA.....	102
8.8.2.1	Enumerated Data Types.....	103
8.8.2.2	HLA Type Compatibility.....	104
8.8.3	Composite Data Types.....	105
8.8.4	Array Data Types.....	105
8.8.5	Union Data Types.....	105
8.8.6	Record Data Types.....	106
8.8.7	Pointer Types.....	111
8.8.8	Thunks.....	112
8.8.9	Class Types.....	114
8.8.10	Regular Expression Types.....	114
9	HLA Literal Constants and Constant Expressions.....	115

9.9.1	HLA Literal Constants	115
9.9.1.1	Numeric Constants.....	115
9.9.1.1.1	Decimal Constants	115
9.9.1.1.2	Hexadecimal Constants.....	115
9.9.1.1.3	Binary Constants.....	116
9.9.1.1.4	Numeric Set Constants.....	116
9.9.1.1.5	Real (Floating-Point) Constants.....	116
9.9.1.2	Boolean Constants	117
9.9.1.3	Character Constants.....	117
9.9.1.4	Unicode Character Constants	117
9.9.1.5	String Constants.....	117
9.9.1.6	Unicode String Constants	117
9.9.1.7	Character Set Constants.....	118
9.9.2	Structured Constants	118
9.9.2.1	Array Constants	118
9.9.2.2	Record Constants.....	119
9.9.2.3	Union Constants	120
9.9.2.4	Pointer Constants.....	123
9.9.2.5	Regular Expression Constants	123
9.9.3	Constant Expressions in HLA.....	124
9.9.3.1	Type Checking and Type Promotion.....	124
9.9.3.2	Type Coercion in HLA.....	125
9.9.3.3	!expr.....	126
9.9.3.4	- expr (unary negation operator).....	127
9.9.3.5	expr1 * expr2.....	128
9.9.3.6	expr1 div expr2.....	129
9.9.3.7	expr1 mod expr2.....	129
9.9.3.8	expr1 / expr2.....	129
9.9.3.9	expr1 << expr2.....	130
9.9.3.10	expr1 >> expr2.....	130
9.9.3.11	expr1 + expr2.....	130
9.9.3.12	expr1 - expr2.....	130
9.9.3.13	Comparisons (=, ==, <>, !=, <, <=, >, and >=)	131
9.9.3.14	expr1 & expr2.....	131
9.9.3.15	expr1 in expr2.....	131
9.9.3.16	expr1 expr2.....	131
9.9.3.17	expr1 ^ expr2.....	131
9.9.3.18	(expr).....	132
9.9.3.19	[comma_separated_list_of_expressions].....	132
9.9.3.20	record_type_name : [comma separated list of field expressions].....	132
9.9.3.21	identifier.....	132
9.9.3.22	identifier1.identifier2 {...}	132
9.9.3.23	identifier [index_list].....	133
10	HLA Program Structure and Organization.....	134
10.10.1	HLA Program Structure	134
10.10.2	The HLA Declaration Section.....	135

10.10.2.1	The HLA LABEL Declaration Section	135
10.10.2.2	The HLA CONST Declaration Section	142
10.10.2.3	The HLA VAL Declaration Section and the Compile-Time "?" Statement..	146
10.10.2.4	The HLA TYPE Declaration Section	150
10.10.2.4.1	typeID.....	151
10.10.2.4.2	newTypeID : typeID;	152
10.10.2.4.3	newTypeID : typeID [list_of_array_bounds];.....	152
10.10.2.4.4	newTypeID : procedure (<<optional_parameter_list>>);.....	153
10.10.2.4.5	newTypeID : record <<record_field_declarations>> endrecord;	153
10.10.2.4.6	newTypeID : union <<union_field_declarations>> endunion;.....	153
10.10.2.4.7	newTypeID : class <<class_field_declarations>> endclass;.....	153
10.10.2.4.8	newTypeID : pointer to typeID;.....	153
10.10.2.4.9	newTypeID : enum{ <<list_of_enumeration_identifiers>> };.....	153
10.10.2.5	The HLA VAR Declaration Section.....	153
10.10.2.6	The HLA STATIC Declaration Section	160
10.10.2.7	The HLA STORAGE Declaration Section.....	164
10.10.2.8	The HLA READONLY Declaration Section	165
10.10.2.9	The HLA PROC Declaration Section.....	167
10.10.2.10	THE HLA NAMESPACE Declaration Section	167
11	HLA Procedure Declarations and Procedure Calls.....	171
11.11.1	Procedure Declarations	171
11.11.1.1	Original Style Procedure Declarations	171
11.11.1.2	"New Style" Procedure Declarations.....	175
11.11.2	Overloaded Procedure/Iterator/Method Declarations	177
11.11.3	The <code>_vars_</code> and <code>_parms_</code> Constants and the <code>_display_</code> Array	182
11.11.4	External Procedure Declarations.....	183
11.11.5	Forward Procedure Declarations.....	184
11.11.6	Setting Default Procedure Options.....	185
11.11.7	Disabling HLA's Automatic Code Generation for Procedures.....	186
11.11.8	Procedure Calls and Parameters in HLA.....	191
11.11.9	Calling HLA Procedures	192
11.11.10	Parameter Passing in HLA, Value Parameters.....	193
11.11.10.1	Passing Byte-Sized Parameters by Value	194
11.11.10.2	Passing Word-Sized Parameters by Value	198
11.11.10.3	Passing Double-Word-Sized Parameters by Value	200
11.11.10.4	Passing Quad-Word-Sized Parameters by Value	200
11.11.10.5	Passing Tbyte-Sized Parameters by Value	201
11.11.10.6	Passing Lword-Sized Parameters by Value.....	201
11.11.10.7	Passing Large Parameters by Value	202
11.11.11	Parameter Passing in HLA, Reference, Value/Result, and Result Parameters ..	203
11.11.12	Untyped Reference Parameters	207
11.11.13	Pass by Value/Result and Pass by Result Parameters.....	208
11.11.14	Parameter Passing in HLA, Name and Lazy Evaluation Parameters.....	213
11.11.15	Hybrid Parameter Passing in HLA.....	215

11.11.16	Parameter Passing in HLA, Register Parameters	216
11.11.17	Instruction Composition and Parameter Passing in HLA	216
11.11.18	Lexical Scope	218
12	HLA Classes and Object-Oriented Programming	222
12.12.1	Class Data Types	222
12.12.2	Classes, Objects, and Object-Oriented Programming in HLA	222
12.12.3	The THIS and SUPER Reserved Words	223
12.12.4	Class Procedure and Method Prototypes	225
12.12.5	Inheritance	228
12.12.6	Abstract Methods	232
12.12.7	Classes versus Objects	232
12.12.8	Initializing the Virtual Method Table Pointer	233
12.12.9	Creating the Virtual Method Table	234
12.12.10	Calling Methods and Class Procedures	234
12.12.11	Non-object Calls of Class Procedures	236
12.12.12	Static Class Fields	237
12.12.13	Taking the Address of Class Procedures, Iterators, and Methods	239
12.12.14	Program Unit Initializers and Finalizers	240
13	The HLA Compile-Time Language	245
13.13.1	HLA Compile-Time Language, Macros, and Pragmas	245
13.13.2	Viewing the Output of the HLA Compile-Time Language	245
13.13.3	#linker Directive	246
13.13.4	The #Include Directive	246
13.13.5	The #IncludeOnce Directive	247
13.13.6	Macros	248
13.13.6.1	Standard Macros	248
13.13.6.2	Where You Declare a Macro Affects its Visibility	251
13.13.6.3	Multi-part (Context Free) Macro Invocations:	252
13.13.6.4	Macro Invocations and Macro Parameters:	256
13.13.6.5	Processing Macro Parameters	257
13.13.7	Built-in Functions:	259
13.13.8	Constant Type Conversion Functions	260
13.13.8.1	Bitwise Type Transfer Functions	261
13.13.8.2	General functions	261
13.13.8.3	String functions:	265
13.13.8.4	String/Pattern matching functions	266
13.13.8.5	Symbol and constant related functions and assembler control functions	272
13.13.8.6	Pseudo-Variables	277
13.13.8.7	Text emission functions	280
13.13.8.8	Miscellaneous Functions	280
13.13.9	#Text and #endtext Text Collection Directives	281
13.13.10	#String and #endstring Text Collection Directives	281
13.13.11	Regular Expression Macros and the @match/@match2 Functions	281
13.13.11.1	#regex.#endregex	283
13.13.11.2	The #return Clause	283
13.13.11.3	Regular Expression Elements	284

13.13.11.4	Kleene Star, Plus, and Numeric Range Specifications	284
13.13.11.5	Matching Characters in a Regular Expression.....	285
13.13.11.6	Case-insensitive Character Matching in a Regular Expression.....	286
13.13.11.7	Negated Character Matching.....	286
13.13.11.8	String Matching in Regular Expressions.....	286
13.13.11.9	Case-insensitive String Matching in Regular Expressions.....	287
13.13.11.10	Negated String Matching.....	287
13.13.11.11	String List Matching.....	288
13.13.11.12	Character Set Matching in a Regular Expression.....	288
13.13.11.13	Negated Character Set Matching.....	289
13.13.11.14	Matching Arbitrary Characters.....	289
13.13.11.15	Sequences (Concatenation) - The ‘,’ Operator	289
13.13.11.16	Alternation - The " " Operator	289
13.13.11.17	Subexpressions - The "(" operator.....	290
13.13.11.18	Extracting Substrings - The Extraction Operator "<>:"	291
13.13.11.19	Invoking Other #regex Macros in a Regular Expression.....	291
13.13.11.20	Lookahead (peeking).....	292
13.13.11.21	Utility Matching Functions.....	292
13.13.11.22	Backtracking.....	294
13.13.11.23	Lazy Versus Greedy Evaluation.....	295
13.13.11.24	The @match and @match2 Functions.....	296
13.13.11.25	Compiling and Precompiling Regular Expressions.....	297
13.13.11.26	The #match..#endmatch Block.....	298
13.13.11.27	Using Regular Expressions in Your Assembly Programs	299
13.13.12	The #asm..#endasm and #emit Directives.....	299
13.13.13	The #system Directive.....	300
13.13.14	The #print and #error Directives	301
13.13.15	Compile-Time File Output (#openwrite, #append, #write, #closewrite)	301
13.13.16	Compile-time File Input (#openread, @read, #closeread).....	302
13.13.17	The Conditional Compilation Statements (#if).....	302
13.13.18	The Compile-Time Loop Statements (#while and #for).....	303
13.13.19	Compile-Time Functions (macros).....	305
13.13.20	Sample Macro: A Modified IF..ELSE..ENDIF Statement.....	306
13.13.21	Text Processing, Lexical Analysis and the #text..#endtext Block	309
14	HLA Language Reference and User Manual.....	321
14.14.1	High Level Language Statements	321
14.14.2	Exception Handling in HLA:try..exception..endtry	321
14.14.3	Exception Handling in HLA:try..always..endtry.....	326
14.14.4	Exception Handling in HLA:raise.....	327
14.14.5	IF..THEN..ELSEIF..ELSE..ENDIF Statement in HLA.....	328
14.14.6	Boolean Expressions for High-Level Language Statements.....	329
14.14.7	WHILE..WELSE..ENDWHILE Statement in HLA	333
14.14.8	REPEAT..UNTIL Statement in HLA	334
14.14.9	The FOR..ENDFOR Statement in HLA.....	334
14.14.10	The FOREVER..ENDFOR Statement in HLA	336
14.14.11	The BREAK and BREAKIF Statements in HLA	336

14.14.12	The CONTINUE and CONTINUEIF Statements in HLA.....	336
14.14.13	The BEGIN..END, EXIT, and EXITIF Statements in HLA.....	337
14.14.14	The SWITCH/CASE/DEFAULT/ENDSWITCH Statement in HLA.....	339
14.14.15	The JT and JF Medium Level Instructions in HLA	341
14.14.16	Iterators and the HLA Foreach Loop	342
15	HLA Units and External Compilation	345
15.15.1	HLA Units and External Compilation.....	345
15.15.2	External Declarations	345
15.15.3	HLA Naming Conventions and Other Languages	347
15.15.4	HLA Calling Conventions and Other Languages	348
15.15.5	Calling Procedures Written in a Different Language.....	349
15.15.6	Calling HLA Procedures From Another Language.....	349
15.15.7	Linking in Code Written in Other Languages.....	349
15.15.8	Calling HLA Code From Other Languages	349
15.15.9	Exercising Complete Control with HLA.....	356
15.15.9.1	Overhead Present in an HLA Program	357
15.15.9.1.1	The "empty" Program	357
15.15.9.2	The empty Program, Part II	362
15.15.9.3	Overhead Associated With Exceptions	364
15.15.9.4	Overhead Associated with Procedures, Iterators, and Methods	371
15.15.9.5	Overhead Associated with Procedure Calls.....	379
15.15.9.6	Bloat in the HLA Standard Library	384
15.15.9.7	Taking Control with HLA Units.....	384
15.15.9.8	Hello World, Revisited	387
16	The HLA Memory Model and Memory Addressing Modes	390
16.16.1	The HLA Memory Model	390
16.16.2	Memory Addressing Modes in HLA.....	390
16.16.3	Type Coercion in HLA.....	394
17	HLA v2.x Language Reference Manual	397
17.17.1	The 80x86 Instruction Set in HLA.....	397
17.17.2	Zero Operand Instructions (Null Operand Instructions)	398
17.17.3	General Arithmetic and Logical Instructions	402
17.17.4	The XCHG Instruction	403
17.17.5	The CMP Instruction.....	404
17.17.6	The Multiply Instructions.....	404
17.17.7	The Divide Instructions.....	406
17.17.8	Single Operand Arithmetic and Logical Instructions.....	408
17.17.9	Shift and Rotate Instructions	409
17.17.10	The Double Precision Shift Instructions	409
17.17.11	The Lea Instruction	410
17.17.12	The Sign and Zero Extension Instructions	411
17.17.13	The Push and Pop Instructions	411
17.17.14	Procedure Calls	412
17.17.15	The Ret Instruction.....	414
17.17.16	The Jmp Instructions	414
17.17.17	The Conditional Jump Instructions	415

17.17.18	The Conditional Set Instructions	415
17.17.19	The Conditional Move Instructions	415
17.17.20	The Input and Output Instructions	416
17.17.21	The Interrupt Instruction	416
17.17.22	Bound Instruction	416
17.17.23	The Enter Instruction	417
17.17.24	CMPXCHG Instruction	417
17.17.25	CMPXCHG8B Instruction	418
17.17.26	The XADD Instruction	418
17.17.27	BSF and BSR Instructions	419
17.17.28	The BSWAP Instruction	419
17.17.29	Bit Test Instructions	419
17.17.30	Floating Point Instructions	420
17.17.31	Additional Floating-Point Instructions for Pentium Pro and Later Processors	423
17.17.32	MMX Instructions	423
17.17.33	SSE Instructions	425
17.17.34	OS/Privileged Mode Instructions	429
17.17.35	Other Instructions and features	431
18	Advanced HLA Programming	433
18.18.1	Writing a DLL in HLA	433
18.18.1.1	Creating a Dynamic Link Library	433
18.18.1.2	Linking and Calling Procedures in a Dynamic Link Library	436
18.18.1.3	Going Farther	437
18.18.2	Compiling HLA	438
18.18.3	Code Generation for HLA HLL Control Structures	440
18.18.3.1	The HLA Standard Library	440
18.18.3.2	Compiling to MASM Code -- The Final Word	441
18.18.3.3	The HLA if..then..endif Statement, Part I	446
18.18.3.4	Boolean Expressions in HLA Control Structures	447
18.18.3.5	The JT/JF Pseudo-Instructions	453
18.18.3.6	The HLA if..then..elseif..else..endif Statement, Part II	453
18.18.3.7	The While Statement	457
18.18.3.8	repeat..until	459
18.18.3.9	for..endfor	459
18.18.3.10	forever..endfor	459
18.18.3.11	break, breakif	459
18.18.3.12	continue, continueif	460
18.18.3.13	begin..end, exit, exitif	460
18.18.3.14	foreach..endfor	460
18.18.3.15	try..unprotect..exception..anyexception..endtry, raise	460
18.18.4	A Modified IF..ELSE..ENDIF Statement	461
18.18.5	Object Oriented Programming in Assembly	468
18.18.5.1	Hoopla and Hyperbole	468
18.18.5.2	Some Basic Definitions	468
18.18.5.3	OOP Language Facilities	469

18.18.5.4	Classes in HLA	469
18.18.5.5	Objects	471
18.18.5.6	Inheritance	473
18.18.5.7	Overriding	473
18.18.5.8	Virtual Methods vs. Static Procedures	474
18.18.5.9	Writing Class Methods, Iterators, and Procedures	476
18.18.5.10	Object Implementation	479
18.18.5.10.1	Virtual Method Tables	482
18.18.5.10.2	Object Representation with Inheritance	484
18.18.5.11	Constructors and Object Initialization	487
18.18.5.12	Dynamic Object Allocation Within the Constructor	488
18.18.6	Compiling Resource Scripts Using HLA	491
18.18.6.1	The Motivation	491
18.18.6.2	The HLA Solution	491
18.18.6.3	The Resource..Endresource Declaration Section	492
18.18.7	Structures in Assembly Language Programs	493
18.18.7.1	What is a Record (Structure)?	493
18.18.7.2	Record Constants	494
18.18.7.3	Arrays of Records	495
18.18.7.4	Arrays and Records as Record Fields	495
18.18.7.5	Controlling Field Offsets Within a Record	496
18.18.7.6	Aligning Fields Within a Record	497
18.18.7.7	Using Records/Structures in an Assembly Language Program	499
18.18.7.8	Implementing Structures in an Assembler	500

1 HLA Overview

HLA, the High Level Assembler, is a vast improvement over traditional assembly languages. With HLA, programmers can learn assembly language faster than ever before and they can write assembly code faster than ever before. John Levine, comp.compilers moderator, makes the case for HLA when describing the PL/360 machine specific language:

1999/07/11 19:36:51, the moderator wrote:

"There's no reason that assemblers have to have awful syntax. About 30 years ago I used Niklaus Wirth's PL360, which was basically a S/360 assembler with Algol syntax and a little syntactic sugar like while loops that turned into the obvious branches. It really was an assembler, e.g., you had to write out your expressions with explicit assignments of values to registers, but it was nice. Wirth used it to write Algol W, a small fast Algol subset, which was a predecessor to Pascal. ... -John"

PL/360, and variants that followed like PL/M, PL/M-86, and PL/68K, were true "mid-level languages" that let you work down at the machine level while using more modern control structures (i.e., those loosely based on the PL/I language). Although many refer to "C" as a "medium-level language", C truly is high level when compared with languages like PL/*. The PL/* languages were very popular with those who needed the power of assembly language in the early days of the microcomputer revolution. While it's stretching the point to say that PL/M is "really an assembler," the basic idea is sound. There really is no reason that assemblers have to have an awful syntax.

HLA bridges the gap between very low level languages and very high level languages. Unlike the PL/* languages, HLA really is an assembly language. You can do just about anything with HLA that you can do with a traditional assembler like MASM, TASM, NASM, or Gas. If you want to write low-level assembly code using x86 machine instructions, HLA does not get in your way; if you want to use compares and conditional branches rather than structured control statements, you can. On the other hand, if you prefer to use more readable high-level control structures, HLA allows this, as well. HLA lets you work at the level you are most comfortable with and at the level that is most appropriate for the task at hand.

Beyond supplying a "non-awful" syntax, HLA has one other important feature -- it's extensible. HLA provides special features that let you add new statements to the language. So if HLA is not "high level" (or "low level") enough for your tastes, you can extend it. This document will expend considerable effort describing exactly how to do this in a later section.

In addition to the HLA language itself, the HLA system provides one other very important component - the HLA Standard Library. This is a collection of hundreds of functions that you can use to write assembly language programs as quickly and easily as you would write C programs.

1.1 What is a "High Level Assembler"?

The name "High Level Assembler" and its abbreviation "HLA" is certainly not new¹. Nor is the concept of a *high level assembler*. David Salomon in his 1992 text "Assemblers and Loaders" (Ellis Horwood, ISBN 0-13-052564-2) uses these terms to describe various assembly languages dating back to 1966. Furthermore, both IBM and Motorola have assembler products with very similar names (e.g., IBM's HLAsm, though it's somewhat debatable whether HLAsm is truly a high level assembler).

Salomon offers the following definitions for a High Level Assembler (or HLA):

*A high-level assembler language (HLA) is a programming language where each instruction is translated into a few machine instructions. The translator is somewhat more complex than an assembler, but much simpler than a compiler. Such a language should not have features like the **if**, **for**, and case control structures,*

1. This section will use the term "HLA/86" when specifically taking about the High Level Assembler product this documentation describes and use "HLA" as a generic term. After this section, this documentation will use the term "HLA" to specifically describe the "HLA/86" product.

complex arithmetic, logical expressions, and multi-dimensional arrays. It should consist of simple instructions, closely resembling traditional assembler instructions, and of a few simple data types.

Since Salomon describes a couple of high level assemblers that exceed this definition, he offers a second definition for high level assemblers that is a bit higher-level:

A high-level assembler language (HLA) is a language that combines most of the features of higher-level languages (easy to use control structures, variables, scope, data types, block structure) with one important feature of assembler languages namely, machine dependence.

Neither definition is particularly useful for describing HLA/86 and other HLAs like Terse, MASM and TASM. Of course the term "High Level Assembler" is very nebulous and offers a fair amount of latitude. Almost any macro assembler could pass as an HLA on the basis that a macro-instruction expands into a few machine instructions.

David Salomon describes several different high level assemblers in his text. The examples he describes are PL/360, NEAT/3, PL516, and BABBAGE.

PL/360 and PL516 are products that conform to the second definition above. They allow simple arithmetic expressions and assignment statements, the use of high level control structures (**if**, **for**, **while**, etc.), high level data declarations, and block structure (among other things). These languages expose the underlying machine's registers and allow the use of machine instructions using a "functional" syntax.

The NEAT/3 language is a much lower-level language; basically it is an assembly language for the NCR Century computers that provide COBOL-style data declarations. Most of its "instructions" translate one-for-one into Century machine instructions, though it does automatically insert code to convert data types from one format to another if the data types of an instruction's operands are incompatible.

The BABBAGE assembly language is an expression-based assembly language (very similar to Terse). It allows simplified high level control structures like **if** and **while**. The interesting thing about this assembler is that it was the only assembler for the GEC4000 family of computers.

In addition to the HLAs that Salomon describes, there have been several other high level assemblers created over the years. PL/M and PL/M-86 was designed by Intel for their 8080 and 8086 CPU families. This was an obvious adaptation of the PL/360 style HLA for Intel's CPUs. PL/68 was also available for the Motorola 680x0 family. SL/65 was a similar adaptation of PL/360 for the 6502 family. At one point there was a product named "High Level Assembler" for the Atari ST system (68K based). Jim Neil has also created an expression-based high level assembler (similar in principle to Babbage) for Intel's x86 family. MASM and TASM (for the x86) also fall into the category of a high level assembler due to their inclusion of high level control structures and logical expressions.

So where does HLA/86 fit into these definitions? In truth, the definition of HLA/86 falls somewhere between these two definitions. So the following paragraphs will define the term "High Level Assembler" as it should apply to HLA/86 and similar high level assemblers.

The first definition above is overly restrictive. It implies that any language that exceeds these limits is a high level language, not a high level assembly or traditional assembly language. Obviously, this definition is too restrictive in the sense that by this definition many traditional assemblers would have to be considered as high level languages (even beyond a high level assembler). Furthermore, it elevates many traditional assemblers to the status of an HLA even though we wouldn't normally think of them as high level assemblers; i.e., most macro assemblers provide the ability to create instructions that translate into a few machine instructions. Macro facilities, however, are something we expect out of a modern assembly language; their presence doesn't make the language a "high level" assembly language in most people's mind. Furthermore, most modern assemblers provide a mechanism for declaring multi-dimensional arrays (even though you still have to use some sequence of instructions to index into said arrays).

The second definition David Salomon provides hits the other extreme. Arguably, languages like C could be called HLAs under this definition (yes, there are some machine dependent features in C, though probably not enough to satisfy David Salomon's original intent).

The definition of high level assemblers like Terse, MASM, TASM, and HLA/86 fall somewhere between these extremes. Therefore, this document will define a high level assembler as follows:

A "high level assembly language" (HLAL) is a language that provides a set of statements or instructions that practically map one-to-one to machine instructions of the underlying architecture. The HLAL exposes the underlying machine architecture

*including access to machine registers, flags, memory, I/O, and addressing modes. Any operation that is possible with a traditional assembler should be possible within the HLA. In addition to providing access to the underlying architecture, the HLA must provide some abstractions that are not normally found in traditional assemblers and that are typically found in traditional high level languages; this could include structured control statements (e.g., **if**, **for**, and **while**), high level data types and data structuring facilities, extensive compile-time language facilities, run-time expression evaluation, and standard library support. A "High Level Assembler" is a translator that converts a high level assembly language to machine code.*

There is a very important difference between this definition and the ones that David Salomon provides. Specifically, a high-level assembly language must provide access to the underlying machine architecture. Within the HLA you must be able to specify any (reasonable) machine instruction that is available on the CPU. The HLA may provide other statements that do not directly map to machine instructions (e.g., an **if** statement), but it must, at least, provide a set of statements that *practically* map one-to-one with the machine instructions. The "practically" modifier appears here for two reasons. First of all, some assembly source statements may map to two or more different, but equivalent, machine instructions. A good example is the x86 "mov reg, reg" which can map to two different (though equivalent) opcodes depending on the setting of the direction bit in the opcode. Most assemblers will map the source statement to only one of these opcodes, hence there is not truly a one-to-one mapping (since there exist some opcodes that do not map back to some source instruction). Another allowable restriction is that the HLA may limit the programmer to a subset of the complete machine instruction set if it makes sense to do so (e.g., many modern x86 assemblers do not support 16-bit mode on the 80x86).

In addition to supporting the underlying machine architecture (which almost any traditional assembler will do), the HLA must also provide support for some features normally found in a high level language. The definition does not require that a HLA support all the features listed above, nor is it restricted to just the features listed, but a HLA must support some of the features traditionally found in a high level language. The number and type of features the HLA supports determines how "high level" the assembly language is. Like HLLs, we can have "low-level" HLAs, "medium-level" HLAs, "high-level" HLAs, and even "very high-level" HLAs. NEAT/3, for example, would be a low-level HLA since it provides higher-level data types, conversions, and not much else.

MASM and TASM are probably best considered medium-to-high-level HLAs since they provide high level data structuring facilities, structured control statements, high level procedure definitions and invocations, a limited block structure, powerful compile-time language (macro) facilities, standard library support (e.g., the UCR Standard Library and many other available library modules), and other high level language features. In actual use, the programmer is expected to normally use standard machine instructions and rise up to the high level statements only as necessary.

The Terse language is a good example of a medium level HLA since it uses an expression syntax but otherwise maps statements fairly closely to the assembly counterparts. It does provide some higher-level data structuring capabilities, though this is inherited from the underlying assembler(s) on which Terse is based.

PL/360 and PL/516 are definitely high-level HLAs because they fully support simplified arithmetic expressions, control structures, high-level data types, and other features. These languages provide access to the underlying architecture, but the emphasis is to use these languages as a high level language and drop down to the machine instructions only as necessary.

HLA/86 probably falls in the high-level-to-very-high-level range because it provides high level data types and data structuring abilities, high level and very high level control structures, extensive parameter passing facilities (more than most high level languages), a very extensive compile time language, a very extensive standard library, built-in parsing facilities for language extension, and many other features. Generally, HLA/86 has a larger feature set than the other HLAs described above. There are a few design goals that limit the "high-levelness" of HLA/86:

- (1) With one exception, HLA never emits any code behind the programmer's back that modifies registers or flags (the one exception is object method invocation, and this is well documented), and

- (2) HLA doesn't support arithmetic expressions (it does support a limited form of logical/boolean expressions).

One interesting aspect of HLA/86 is that it is extensible. Using features built into the language, you can extend HLA/86's syntax by adding new statements and other features. This

feature gives you the ability to make HLA/86 as high level as you desire (though it may take some effort to achieve certain language features). The bottom line is this: in some ways, HLA/86 is lower level than languages like PL/360 and PL516; in other ways, it's higher level than these HLALs. However, as the definition requires, almost anything you can do with a traditional assembler is possible in HLA/86.

1.2 What is an "Assembler"

Because high-level assemblers are clearly different than traditional assemblers, one might question whether a high level assembly language is truly an assembly language and whether translators for high-level assembly languages can be properly called an assembler. Unfortunately, there is a considerable range of opinions as to exactly what constitutes an "assembler" versus other translators. This document will not attempt to get involved in this debate. Instead, this section provides a set of definitions that are useful for describing assemblers at various levels of abstraction.

Pure Assembler:

A "pure assembler" is a program that processes an assembly language source file and translates the source code using a direct mapping from source code instructions to individual machine instructions (each source instruction is mapped to exactly one machine instruction). The assembler only provides machine-primitive data types like bytes, words, double words, etc. A pure assembler does not provide macro facilities. A pure assembler always produces machine code as output.

Traditional Assembler:

A "traditional assembler" is a pure assembler plus macro facilities. The assembler may provide some "built-in macros" and instruction synonyms, but in general, the built-in statements should still map to individual machine instructions (note that the programmer may extend this by writing macros). There is no support by the assembler for run-time arithmetic or boolean expressions. A traditional assembler may also provide some simple data typing facilities (such as the ability to rename primitive data types as something else, e.g., byte->char). A traditional assembler always emits machine code as output.

High Level Assembler:

A high-level assembler is a macro assembler plus some additional high-level language-like facilities, such as high-level control constructs or high-level-like procedure calls. If a programmer elects to ignore these additional facilities, they still have all the capabilities of a macro assembler at their disposal.

1.3 Is HLA a True Assembly Language?

Some people are confused by HLA. On the one hand, it looks like a High Level Language, employing syntax similar to Pascal and C/C++. On the other hand, it does support the machine instructions found in a typical assembly language. Many people accuse HLA of being a compiler rather than an assembler. What's the truth?

The truth is, assembly languages have evolved, just as high-level languages have evolved, and we can no longer use a definition for an assembler that made sense in the 1950s when describing modern assemblers such as MASM, TASM, and HLA. Today, the best definition we can use is that an assembler is a compiler for an assembly language. An assembler accepts a source file written in some sort of assembly language and produces an object file as its output.

The real question, then, is not whether HLA is an assembler, but whether the HLA language is an assembly language. Some people argue that any compiler that includes any sort of statement that compiles into more than one machine instruction cannot be called an "assembler." However, such an argument immediately eliminates macro assemblers. Eliminating macro assemblers is unsatisfactory because almost every modern assembler provides, at the very least, some simple macro facilities. Whether you implement an "IF" statement with a macro (generally supplied by the assembler's author, as is the case, for example, with FASM) that you have to include into your source file, or via a 'macro' that the assembler's author has provided as part of the assembler is really a matter of implementation. To the end user of the assembler, the "IF" statement is just as much a part of the language that they can use regardless of the implementation. The fact that assemblers such as MASM, TASM, and HLA provide these high-level-like control structures in

assembly language does not imply that the languages these products implement are not assembly languages.

Some people argue that "high-level assemblers" such as MASM, TASM, and HLA are not assemblers any more than C/C++ compilers could be considered assemblers if those C/C++ compilers support an in-line assembly capability. However, their arguments strengthen the case for calling a product like HLA an "assembler." After all, if we're going to continue to call C/C++ a high-level language even though it provides support for machine instructions, then there is no reason we cannot call a product like MASM, TASM, or HLA "assemblers" even though they provide a modicum of support for high-level-like control structures. Ultimately, language's focus determines its type. C/C++'s focus is on writing high-level language programs, with a few machine instructions thrown in now and then when the high-level language doesn't quite handle everything. High-level assemblers, such as HLA, MASM, and TASM are focused on writing assembly language modules. They have some high-level control structures thrown in to simplify some tasks (e.g., in the case of HLA, the high-level control structures exists as a bridge between HLLs and assembly during the learning process), but the focus is mainly on writing assembly language code.

Some people feel that if you learn HLA (or some other high level assembler), then you're not really learning "assembly language." This is utter nonsense. If you thoroughly learn HLA, you'll know assembly language programming inside and out. Switching to a different assembler from HLA would be no different, say, than switching from Gnu's Gas assembler to MASM (or vice versa). One might bemoan the features lost in such a translation, but when going from HLA to some other assembler you're typically *giving up* features rather than gaining anything.

Still there is a pervasive argument that high-level control structures like IF/WHILE/FOR/etc. don't belong in a true assembler. Well, HLA, MASM, and TASM users can elect to ignore these statements (as many old-time MASM programmers do; with HLA you can even disable these statements). As long as the rest of the assembler supports a language that allows one to write "pure" assembly language code, why would anyone question the validity of the title "assembly language" for the code? (Unless, of course, they have an ax to grind.) For those who are diametrically opposed to allowing any language that contains IF/WHILE/FOR/etc. statements to be called assembly language, well, that's why we call these things *high level assembly languages*: to note the fact that they are a little more powerful than traditional assembly languages.

The bottom line is this: if you learn HLA, you will learn assembly language programming. As long as you understand how to write the low-level code (within HLA) and don't rely exclusively on the high-level control statements in your programs, no one can truthfully question your assembly language programming knowledge.

1.4 HLA Design Goals

HLA was originally conceived as a tool to teach assembly language programming. In early 1996 I decided to do a Windows version of my electronic text "the Art of Assembly Language Programming" (AoA). After an attempt to develop a new version of the "UCR Standard Library for 80x86 Programmers" (a mainstay of AoA), I came to the conclusion that MASM just wasn't powerful enough to make learning assembly language really easy. I decided to develop an assembler with sufficient power, providing the tools for a good standard library as well as satisfy some other requirements. Therefore, HLA has two important goals: provide a system that is powerful enough to develop code and macros to make learning assembly language, which simultaneously providing a system that is easy for beginners to learn.

The principle goal of HLA was to leverage student's existing programming knowledge. For example, a good Pascal programmer can get their first C/C++ program operational in a few minutes. All they have to do is note the similarities between the two programming languages, make the appropriate syntactical changes, and they're up and running. Take that same Pascal programmer and expect them to learn LISP or Prolog the same way, and you'll not meet with the same success. LISP and Prolog are completely different, they use a different "programming paradigm," so the student has to "start over from scratch" when learning these languages. Although assembly language is an imperative language (like Pascal and C/C++), there is a considerable "paradigm shift" when moving from one of these high level languages to assembly. In HLA, I wanted to create a language with high level control structures and declarations that made it possible for someone familiar with an imperative language like Pascal or C/C++ to get their first HLA program running in a matter of minutes (or, at worst, a matter of hours). Of course, to achieve this goal, I needed to add high-level data declarations and high-level control constructs to the HLA language.

The astute reader will quickly point out that high level control structures are not assembly language and letting the students use these types of statements is not really teaching them assembly

language. This is quite true; since the purpose of teaching an assembly language course is to teach the students *assembly* language programming it is quite clear that HLA would fail if it *only* provided these high level control structures (e.g., like the PL/M language does). Fortunately, this is not the case. HLA supports all standard assembly language instructions including CMP and Jcc instructions, so you can still write "pure" assembly language programs without using those high-level language control structures. However, it does take time to learn the several hundred different machine instructions. Traditionally, it's taken my students (using only MASM) about five weeks before they could really write any meaningful programs in assembly language (you have to cover things like numeric representation, basic CPU architecture, addressing modes, data types, and introduce the instruction set before any real programs can be written).

HLA lets students write *meaningful* programs within about a week of its introduction (e.g., the first assignment I give in a typical quarter is to write an "addition table" program that computes the outer product [addition table] of the two vectors 0..15 and 0..15, printing the table formatted nicely). They achieve this by using statements they already know (like IF and WHILE) with the injection of just a few assembly language concepts (registers, and the MOV and ADD instructions) plus an introduction to the HLA Standard Library. Over the next several weeks, these students write increasingly complex programs as they are introduced to new assembly language and HLA concepts (e.g., data representation, basic architecture, addressing modes, data types, and additional instructions). At about the sixth week, I begin "weaning" these students off the high-level language statements and force them to use the low-level machine instructions. It turns out that they learn how to simulate an IF statement at roughly the same point in the quarter as they did when they used only MASM, but the big difference is that they've written a lot more code up to that point proving out other concepts in machine organization and assembly language programming. In my limited experience with classroom testing, I've found that students spend less time on the class, cover more material, and retain the knowledge better (by the time of the final exam) than they did when I only used MASM.

The general goal of reducing the learning curve for students is achieved several ways.

- (1) As noted above, HLA allows a gradual transition from high-level languages into pure assembly language. My favorite analogy here is the Nicoderm CQ smoking cessation system ("gradual steps are better."). Like the Nicoderm system, HLA allows students learn assembly language in gradual steps rather than throwing them into the water and shouting "sink or swim!"
- (2) In addition to letting the students employ high level language statements in their assembly language programs, HLA contains several other familiar concepts and syntactical items that ease the transition from high-level language programming to assembly language. For example, HLA uses the familiar (to C/C++ programmers) `"/*` and `*/` comment delimiters (as well as the `"/` comment delimiter). Statements generally end with a semicolon (just as in high level languages). Machine instructions use a functional notation rather than "mnemonic-operand" notation. Constant, type, and variable declarations should look very familiar to Pascal programmers. HLA's standard library should look comfortable to anyone who has used the C/C++ standard library.

In addition to syntactical similarities, well-written HLA programs share a similar programming style with modern high-level languages. Therefore, a student who has learned how to write readable Pascal, C/C++, or Java programs will be able to write readable HLA programs with almost no additional study. Contrast this with the style guide I've written for (MASM) assembly language programmers that is quite a bit different than high level languages and takes a while to master.

Another factor many people don't consider is the evaluation of a programming project. At UC Riverside instructors are given about 1.5-2 hours per student per quarter of reader (student grader) time to grade projects. Experienced readers who can grade (or want to grade) assembly language projects are few and far in-between. Most readers are "stuck" with grading the assembly class rather than volunteer for the job. The fact that most student assembly language projects have a horrible programming style and are hard to read only exacerbates this situation. HLA helps solve this problem. Since good HLA programming style is very similar to good C/C++ style, UC Riverside's readers have a much easier time reading the projects and evaluating their programming style. In addition, since the students have (presumably) learned good programming style in the prerequisite course(s), they tend to write easier to read HLA programs than MASM programs. This lets the instructor assign more projects without fear of exceeding my reader budget each quarter.

HLA's advantages are easily summed up by a complaint I had from a student once. She said "HLA drives me nuts. It's so similar to C++ that I often get confused and try out something that would work in C++ only to have the HLA compiler reject it." I agreed with this student that this was a bit of a problem, but I also mentioned, "what about all the times you've tried something from C++ and it HAS worked?" She thought about it for a moment and walked away agreeing with my assessment of her complaint. Had this student been learning assembly the traditional way, she wouldn't have bothered to try anything. She would had to have spent extra time learning how to achieve what she wanted by reading an assembly text or she would have missed out on the opportunity to actually learn something new. HLA's similarity to C++ encouraged her to try something out on her own. The experiments weren't always successful, but in those cases where they were, she benefited greatly from this. This anecdote, more than any other, sums up what my goals with HLA were and describes the success I believe I have achieved with it.

1.5 How to Learn Assembly Programming Using HLA

Of course, a compiler without a language reference manual and tutorial is useless. This document will provide a reference to the HLA programming language. It is not, however, appropriate pedagogy for beginners (it's more suitable for those who already know assembly language programming and wish to learn HLA's syntax). A better text for beginners is "The Art of Assembly Language Programming, Second Edition" available from No Starch Press. This provides a complete college level textbook that teaches assembly language programming from the ground up using HLA. You can also find an electronic copy of "AoA" on Webster at <http://webster.cs.ucr.edu>. Webster also contains the latest version of HLA as well as tons of HLA sample source code. That's the first place you should go for information on learning HLA.

1.6 Legal Notice

The HLA v2.x implementation is a prototype intended to test language design and implementation features. I (Randall Hyde) have placed this code and language design in the public domain so others may benefit from this work. However, keep in mind that, as a prototype, HLA is not up to contemporary commercial standards for software quality. It is your responsibility to evaluate whether HLA is suitable for whatever purpose you have.

At any given time, there are several known and unknown defects in this software. Some may be corrected in later releases of HLA v2.x; some may never be corrected in the v2.x series. I (Randall Hyde) do not warrant or guarantee this software in any way. In particular, you cannot expect corrections of any given defect in the system. Obviously, I try to fix known problems (if possible), but I refuse to be held legally responsible for such defects in the software.

The purpose of developing a prototype implementation of the HLA language was to try out language design and implementation ideas. The prototype phase of HLA development is rapidly coming to an end and an "official" HLA language design will be forthcoming. HLA v3.0 will implement this new language. The only guarantees I make about compatibility between HLA v2.x and HLA v3.0 is that there *will* be some incompatibilities. The exact nature and magnitude of those incompatibilities is unknown at this point, but it is safe to assume that no HLA v2.x program will compile under HLA v3.0 without at least some minor source code changes. So please don't get the idea that any investment you make in HLA source code will be protected in v3.0 (note: after the release of v3.0 this is a relatively safe assumption to make, though there will still be no guarantees).

Because HLA is constantly changing (typical of a prototype), it is very difficult to keep the documentation in phase with the language. You can expect this documentation (and all HLA documentation) to contain omissions (e.g., of new features that have yet to be documented), discussion of features removed from HLA, and incorrect descriptions of HLA features. Every attempt will be made to keep the documentation in phase with the software, but like so many free software projects, lack of time and motivation prevents perfection¹.

This software is not fit for use in mission-critical or life-support software systems. This software is principally intended for evaluation and educational (i.e., learning assembly language) purposes only. It has been successfully used to develop commercial and industrial applications (including a nuclear reactor control system) and it has been successfully used in educational environments, but again, you are personally responsible for determining the fitness of this software and documentation for your particular application and you must take responsibility for that choice.

1. You must admit, though, HLA's documentation is better than that of most free software.

HLA's current design makes use of other software tools that I (Randall Hyde) did not write. These tools include the Microsoft Linker, the Microsoft Librarian, the Pelles C linker, the Pelles C librarian, and the Free Software Foundations *ld* and *as* programs. It can optionally make use of programs such as MASM, FASM, TASM, and NASM. Because some of these tools are commercial products and are covered by various license agreements, not all of these tools come with the HLA distribution. For example, if you want to use the Microsoft or Borland tools, you'll have to obtain copies of them from some other source. Note that using HLA does not require the Microsoft or Borland tools; HLA is simply compatible with these tools if you already own them and would prefer to use them. HLA does ship with all the tools you need to effectively use HLA; the use of these non-free tools is optional.

1.7 Teaching Assembly Language using HLA

I first began teaching assembly language programming at Cal Poly Pomona in the Winter Quarter of 1987. I quickly discovered that good pedagogical material was difficult to come by; even the textbooks available for the course left something to be desired. As a result, my students were learning very little assembly language in the ten weeks available to the course. After about two quarters, I decided to do something about the textbook problem, so I began writing a text I entitled "How to Program the IBM PC Using 8088 Assembly Language" (obviously, this was back in the days when schools still used PCs made by IBM and the main CPU you could always count on was the 8088). "How to Program..." became the epitome of a "work in progress." Each quarter I would get feedback from the students, update the text, and give it to Kinko's (and the UCR Printing and Reprographics Department) to run off copies for my students the very next quarter.

The original "How to Program..." text provided a basic set of library routines to print strings, input characters and lines of text, and a few other basic functions. This allowed the students to quickly begin writing programs without having to learn about the INT instruction, DOS, or BIOS. However, I discovered that students were spending a significant time each quarter writing their own numeric conversion routines, string manipulation routines, etc. One student commented on "how much easier it was to program in 'C' than assembly language since all those conversions and string operations were built into the language." I replied that the real savings were due more to the 'C' standard library than the language itself and that a comparable library for assembly language programmers would make assembly language programming almost as easy as 'C' programming. At that moment a little light went on in my head and I sat down and wrote the first few routines of what ultimately became the "UCR Standard Library for 80x86 Assembly Language Programmers" (You can still get a copy of the UCR stdlib from webster at the URL given above). As I finished each group of routines in the standard library, I incorporated them into my courses. This reaped immediate benefits as students spent less time writing numeric conversion routines and spent more time learning assembly language. My students were getting into far more advanced topics than was possible before the advent of the UCR Stdlib.

In the early 1990's, the 8088 CPU finally died off and IBM was no longer the major supplier of PCs. Not only was it time to change the title of my text, but I also needed to update references to the 8088 (that were specific to that chip) and bring the text into the world of the 80386 and 80486 processors. DOS was still King and 16-bit code was still what everyone was writing, but issues of optimization and the like were a little outdated in the text. In addition to the changes reflecting the new Intel CPUs, I also incorporated the UCR Standard Library into the text since it dramatically improved the speed at which students progressed beyond the basic assembly programming skills. I entitled the new version of the text "The Art of Assembly Language Programming," an obvious knock-off of Knuth's series ("The Art of Computer Programming").

In early 1996 it became obvious to me that DOS was finally dying and I needed to modify "The Art of Assembly Language Programming" (AoA) to use Windows as the development platform. I wasn't interested in having students write Windows GUI applications in assembly language. (The time spent teaching event-oriented programming would interfere with the teaching of basic machine organization and assembly language programming.) However, it was clear that the days of writing code that arbitrarily pokes around in memory and accesses I/O addresses directly (things that AoA taught) were over. Therefore, I decided to get started on a new version of AoA that used Windows as the basic development environment with the emphasis on writing console applications.

The UCR Standard Library was the single most important pedagogical tool I'd discovered that dramatically improved my students' progress. As I began work on a new version of AoA for Windows 3.1 my first task was to improve upon the UCR Standard Library to make it even easier to use, more flexible, more efficient, and more "high level." After six months of part time work, I eventually gave up on the UCR Stdlib v2.0. The idea was right; unfortunately, the tools at my disposal (specifically, MASM 6.11) weren't quite up to the task. I was writing some tricky macros, obviously exploiting code inside MASM that Microsoft's engineers had never run (i.e., I discovered lots of bugs). I would code in some workarounds to the defects only to have the macro package break at the next minor patch of MASM (e.g., from MASM 6.11a to MASM 6.11b).

There was also a robustness issue. Although MASM's macro capabilities are quite powerful and it almost let me do everything I wanted, it was very easy to confuse the macro package. This would cause MASM to generate some totally weird (but absolutely correct) diagnostic messages that correctly described what was going wrong in the macro but made absolutely no sense whatsoever at all. As it became clear that the UCR Stdlib v2.0 would never be robust enough for student use, I decided to take a different approach.

About this time, I was talking with my Department Chair about the assembly language course. We were identifying some of the problems that students had learning assembly language. One problem, of course, was the paradigm shift- learning to solve problems using machine language rather than a high level language. The second problem we identified is that students get to apply very little of what they've learned from other courses to the assembly language class. A third problem was the primitive tools available to assembly language programmers. Energized by this discussion, I decided to see how I could solve these problems and improve the educational process.

Problem 1, the paradigm shift, had to be handled carefully. After all, the whole purpose of having students take an assembly language programming course in the first place is to acquaint them with the low-level operation of the machine. However, I felt it was certainly possible to redefine parts of assembly language so that would be more familiar to students. For example, one might test the carry flag after an addition to determine if an unsigned overflow has occurred using code like the following:

```
add eax, 5

jnc NoOverflow

    << code to execute if overflow occurs >>

NoOverflow:
```

Although this code is straightforward, you would be surprised how many students cannot visualize this code. On the other hand, if you feed them some pseudo code like:

```
add eax, 5

if( the carry flag is set ) then

    << code to execute if overflow occurs >>

endif
```

Those same students won't have any problems understanding this code. To take advantage of this difference in perspective, I decided to explore changing the definition of assembly language to allow the use of the "if condition then do something" paradigm rather than the "if a condition is false then skip over something" paradigm. Fundamentally, this does not change the material the student has to learn; it just presents it from a different point of view to which they're already accustomed. This certainly wasn't a gigantic leap away from assembly language as it existed in 1996. After all, MASM and other assemblers were already allowing statements like ".if" and ".endif" in the code. Therefore, I tried these statements out on a few of my students. What I discovered is that the students picked up the basic "high level" syntax very rapidly. Once they mastered the high level syntax, they were able to learn the low-level syntax (i.e., using conditional jumps) faster than ever before.

The second problem, students not being able to leverage their programming skills from other classes, is largely linked to the syntax of Intel x86 assembly language. Many skills students pick up, such as programming style, indentation, appropriate programming construct selection, etc., are useless in a typical assembly language class. Even skills like commenting and choosing good variable names are slightly different in assembly language programs. As a result, students spend considerable (unproductive) time learning the new "rules of the game" when writing assembly language programs. This directly equates to less progress over the ten-week quarter. Ideally, students should be able to applying knowledge like program style, commenting style, algorithm organization, and control construct selection they learned in a C/C++ or Pascal course to their assembly language programs. If they could, they'd be "up and writing" in assembly language much faster than before.

The third problem with teaching assembly language is the primitive state of the tools. While MASM provides a wonderful set of high level language control constructs, very little else about MASM supports this "brave new world" of assembly language I want to teach. For example, MASM's variable declarations leave a lot to be desired (the syntax is straight out of the 1960's). As I noted earlier, as powerful as MASM's macro facilities are, they weren't sufficient to develop a robust library package for my students. I briefly looked at TASM, but it's "ideal" mode fared little better than MASM. Likewise, while development environments for high-level languages have been improving by leaps and bounds (e.g., Delphi and C++ Builder), assembly language programmers are still using the same crude command line tools popularized in the early 1970's. Codeview, which is practically useless under Windows, was the most advanced tool Microsoft provided specifically for assembly language programmers.

Faced with these problems, I decided the first order of business was to create a new x86 assembly language and write a compiler for it. I decided to give this language the somewhat-less-than-original name of "the High Level Assembler," or HLA (IBM and Motorola both already have assemblers that use a variant of this name). It took three years, but the first version of HLA was ready for public consumption in September of 1999.

I began using HLA in my CS 61 course (machine organization and assembly language programming) at UCR in the Fall Quarter, 1999. With no pedagogical material other than a roughly written reference guide to the language, I was expecting a complete disaster. It turns out that I was pleasantly surprised. Although the students did have major problems, the course went far more smoothly than I anticipated and we managed to cover about the same material I normally covered when using MASM.

Although things were going far better than I expected, this is not to say that things were going great, or even as smoothly as I would have liked. The major problem, of course, was the lack of a textbook. The only material the students had to study from was their lecture notes. Clearly, something needed to be done about this. Of course, the whole reason for spending three years writing HLA was to allow me to write a new version of AoA. Therefore, in November 1999 I began work on the new edition of the text. By the start of the Winter Quarter in January 2000, I had roughed together five chapters, about 50% of the material was brand new and the other 50% was cut, pasted, and updated from the older version of the text. During the quarter, I rushed out two more chapters bringing the total to seven. The Winter Quarter went far more smoothly than the Fall Quarter. Student projects were much better and the progress of the class outstripped any assembly language course I'd taught prior to that point. Clearly, the class was benefiting from the use of HLA.

By the start of the Spring Quarter in April 2000, I'd managed to make one proofreading pass over the first six chapters and I'd written the first draft of the eighth chapter. By the middle of 2002, The Art of Assembly Language was on-line and receiving rave reviews across the internet. In 2003, No Starch Press published an edited and revised edition in "treeware" form.

Well, this has been a long-winded report of HLA's justification. You're probably wondering what HLA is and whether it is applicable to you (especially if you're a programmer rather than an educator). Fair enough, the rest of this article will discuss the HLA system and how you would use it.

HLA (under Windows) is a Win32 console application and it generates Win32 applications. By default, it generates console applications although it does not restrict you to writing console applications under Windows. There is absolutely no support for DOS applications. HLA v2.0 also supports Mac OS X, Linux, and FreeBSD. Applications written in HLA that use the HLA Standard Library can run under all four operating systems with nothing more than a recompile. This allows a student, for example, to work under Windows at home and submit projects under Linux (or any of the other OSes) at school.

When designing the HLA language, I chose a syntax that is very similar to common imperative high-level languages such as Pascal/Delphi, Ada, Modula-2, FORTRAN77, C/C++, and Java. That is not to say that HLA compiles Pascal programs, but rather, a Pascal programmer will note many similarities between Pascal and HLA (and ditto for the other languages). HLA stole many of the ideas for data declarations from the Algol-based languages (Pascal, Modula-2, and Ada), it grabbed the ideas for many of its control structures from FORTRAN77, Ada, and C/C++/Java, and the structure of the HLA Standard Library is based on the C Standard Library. So regardless of which high level language you're most comfortable with in this set, you'll certainly recognize some elements of your favorite HLL in HLA.

A carefully written HLA program will look almost like a high-level language program. Consider the following sample program:

```
program SampleHLApgm;

#include( "stdlib.hhf" )

const

    HelloWorld := "Hello World";

begin SampleHLApgm;

    stdout.put( "The classical 'Hello World' program: ", HelloWorld, nl );

end SampleHLApgm;
```

This program does the obvious thing. Anyone with any high-level language background can probably figure out everything except the purpose of "nl" (which is the newline string imported by the standard library). This certainly doesn't look like an assembly language program; there isn't even a real machine instruction in sight. Of course, this is a trivial example; nonetheless, I've managed to write reasonable HLA programs that were just over 1,000 lines of code that contained only one or two identifiable machine language instructions. If it's possible to do this, how can I get away with calling HLA an assembly language?

The truth is, you can actually write a very similar looking program with MASM. Here's an example I trot out for unbelievers. This code is compilable with MASM (assuming you include the UCR Standard Library v2.0 and some additional code I've cut out for brevity:

```
var

    enum colors,<red,green,blue>

    colors c1, c2

endvar

Main      proc

    mov    ax, dseg

    mov    ds, ax

    mov    es, ax

    MemInit

    InitExcept

    EnableExcept

    finit

    try

        cout    "Enter two colors:"

        cin     c1, c2

        cout    "You entered ",c1," and ",c2,nl

        .if     c1 == red
```

```
        cout "c1 was red"

        .endif

    except $Conversion

        cout    "Conversion error occured",nl

    except $Overflow

        cout    "Overflow error occured",nl

    endtry

    CleanUpEx

    ExitPgm                ;DOS macro to quit program.

Main    endp
```

As you can see, the only identifiable machine instructions here are the ones that initialize the segment registers at the beginning of the program (which is unnecessary in a Win32 environment). So allow me to blunt criticism from "die-hard" assembly fans right at the start: HLA doesn't open up all kinds of new programming paradigms that weren't possible before. With some clever macros (e.g., `enum`, `cout`, and `cin` in the MASM code), it is quite possible to do some amazing things. If you're wondering why you should bother with HLA if MASM is so wonderful, don't forget my comments about the robustness of these macros. Both HLA and MASM (with the UCR Standard Library v2.0) work great as long as you write perfect code and don't make any mistakes. However, if you do make mistakes, the MASM macro scheme rapidly gets ugly.

The "die-hard" assembly fan will probably observe that they would never write code like the MASM code I've presented above; they would write traditional assembly code. They want to write traditional code. They don't want this high level syntax forced upon them. Well, HLA doesn't force you to use high-level control structures rather than machine instructions. You can always write the low level code if you prefer it that way. Here is the original HLA program rewritten to use familiar machine instructions:

```
program SampleHLApgm2;

#include( "stdlib.hhf" )

static
```



```
        dword 37, 37;

TcHWpStr: dword; @nostorage;

        byte  "The classical 'Hello World' program: ",0,0,0;

        dword 11, 11;

HWstr:   dword; @nostorage;

        byte  "Hello World",0;

begin SampleHLApgm2;

    lea( eax, TcHWpStr );

    push( eax );

    call stdout.puts;

    lea( eax, HWstr );

    push( eax );

    call stdout.puts;

    call stdout.newln;

end SampleHLApgm2;
```

The `stdout.puts` and `stdout.newln` procedures come from the HLA Standard Library. I will leave it up to the interested reader to translate these into Win API Write calls if this code isn't sufficiently low level to satisfy. Note that HLA strings are not simple zero terminated strings like C/C++. This explains the extra zeros and dword values in the `STATIC` section (the dword values hold the string lengths; I offer these without further explanation, see the HLA documentation for more details on HLA's string format).

One thing you've probably noticed from this second example is that HLA uses a functional notation for assembly language statements. That is, the instruction mnemonics look like function calls in a high level language and the operands look like parameters to those functions. The neat thing about this notation is that it easily allows the use of "instruction composition." Instruction composition, like functional composition, means that you get to use one instruction as the operand of another. For example, an instruction like `mov(mov(0, eax), ebx);` is perfectly legal in HLA.

The HLA compiler will compile the innermost instruction first and then substitute the destination operand of the innermost instruction for the operand position occupied by the instruction. HLA's MOV instruction takes the generic form "MOV(source, destination);" so the former instruction translates to the following two instruction sequence:

```
mov( 0, eax );      // intel syntax:  mov eax, 0
mov( eax, ebx );   // intel syntax:  mov ebx, eax
```

By and of itself, instruction composition is somewhat interesting, but programmers striving to write readable code need to exercise caution when using instruction composition. It is very easy to write some unreadable code if you abuse instruction composition. E.g., consider:

```
mov( add( mov( 0, eax ), sub( ebx, ecx) ), edx ), mov( i, esi ) );
```

Egads! What does this mess do? Some might consider the inclusion of instruction composition in HLA to be a fault of the language if it allows you to write such unreadable code. However, I've never felt it was the language syntax's job to enforce good programming style. If there's really a reason for writing such messy code, the compiler shouldn't prevent it.

Although you can produce some truly unreadable messes with instruction composition, if you use it properly it can enhance the readability of your programs. For example, HLA lets you associate an arbitrary string with a procedure that HLA will substitute for that procedure name when the procedure call appears as an operand of another instruction. Most functions that return a value in a register specify that register name as their "returns" string (the string HLA substitutes for the procedure call). For example, the "str.eq(str1, str2)" function compares the two string operands and returns true or false in AL depending on the result of the comparison. This allows you to write code like the following:

```
if( str.eq( str1, "Hello" ) ) then

    stdout.put( "str1 = 'Hello'" nl );

endif;
```

HLA directly translates the IF statement into the following sequence:

```
str.eq( str1, "Hello" );
if( @c ) then

    stdout.put( "str1= 'Hello'" nl );

endif;
```

Arguably, the former version is a little more readable than the latter version. Instruction composition, when you use it in this fashion, lets you write code that "looks" a little more high level

without the compiler having to generate lots of extra code (as it would if HLA supported a generalized arithmetic expression parser).

Like MASM, HLA supports a wide variety of high level control structures. HLA's set is both higher level and lower level at the same time. There is a good reason HLA's control structures aren't always as powerful as MASM's. First, with the sole exception of object method invocations, I made a rule that HLA's high level control structures would not modify any general purpose registers behind the programmer's back. MASM, for example, may modify the value in EAX for certain boolean expressions or parameter values it must compute.

Although I designed HLA as a tool to teach assembly language programming, this is also a tool that I intend to use so I included many goodies for advanced assembly language programmers. For example, HLA's macro facilities are more powerful than I've seen in any programming-language-based macro processor. One unique feature of HLA's macro preprocessor is the ability to create "context free" control structures using macros. For example, suppose that you decide that you need a new type of looping construct that HLA doesn't provide; let's say, a loop that will repeat once for each character in a string supplied as a parameter to the loop. Let's call this loop "OnceForEachChar" and decide on the following syntax:

```
OnceForEachChar( SomeString )

    << Loop Body >>

endOnceForEachChar;
```

On each iteration of this loop, the AL register will contain the corresponding character from the string specified as the OnceForEachChar operand. You can easily implement this loop using the following HLA macro:

```
#macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );          // index into string.

    TopOfLoop:

        inc( (type dword [esp] ) );    // Bump up index into string.
        #if( @IsConst( SomeString ) )

            // Load address of string constant into EAX.

            lea( eax, SomeString );

        #else

            mov( SomeString, eax );    // Get ptr to string.

        #endif
        add( [esp], eax );    // Point at next available character
        mov( [eax], al );    // Get the next available character
        cmp( al, 0 );        // See if we're at the end of the string
        je LoopExit;

#terminator endOnceForEachChar;

        jmp TopOfLoop;        // Return to the top of the loop and repeat.

    LoopExit:
```

```
        add( 4, esp );        // Remove index into string from stack.

#endmacro
```

Anyone familiar with MASM's macro processor should be able to figure out most of this code. Note that the symbols "TopOfLoop" and "LoopExit" are local symbols to this macro. Hence, if you repeat this macro several times in the code, HLA will emit different actual labels for these symbols to the MASM output file. The "@IsConst" is an HLA compile-time function that returns true if its operand is a constant. Obtaining the address for a constant is fundamentally different than obtaining the address of a string variable (since HLA string variables are actually pointers to the string data). The most interesting feature of this macro definition is the "terminator" line. This actually defines a second macro that is active only after HLA encounters the "OnceForEachChar" macro and control returns to the first statement after the OnceForEachChar invocation. Invocations of "context free" macros always occur in pairs; that is, for every "OnceForEachChar" invocation there must be a matching "endOnceForEachChar" invocation. The following program demonstrates this macro in use; it also demonstrates that you can nest this newly created control structure in your program:

```
program SampleHLApgm3;
#include( "stdlib.hhf" )

#macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );        // index into string.
    TopOfLoop:
        inc( (type dword [esp] ) );
        #if( @IsConst( SomeString ) )

            lea( eax, SomeString );

        #else

            mov( SomeString, eax );

        #endif
        add( [esp], eax );
        mov( [eax], al );
        cmp( al, 0 );
        je LoopExit;

#terminator endOnceForEachChar;

        jmp TopOfLoop;

    LoopExit:

        add( 4, esp );

#endmacro
```

```

static
    strVar: string := ":" nl;

begin SampleHLApgm3;

    OnceForEachChar( "Hello" )

        stdout.putc( al );
        OnceForEachChar( strVar )

            stdout.putc( al );

        endOnceForEachChar;

    endOnceForEachChar;

end SampleHLApgm3;

```

This program produces the output:

```

H:
e:
l:
l:
o:

```

Here's some sample MASM code, similar to what the HLA compiler emits (when using the -masm and -source command-line options) for the sequence above:

```

strings          segment page public 'data'
                 align      4
?635_len         dword      5
                 dword      5
?635_str         byte       "Hello",0,0,0

strings          ends

                 pushd      -1

?634__0278_:
                 inc        dword ptr [esp+0]          ;(type dword [esp])
                 lea        eax, ?635_str
                 add        eax, [esp+0] ;[esp]
                 mov        al, [eax+0] ;[eax]
                 cmp        al, 0
                 je         ?636__0279_
                 push       eax
                 call       stdio_putc          ;putc

```

```

        pushd    -1

?639__027d_:
        inc     dword ptr [esp+0]      ;(type dword [esp])
        mov     eax, dword ptr ?630_strVar[0] ;strVar
        add     eax, [esp+0] ;[esp]
        mov     al, [eax+0] ;[eax]
        cmp     al, 0
        je     ?640__027e_
        push   eax
        call   stdio_putc      ;putc
        jmp    ?639__027d_

?640__027e_:
        add     esp, 4
        jmp    ?634__0278_

?636__0279_:
        add     esp, 4

```

In addition to the "terminator" clause, HLA macros also support a "keyword" clause that let you bury reserved words within a context-free language construct. For example, although the HLA language provides a SWITCH/CASE statement, you can create a new one with slightly different semantics. I implemented the SWITCH .. CASE .. DEFAULT .. ENDCASE statement using HLA's macro facilities (as a demonstration of HLA's power). An HLA SWITCH statement (using this macro) takes the following form:

```

switch( reg32 )

    case( constantList1 )

        << statements >>

    case (constantList2 )

        << statements >>

        .
        .
        .

    default // This is optional

        << statements >>

endswitch;

```

The switch macro implements the "switch" and "endswitch" reserved words using the macro and terminator clauses in the macro declaration. It implements the "case" and "default" reserved words using the HLA "keyword" clause in a macro definition. The "keyword" clause is similar to the "terminator" clause except it doesn't force the end of the macro expansion in the invoking code. The actual code for the HLA SWITCH statement is a little too complex to present here, so I will

extend the example of the `OnceForEachChar` macro to demonstrate how you code use the "keyword" clause in a macro.

Let's suppose you wanted to add a `"_break"` clause to the `"OnceForEachChar"` loop (I'm using `"_break"` with an underscore because `"break"` is an HLA reserved word). You could easily modify the `"OnceForEachChar"` macro to achieve this by using the following code:

```
#macro OnceForEachChar( SomeString ): TopOfLoop, LoopExit;

    pushd( -1 );          // index into string.
    TopOfLoop:

        inc( (type dword [esp] ) );
        #if( @IsConst( SomeString )

            lea( eax, SomeString );

        #else

            mov( SomeString, eax );

        #endif
        add( [esp], eax );
        mov( [eax], al );
        cmp( al, 0 );
        je LoopExit;

    #keyword _break;
        jmp LoopExit;

#terminator endOnceForEachChar;

    jmp TopOfLoop;

LoopExit:

    add( 4, esp );

#endmacro
```

The `"#keyword"` clause defines a macro (`"_break"`) that is active between the `"OnceForEachChar"` and `"endOnceForEachChar"` invocations. This macro simply expands to a `jmp` instruction that exits the loop. Note that if you have nested `"OnceForEachChar"` loops and you `"_break"` out of the innermost loop, the code only jumps out of the innermost loop, exactly as you would expect.

HLA's macro facilities are part of a larger feature I refer to as the "HLA Compile-Time Language." HLA actually contains a built-in interpreter than executes while it is compiling your program. The compile-time language provides conditional compilation (the `#IF.#ELSE.#ENDIF` statements in the previous example), interpreted procedure calls (macros), looping constructs (`#WHILE.#ENDWHILE`), a very powerful constant expression evaluator, compile-time I/O facilities (`#PRINT, #ERROR, #INCLUDE, and #TEXT.#ENDTEXT`), and dozens of built-in compile time functions (like the `@IsConst` function above).

The HLA built-in string functions (not to be confused with the HLA Standard Library's string functions) are actually powerful enough to let you write a compiler for a high level language completely within HLA. I mentioned earlier that it is possible to write an expression compiler within HLA; I was serious. The HLA compile-time language will let you write a sophisticated recursive descent parser for arithmetic expressions (and other context-free language constructs, for that matter).

HLA is a great tool for creating low-level Domain Specific Embedded Languages (DSELS). DSELS are mini-languages that you create on a project-by-project basis to help reduce development time. HLA's compile time language lets you create some very high level constructs. For example, HLA implements a very powerful string pattern matching language in the "patterns" module found in the HLA Standard Library. This module lets you write pattern-matching programs that use techniques found in language like SNOBOL4 and Icon. As a final example, consider the following HLA program that translate RPN (reverse polish notation) expressions into their equivalent assembly language (HLA) statements and displays the results to the standard output:

```
// This program translates user RPN input into an
// equivalent sequence of assembly language instrs (HLA fmt).

program RPNtoASM;
#include( "stdlib.hhf" );

static
    s:          string;
    operand:    string;
    StartOperand: dword;

#macro mark;

    mov( esi, StartOperand );

#endmacro

#macro delete;

    mov( StartOperand, eax );
    sub( eax, esi );
    inc( esi );
    sub( s, eax );
    str.delete( s, eax, esi );

#endmacro

procedure length( s:string ); @returns( "eax" ); @nodisplay;
begin length;

    push( ebx );
    mov( s, ebx );
    mov( (type str.strRec [ebx]).length, eax );
    pop( ebx );

end length;

begin RPNtoASM;

    stdout.put( "-- RPN to assembly --" nl );
    forever
```



```

stdout.put( nl nl "Enter RPN sequence (empty line to quit): " );
stdin.a_gets();
mov( eax, s );
breakif( length( s ) = 0 );
while( length( s ) <> 0 ) do

    pat.match( s );

    // Match identifiers and numeric constants

    mark;
    pat.zeroOrMoreWS();
    pat.oneOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_' } );
    pat.a_extract( operand );
    stdout.put( "    pushd( ", operand, " );" nl );
    strfree( operand );
    delete;

pat.alternate;

    // Handle the "+" operator.

    mark;
    pat.zeroOrMoreWS();
    pat.oneChar( '+' );
    stdout.put
    (
        "    pop( eax );" nl
        "    add( eax, [esp] );" nl
    );
    delete;

pat.alternate;

    // Handle the '-' operator.

    mark;
    pat.zeroOrMoreWS();
    pat.oneChar( '-' );
    stdout.put
    (
        "    pop( eax );" nl
        "    pop( ebx );" nl
        "    sub( eax, ebx );" nl
        "    push( ebx );" nl
    );
    delete;

pat.alternate;

    // Handle the '*' operator.

    mark;
    pat.zeroOrMoreWS();
    pat.oneChar( '*' );
    stdout.put
    (

```

```

        "    pop( eax );" nl
        "    imul( eax, [esp] );" nl
    );
    delete;

pat.alternate;

    // handle the '/' operator.

    mark;
    pat.zeroOrMoreWS();
    pat.oneChar( '/' );
    stdout.put
    (
        "    pop( ebx );" nl
        "    pop( eax );" nl
        "    cdq();" nl
        "    idiv( ebx, edx:eax );" nl
        "    push( ebx );" nl
    );
    delete;

pat.if_failure

    // If none of the above, it must be an error.

    stdout.put( nl "Illegal RPN Expression" nl );
    mov( s, ebx );
    mov( 0, (type str.strRec [ebx]).length );

pat.endmatch;

    endwhile;

endfor;

end RPNtoASM;

```

Consider for a moment the code that matches an identifier or an integer constant:

```

    mark;
    pat.zeroOrMoreWS();
    pat.oneOrMoreCset( { 'a'..'z', 'A'..'Z', '0'..'9', '_' } );
    pat.a_extract( operand );
    stdout.put( "    pushd( ", operand, " );" nl );
    strfree( operand );
    delete;

```

The "mark;" invocation saves a pointer into the "s" string where the current identifier starts. The pat.ZeroOrMoreWS pattern matching function skips over zero or more whitespace characters.

The `pat.OneOrMoreCset` pattern match function matches one or more alphanumeric and underscore characters (a crude approximation for identifiers and integer constants). The `pat.a_extract` function makes a copy of the string between the "mark" and the "a_extract" calls (this corresponds to the whitespace and identifier/constant). The `stdout.put` statement emits the HLA machine instruction that will push this operand on to the x86 stack for later computations. The remaining statements clean up allocated string storage space and delete the matched string from "s".

Although the "pat.xxxxx" statements look like simple function calls, there's actually a whole lot more going on here. HLA's pattern matching facilities, like SNOBOL4 and Icon, support success, failure, and backtracking. For example, if the `pat.oneOrMoreChar` function fails to match at least one character from the set, control does not flow down to the `pat.a_extract` function. Instead, control flows to the next "pat.alternate" or "pat.if_failure" clause. Some calls to HLA pattern matching routines may even cause the program to back up in the code and reexecute previously called functions in an attempt to match a difficult pattern (i.e., the backtracking component). This article is not the place to get into the theory of pattern matching; however, these few examples should be sufficient to show you that something really special is going on here. And all these facilities were developed using the HLA compile-time language. This should give you a small indication of what is possible when using the HLA compile-time language facilities.

2 The Quick Guide to HLA

2.1 Overview

This guide is designed to help those who are already familiar with x86 assembly language programming to get up to speed with HLA as rapidly as possible. HLA was designed as a tool for teaching assembly language programming to University/College students who have no prior experience with assembly language but have some high level language programming experience (C/C++, Pascal, Java, etc.). The documentation that exists for HLA comes in two forms: the HLA reference manuals and the "Art of Assembly Language Programming/32-bit Edition." The "Art of Assembly" text is suitable for students and beginners to assembly language programming; it starts from square one and teaches assembly language programming using HLA. Unfortunately, this text is not particularly suitable for those programmers who already know assembly language. The HLA reference manuals are great when you need to look up some particular feature. They do fully explain the HLA language, however, the HLA language is rather large so the assembly programmer who is new to HLA is faced with reading a tremendous amount of material just to get started with HLA. Most individuals won't bother. The purpose of this guide is to present a very small subset of HLA to the advanced x86 assembly language programmer in as few pages as possible. This guide does not attempt to teach any of HLA's special features; it assumes the reader is using an assembler such as MASM, TASM, NASM, Gas, etc., and is interested in learning how to write assembly code using HLA in a fashion comparable to those assemblers. Of course, the whole reason for such a person to learn HLA is to be able to take advantage of HLA's advanced features. However, one has to learn to walk before they run, this is the guide that will get that person walking. Once the reader is comfortable using HLA in a "traditional assembly" sense, then that reader can refer to the HLA reference manuals in order to learn the more advanced features of the language.

2.2 Running HLA

HLA is a command line tool that you run from the Win32, Mac OSX, Linux, or FreeBSD Command Prompt. This document assumes that you are familiar with basic command prompt syntax and you're familiar with various commands like "DIR" and "RENAME" (under Windows) or "ls" and "mv" (under *NIX). To run HLA from the command line prompt, you use a command like the following:

```
hla optional_command_line_parameters Filename_list
```

The filename list consists of one or more unambiguous filenames having the extension: HLA, ASM, or OBJ. HLA will first run the HLPARSE program on all files with the HLA extension (producing files with the same base name and an .obj/.o extension). Finally, HLA runs the linker to combine all the object files together. The ultimate result, assuming there were no errors along the way, is an executable file .

HLA supports the following command line parameters:

```
options:
-@           Do not generate linker response file.
-@@          Always generate a linker response file.
-thread      Enable thread-safe code generation and linkage.
-axxxxx     Pass xxxxx as command line parameter to assembler.
-dxx         Define VAL symbol xx to have type BOOLEAN and value TRUE.
-dxx=yy      Define VAL symbol xx to have type STRING and value "yy".
-e:name      Executable output filename (appends ".exe" under Windows).
-x:name      Executable output filename (does not append ".exe").
-b:name      Binary object file output name (only when using HLABE).
-i:path      Specifies path to HLA include file directory.
-lib:path    Specifies path to the HLALIB.LIB file.
```

-license	Displays copyright and license info for the HLA system.
-lxxxxx	Pass xxxxx as command line parameter to linker.
-m	Create a map file during link
-p:path	Specifies path to hold temporary working files.
-r:name	<name> is a text file containing cmd line options.
-obj:path	Specifies path to place object files.
-main:name	Use 'name' as the name of the HLA main program.
-source	Compile to human readable source file format.
-s	Compile to .ASM files only.
-c	Compile and assemble to object files only.
-fasm	Use FASM as back-end assembler (applies to -s and -c)
-gas	Use GAS as back-end (Linux/BSD, applies to -s and -c)
-gasx	Use Gas as back-end (Mac OSX, --s and -c only)
-hla	Produce a pseudo-HLA source file as output (implies -s).
-hlabe	(Default) Produce object code using the HLA Back Engine.
-masm	Use MASM as back-end assembler (applies to -s and -c)
-nasm	Use NASM as back-end assembler (applies to -s and -c)
-tasm	Use TASM as back-end assembler (applies to -s and -c)
-sym	Dump symbol table after compile.
-win32	Generate code for Win32 OS.
-linux	Generate code for Linux OS.
-freebsd	Generate code for FreeBSD OS.
-macos	Generate code for Mac OSX.
-test	Send diagnostic info to stdout rather than stderr (This option is intended for HLA test/debug purposes).
-v	Verbose compile.
-level=h	High-level assembly language
-level=m	Medium-level assembly language
-level=l	Low-level assembly language
-level=v	Machine-level assembly language (very low level).
-w	Compile as windows app (default is console app).
-?	Display this help message.

Please see the appropriate chapter in the HLA Reference Manual chapter *Using the HLA Command-Line Compiler* for an explanation of each of these options. Most of the time, you will not use any of these options when compiling typical HLA programs. The "-c" and "-s" options are the ones you will use most commonly (and this document assumes that you understand their purpose).

2.3 HLA Language Elements

Starting with this section we begin discussing the HLA source language. HLA source files must contain only seven-bit ASCII characters. These are Windows text files with each source line record containing a carriage return/line feed termination sequence or *NIX (Mac OSX, Linux, and FreeBSD) source files with a line feed terminating each line. White space consists of spaces, tabs, and newline sequences. Generally, HLA does not appreciate other control characters in the file and may generate an error if they appear in the source file.

2.3.1 Comments

HLA uses "/" to lead off single line comments. It uses "/*" to begin an indefinite length comments and it uses "*/" to end an indefinite length comment. C/C++, Java, and Delphi users will be quite comfortable with this notation.

2.3.2 Special Symbols

The following characters are HLA lexical elements and have special meaning to HLA:

```
* / + - ( ) [ ] { } < > : ; , . = ? & | ^ ! @
&& || <= >= <> != == := .. << >>
## #( )# #{ }#
```

This document will not explain the meaning of all these symbols, only the minimum necessary to write simple HLA programs. See the HLA Reference Manual for more details.

2.3.3 Reserved Words

HLA supports a large number of reserved words (mostly, they are machine instructions). For brevity, that list does not appear here; please see the HLA reference manual chapter *HLA Language Elements* for a complete and up-to-date list. Note that HLA does not allow you to use a reserved word as a program identifier, so you should scan over the list at least once to familiarize yourself with reserved words that you might be used to using as identifiers in your assembly language programs. HLA reserved words are case insensitive. That is, "MOV" and "mov" (as well as any permutation with respect to case) both represent the HLA "mov" reserved word.

2.3.4 External Symbols and Assembler Reserved Words

HLA v2.0 produces an option to produce an assembly language file during compilation and can invoke an assembler such as MASM, FASM, NASM, or Gas to complete the compilation process. HLA automatically translates normal identifiers you declare in your program to benign identifiers in the assembly language program. However, HLA does not translate EXTERNAL symbols, but preserves these names in the assembly language file it produces. Therefore, you must take care not to use external names that conflict with the underlying assembler's set of reserved words or that assembler will generate an error when it attempts to process HLA's output.

For a list of the back-end assembler's reserved words, please see the documentation for the assembler you are using to process HLA's output (i.e., MASM, NASM, FASM, or Gas).

2.3.5 HLA Identifiers

HLA identifiers must begin with an alphabetic character or an underscore. After the first character, the identifier may contain alphanumeric and underscore symbols. There is no technical limit on identifier length in HLA, but you should avoid external symbols greater than about 32 characters in length since the assemblers and linkers that process HLA output may not be able to handle such symbols.

HLA identifiers are always *case neutral*. This means that identifiers are case sensitive insofar as you must always spell an identifier exactly the same (with respect to alphabetic case). However, you are not allowed to declare two identifiers whose only difference is alphabetic case.

2.3.6 External Identifiers

HLA lets you explicitly provide a string for external identifiers. External identifiers are not limited to the format for HLA identifiers. HLA allows any string constant to be used for an external identifier. It is your responsibility to use only those characters that are legal in the back-end assembler (if you are using one). Note that this feature lets you use symbols that are not legal in HLA but are legal in external code (e.g., Win32 APIs use the '@' character in identifiers). See the discussion of the **external** option for more details.

2.4 Data Types in HLA

2.4.1 Native (Primitive) Data Types in HLA

HLA provides the following basic primitive types:

```
One-byte types: byte, boolean, enum, uns8, int8, and char.
Two-byte types: word, uns16, int16.
Four-byte types: dword, uns32, int32, real32, string, pointer
Eight-byte types: uns64, int64, qword, thunk, and real64.
Ten-Byte types: tbyte, and real80.
Sixteen-byte types: uns128, int128, lword, and cset
```

For details on these particular types, please consult the HLA Reference Manual chapter *HLA Data Types*. This document will make use of the following types:

`byte`, `word`, `dword`, `string`, `real32`, `qword`, `real64`, and `real80`

These are the typical types assembly language programmers use.

BYTE variables and objects may hold integer numeric values in the range -128..+255, any ASCII character constant, and the two predefined boolean values *true* (1) and *false* (0). Normally, HLA does a small amount of type checking; however, you can associate any value that can fit into eight bits with a byte-sized variable (or other object).

WORD variables and object may hold integer numeric values in the range -32768..+65535. Generally, HLA does not allow the association of other values with a WORD object.

DWORD variables and objects may hold integer numeric values in the range -2147483647..+4294967295, or the address of an object (using the "&" address-of operator).

STRING variables are also DWORD objects. STRING objects hold the address of a sequence of zero or more ASCII characters that end with a zero byte. In the four bytes immediately preceding the location contained in the string pointer is the current length of the string. In the four bytes preceding the current length is the maximum allowable length of the string. Note that HLA strings are "read-only" compatible with ASCIIZ strings used by Windows and C/C++ (read-only meaning that you can pass an HLA string to a Windows API or C/C++ function but that function should not modify the string).

QWORD, UNS64, and INT64 objects consume eight bytes of memory. TBYTE objects consume ten bytes (80 bits). LWORD, UNS128, and INT128 values are also legal and support 128-bit hexadecimal, unsigned, or signed constants.

REAL32, REAL64, and REAL80 types in HLA support the three different IEEE floating-point formats.

2.4.2 Composite Data Types

In addition to the primitive types above, HLA supports arrays, records (structures), unions, classes, and pointers of the above types (except for text objects).

2.4.3 Array Data Types

HLA allows you to create an array data type by specifying the number of array elements after a type name. Consider the following HLA type declaration that defines `intArray` to be an array of `dword` objects:

```
type intArray : dword[ 16 ];
```

The "[16]" component tells HLA that this type has 16 four-byte double words. HLA arrays use a zero-based index, so the first element is always element zero. The index of the last element, in this example, is 15 (total of 16 elements with indices 0..15).

HLA also supports multidimensional arrays. You can specify multidimensional arrays by providing a list of indices inside the square brackets, e.g.,

```
type intArray4x4 : dword[ 4, 4 ];
type intArray2x2x4 : dword[ 2,2,4 ];
```

2.4.4 Record Data Types¹

HLA's records allow programmers to create data types whose fields can be different types. The following HLA static variable declaration defines a simple record with four fields:

```
static Planet:
```

1. For C/C++ programmers: an HLA record is similar to a C struct. In language design terminology, a record is often referred to as a "cartesian product."

```
record
    x:          dword;
    y:          dword;
    z:          dword;
    density: real64;
endrecord;
```

Objects of type Planet will consume 20 bytes of storage at run-time.

The fields of a record may be of any legal HLA data type including other composite data types. You use dot-notation to access fields of a record object, e.g.,

```
mov( Planet.x, eax );
```

2.5 Literal Constants

Literal constants are those language elements that we normally think of as non-symbolic constant objects. HLA supports a wide variety of literal constants. The following sections describe those constants.

2.5.1 Numeric Constants

HLA lets you specify several different types of numeric constants.

2.5.1.1 Decimal Constants

The first and last characters of a decimal integer constant must be decimal digits (0..9). Interior positions may contain decimal digits and underscores. The purpose of the underscore is to provide a better presentation for large decimal values (i.e., use the underscore in place of a comma in large values). Example: 1_234_265.

2.5.1.2 Hexadecimal Constants

Hexadecimal literal constants must begin with a dollar sign ("\$\$") followed by a hexadecimal digit and must end with a hexadecimal digit (0..9, A..F, or a..f). Interior positions may contain hexadecimal digits or underscores. Hexadecimal constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., \$1A_2F34_5438.

2.5.1.3 Binary Constants

Binary literal constants begin with a percent sign ("%") followed by at least one binary digit (0/1) and they must end with a binary digit. Interior positions may contain binary digits or underscore characters. Binary constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., %10_1111_1010.

2.5.1.4 Real (Floating Point) Constants

Floating point (real) literal constants always begin with a decimal digit (never just a decimal point). A string of one or more decimal digits may be optionally followed by a decimal point and zero or more decimal digits (the fractional part). After the optional fractional part, a floating point number may be followed by "e" or "E", a sign ("+" or "-"), and a string of one or more decimal digits (the exponent part). Underscores may appear between two adjacent digits in the floating point number; their presence is intended to substitute for commas found in real-world decimal numbers.

2.5.1.5 Boolean Constants

Boolean constants consist of the two predefined identifiers *true* and *false*. Note that your program may redefine these identifiers, but doing so is incredibly bad programming style.

2.5.1.6 Character Constants

Character literals generally consist of a single (graphic) character surrounded by apostrophes. To represent the apostrophe character, you use four apostrophes, e.g., ''''.

Another way to specify a character constant is by typing the "#" symbol followed by a numeric literal constant (decimal, hexadecimal, or binary). Examples: #13, #SD, #%1101.

2.5.1.7 String Constants

String literal constants consist of a sequence of (graphic) characters surrounded by quotes. To embed a quote within a string, insert a pair of quotes into the string, e.g., "He said ""This"" to me."

If two string literal constants are adjacent in a source file (with nothing but whitespace between them), then HLA will concatenate the two strings and present them to the parser as a single string. Furthermore, if a character constant is adjacent to a string, HLA will concatenate the character and string to form a single string object. This is useful, for example, when you need to embed control characters into a string, e.g.,

```
"This is the first line" #d #a "This is the second line" #d #a
```

HLA treats the above as a single string with a newline sequence (CR/LF) at the end of each of the two lines of text.

2.5.1.8 Pointer Constants

HLA allows a very limited form of a pointer constant. If you place an ampersand in front of a static object's name (i.e., the name of a **static** variable, **readonly** variable, uninitialized (**storage**) variable, procedure, method, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize static or **readonly** variables or as constant expressions in 80x86 instructions.

2.5.1.9 Structured Constants

HLA also supports certain structured constants including character set constants, array constants, union constants and record constants. Please see the HLA Reference Manual chapter *HLA Constants* for more details.

2.6 Constant Expressions in HLA

HLA provides a rich expression evaluator to process assembly-time expressions. HLA supports the following operators (sorting by decreasing precedence):

```
! (unary not), - (unary negation)
*, div, mod, /, <<, >>
+, -
=, ==, <>, !=, <=, >=, <, >
&, |, &, in
```

```
!expr
```

The expression must be either boolean or a number. For boolean values, *not* ("!") computes the standard logical not operation. For numbers, *not* ("!") computes the bitwise not operation on the bits of the number.

```
- expr          (unary negation operator)
expr1 * expr2   (multiplication operator)
expr1 div expr2 (integer division operator)
expr1 mod expr2 (integer remainder operator)
expr1 / expr2   (real division operator)
expr1 << expr2  (integer shift left operator)
expr1 >> expr2  (integer shift right operator)
expr1 + expr2   (addition operator)
expr1 - expr2   (subtraction operator)
expr1 = expr2   (equality comparison operator)
expr1 <> expr2  (inequality comparison operator)
```

```

expr1 < expr2      (less than comparison operator)
expr1 <= expr2     (less than or equal comparison operator)
expr1 > expr2      (greater than comparison operator)
expr1 >= expr2     (greater or equal comparison operator)
expr1 & expr2      (logical/boolean AND operator)
expr1 | expr2      (logical/boolean OR operator)
expr1 ^ expr2      (logical/boolean XOR operator)
( expr )            (override operator precedence)

```

HLA supports several other constant operators. Furthermore, many of the above operators are overloaded depending on the operand types. Note that for numeric (integer) operands, HLA fully support 128-bit arithmetic. Please see the HLA Reference Manual chapter *HLA Constants* for more details.

2.7 Program Structure

An HLA program uses the following general syntax:

```

program identifier ;
    declarations
begin identifier;
    statements
end identifier;

```

The three identifiers above must all match. The declaration section (declarations) consists of **type**, **const**, **val**, **var**, **static**, **storage**, **readonly**, **procedure**, **iterator**, and **method** definitions. Any number of these sections may appear and they may appear in any order; more than one of each section may appear in the declaration section.

If you wish to write a library module that contains only procedures and no main program, you would use an HLA unit. Units have a syntax that is nearly identical to programs, there isn't a **begin** associated with the unit, e.g.,

```

unit TestPgm;

    procedure LibraryRoutine;
    begin LibraryRoutine;
        << etc. >>
    end LibraryRoutine;

end TestPgm;

```

2.8 Procedure Declarations

Procedure declarations are nearly identical to program declarations.

```

procedure identifier; @noframe;
begin identifier;
    statements
end identifier;

```

Note that HLA procedures provide a very rich set of syntactical options. The template above corresponds to the syntax that creates procedures most closely resembling those that other assemblers use. HLA's procedures allow parameters, local variable declarations, and many other features this document won't describe. For more details, please see the HLA Reference Manual chapter on *HLA Procedures*.

Note, *and this is very important*, that the procedure option **@noframe** must appear in the **procedure** declaration. Without this declaration, HLA inserts some additional code into your procedure and it will probably fail to work as you intend (indeed, it's likely the inserted code will crash when it runs).

Example of a procedure:

```
procedure ProcDemo; @noframe;
begin ProcDemo;

    add( 5, eax );
    ret();

end ProcDemo;
```

2.8.1 Declarations

Programs, units, procedures, methods, and iterators all have a declaration section. Classes and namespaces also have a declaration section, though it is somewhat limited. A declaration section can contain one or more of the following components (among other things this document doesn't cover):

- A type section.
- A const section.
- A static section.
- A procedure.

The order of these sections is irrelevant as long as you ensure that all identifiers used in a program are defined before their first use. Furthermore, as noted above, you may have multiple sections within the same set of declarations. For example, the two const sections in the following procedure declaration are legal:

```
program TwoConsts;
const  MaxVal := 5;
type   Limits: dword[ MaxVal ];
const  MinVal := 0;
begin TwoConsts;

    //...

end TwoConsts;
```

2.8.2 Type Section

You can declare user-defined data types in the type section. The type section appears in a declaration section and begins with the reserved word **type**. It continues until encountering another declaration reserved word (e.g., const, var, or val) or the reserved word **begin**. A typical type definition begins with an identifier followed by a colon and a type definition. The following paragraphs demonstrate some of the legal forms of type definitions. See the HLA Reference Manual chapter on *HLA Program Structure* for more examples.

```
id1 : id2;           // Defines id1 to be the same as type id2.
id1 : id2 [ dim_list ]; // Defines id1 to be an array of type id2.
id1 : record        // Defines id1 as a record type.
    field_declarations
endrecord;
```

2.8.3 Const Section

You may declare manifest constants in the **const** section of an HLA program. It is illegal to attempt to change the value of a constant at some later point during assembly. Of course, at run-time the constant always has a fixed value.

The constant declaration section begins with the reserved word **const** and is followed by a sequence of constant definitions. The constant declaration section ends when HLA encounters a keyword such as **const**, **type**, **var**, **val**, etc. Actual constant definitions take the forms specified in the following paragraphs.

```
id := expr; // Assigns the value and type of expr to id
id1 : id2 := expr; // Creates constant id1 of type id2 of value expr.
```

Note that HLA supports several types of constants this section doesn't discuss (e.g., array and record constants and well as compile-time variables). See the HLA Reference Manual chapter on *HLA Program Structure* for more details.

2.8.4 Static Section

The **static** section lets you declare static variables you can reference at run-time by your code. The following paragraphs list some of the forms that are legal in the **static** section. As usual, see the HLA Reference Manual chapter on *HLA Program Structure* for lots of additional features that HLA supports in the **static** section.

```
static
  id1 : id2; // Declares variable id1 of type id2
  id1 : id2 := expr; // Declares variable id1 of type id2, init'd with
  expr
  id1 : id2[ expr ]; // Declares array id1 of type id2 with expr
  elements
```

2.8.4.1 The @NOSTORAGE Option

The **@nostorage** option tells HLA to associate the current offset in the segment with the specified variable, but don't actually allocate any storage for the object. This option effectively creates an alias of the current variable with the next object you declare in one of the static sections. Consider the following example:

```
static
  b: byte; @nostorage;
  w: word; @nostorage;
  d: dword;
```

Because the *b* and *w* variables both have the **@nostorage** option associated with them, HLA does not reserve any storage for these variables. The *d* variable does not have the **@nostorage** option, so HLA does reserve four bytes for this variable. The *b* and *w* variables, since they don't have storage associated with them, share the same address in memory with the *d* variable.

2.8.4.2 The EXTERNAL Option

The **external** option gives you the ability to reference variables that you declare in other files. Like the external clause for procedures, there are two different syntaxes for the external clause appearing after a variable declaration:

```
varName: varType; external;
varName: varType; external( "external_Name" );
```

The first form above uses the variable's name for both the internal and external names. The second form uses *varName* as the internal name that HLA uses and it associates this variable with *external_Name* in the external modules. The **external** option is always the last option associated with a variable declaration.

If the actual variable definition for an external object appears in a source file after an external declaration, this tells HLA that the definition is a public variable that other modules may access

(the default is local to the current source file). This is the only way to declare a variable public so that other modules can use it. Usually, you would put the external declaration in a header file that all modules (wanting to access the variable) include; you also include this header file in the source file containing the actual variable declaration.

2.8.5 Macros

HLA has one of the most powerful macro expansion facilities of any programming language. HLA's macros are the key to extending the HLA language. If you're a big user of macros then you will want to read the HLA Reference Manual chapter *The HLA Compile-Time Language* to learn all about HLA's powerful macro facilities. This section will describe HLA's limited "Standard Macro" facility that is comparable to the macro facilities other assemblers provide.

You can declare macros in the declaration section of a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
    statements
#endmacro;
```

Example:

```
#macro MyMacro;
    ?i = i + 1;
#endmacro;
```

The optional parameter list must be a list of one or more identifiers separated by commas. HLA automatically associates the type "text" with all macro parameters (except for one special case noted below). Example:

```
#macro MacroWParms( a, b, c );
    ?a = b + c;
#endmacro;
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (":") and a comma separated list of identifiers, e.g.,

```
#macro ThisMacro(parm1):id1,id2;
    ...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols of the form "_XXXX_" where XXXX is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library). Example:

```
#macro LocalSym : i,j;

j: cmp(ax, 0)
   jne( i )
   dec( ax )
   jmp( j )
i:
```

```
#endmacro;
```

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

"v1" is the text for parameter #1, "x+2*y" is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand do the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro's actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is legal:

```
example( v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

2.9 The #Include Directive

Like most languages, HLA provides a source inclusion directive that inserts some other file into the middle of a source file during compilation. HLA's #INCLUDE directive is very similar to the pragma of the same name in C/C++ and you primarily use them both for the same purpose: including library header files into your programs.

HLA's include directive has the following syntax:

```
#include( string_expression );
```

2.10 The Conditional Compilation Statements (#if)

The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )
```

```
<< Statements to compile if the >>  
<< expression above is true. >>
```

```
#elseif( constant_boolean_expression )
```

```
<< Statements to compile if the >>  
<< expression immediately above >>  
<< is true and the first expres->>  
<< sion above is false. >>
```

```
#else
    << Statements to compile if both    >>
    << the expressions above are false. >>
#endif
```

The **#elseif** and **#else** clauses are optional. As you would expect, there may be more than one **#elseif** clause in the same conditional if sequence.

Unlike some other assemblers and high-level languages, HLA's conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn't recommended (because it would make your code very hard to read), it's nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

An important thing to note about this directive is that the constant expression in the **#if** and **#elseif** clauses must be of type boolean or HLA will emit an error. Any legal constant expression that produces a boolean result is legal here.

Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

2.11 The 80x86 Instruction Set in HLA

One of the most obvious differences between HLA and standard 80x86 assembly language is the syntax for the machine instructions. The two primary differences are the fact that HLA uses a functional notation for machine instructions and HLA arranges the operands in a (source, dest) format rather than the (dest, source) format used by Intel.

2.11.1 Zero Operand Instructions (Null Operand Instructions)

The following instructions do not require any operands. There are two syntactically allowable forms for each instruction:

```
instr;
instr();
```

The zero-operand instruction mnemonics are

```
aaa, aad, aam, aas, cbw, cdq, clc, cld, cli, cmc, cmpsb, cmpsd, cmpsw, cpuid, cwd, cwde, daa,
das,
insb, insd, insw, into, iret, iretd, lahf, leave, lodsb, lodsd, lodsw, movsb, movsd, movsw, nop,
outsb,
outsd, outsw, popad, popa, popf, popfd, pusha, pushad, pushf, pushfd, rdtsc, rep.insb, rep.insd,
rep.insw, rep.movsb, rep.movsd, rep.movsw, rep.outsb, rep.outsd, rep.outsw, rep.stosb,
rep.stosd,
rep.stosw, repe.cmpsb, repe.cmpsd, repe.cmpsw, repe.scasb, repe.scasd, repe.scasw,
repne.cmpsb,
repne.cmpsd, repne.cmpsw, repne.scasb, repne.scasd, repne.scasw, sahf, scasb, scasd, scasw,
stc, std, sti, stosb, stosd, stosw, wait, xlat
```

2.11.2 General Arithmetic and Logical Instructions

These instructions include `adc`, `add`, `and`, `mov`, `or`, `sbb`, `sub`, `test`, and `xor`. They all take the same basic form:

Generic Form:

```
adc( source, dest );
add( source, dest );
and( source, dest );
```

```
mov( source, dest );
sbb( source, dest );
sub( source, dest );
test( source, dest );
xor( source, dest );
```

2.11.3 The XCHG Instruction

The `xchg` instruction allows the following syntactical forms:

Generic Form:

```
xchg( source, dest );
```

2.11.4 The CMP Instruction

The `cmp` instruction uses the following general forms:

Generic:

```
cmp( LeftOperand, RightOperand );
```

Note that the `CMP` instruction's operands are ordered "dest, source" rather than the usual "source,dest" format (that is, the operands are in the same order as `MASM` expects them). This is to allow an intuitive use of the instruction mnemonic (that is, `CMP` normally reads as "compare dest to source."). We will avoid this confusion by simply referring to the operands as the "left operand" and the "right operand". Left vs. right signifies the placement of the operands around a comparison operator like "`<=`" (e.g., "left `<=` right").

2.11.5 The Multiply Instructions

HLA supports several variations on the 80x86 `MUL` and `IMUL` instructions. Some of the supported forms are:

Standard Syntax:

```
mul( src )
imul( src )
```

```
intmul( const, Reg )
intmul( const, Reg, Reg )
intmul( Reg, Reg )
intmul( mem, Reg )
```

The first, and probably most important, thing to note about HLA's multiply instructions is that HLA uses a different mnemonic for the extended-precision integer multiply versus the single-precision integer multiply (i.e., `imul` vs. `intmul`).

Note that the forms listed above correspond to the standard `mul` and `imul` instructions most assemblers provide. HLA actually provides several additional forms, please see the HLA documentation on "The 80x86 Instruction Set in HLA" for more details.

2.11.6 The Divide Instructions

HLA support several variations on the 80x86 DIV and IDIV instructions. The supported forms are:

Generic Forms:

```
div( source );  
idiv( source );
```

Note that the forms listed above correspond to the standard **div** and **idiv** instructions most assemblers provide. HLA actually provides several additional forms; please see the HLA Reference manual chapter on **The 80x86 Instruction Set in HLA** for more details.

2.11.7 Single Operand Arithmetic and Logical Instructions

These instructions include **dec**, **inc**, **neg**, and **not**. They take the following general forms (substituting the specific mnemonic as appropriate):

Generic Form:

```
dec( dest );  
inc( dest );  
neg( dest );  
not( dest );
```

2.11.8 Shift and Rotate Instructions

These instructions include **rcl**, **rcr**, **rol**, **ror**, **sal**, **sar**, **shl**, and **shr**. These instructions support the following generic syntax, making the appropriate mnemonic substitution.

Generic Form:

```
shl( count, dest );  
shr( count, dest );  
sar( count, dest );  
sal( count, dest );  
rcl( count, dest );  
rcr( count, dest );  
rol( count, dest );  
ror( count, dest );
```

2.11.9 The Double Precision Shift Instructions

These instruction use the following general form:

Generic Form:

```
shld( count, source, dest )
```

```
shrd( count, source, dest )
```

2.11.10 The Lea Instruction

These instructions use the following syntax:

```
lea( Reg32, memory )
lea( Reg32, ProcID )

lea( Reg32, LabelID )
```

Note: HLA does not support an **lea** instruction that loads a 16-bit address into a 16-bit register. That form of the **lea** instruction is not useful in 32-bit programs running on 32-bit operating systems.

2.11.11 The Sign and Zero Extension Instructions

The HLA **movsx** and **movzx** instructions use the following syntax:

Generic Forms:

```
movsx( source, dest );
movzx( source, dest );
```

2.11.12 The Push and Pop Instructions

These instructions take the following general forms:

```
pop( reg );
pop( mem );
pushw( Reg16 )
pushw( memory )
pushw( Const )

pushd( Reg32 )
pushd( memory )
pushd( Const )
```

These instructions push or pop their specified operand.

2.11.13 Procedure Calls

Given a procedure or a DWORD variable (containing the address of a procedure) named "MyProc" you can call this procedure as follows:

```
call(MyProc);
```

HLA actually supports several other syntaxes for calling procedures, including a syntax that will automatically push parameters on the stack for you. See the HLA Reference Manual chapter on *HLA Procedures* for more details.

2.11.14 The Ret Instruction

The **ret** statement allows two syntactical forms:

```
ret ( );
ret ( integer_constant_expression );
```

2.11.15 The Jmp Instructions

The HLA **jmp** instruction supports the following syntax:

```
jmp Label;
jmp ProcedureName;
jmp( dwordMemPtr );
jmp( anonMemPtr );
jmp( reg32 );
```

2.11.16 The Conditional Jump Instructions

These instructions include **ja**, **jae**, **jb**, **jbe**, **jc**, **je**, **jb**, **jge**, **jl**, **jle**, **jo**, **jp**, **jpe**, **jpo**, **js**, **jz**, **jna**, **jnae**, **jnb**, **jnbe**, **jnc**, **jne**, **jng**, **jnge**, **jnl**, **jnle**, **jno**, **jnp**, **jns**, **jnz**, **jcxz**, **jecxz**, **loop**, **loope**, **loopz**, **loopne**, and **loopnz**. They all take the following generic form (substituting the appropriate instruction for **ja**).

```
ja LocalLabel;
```

2.11.17 The Conditional Set Instructions

These instructions include: **seta**, **setae**, **setb**, **setbe**, **setc**, **sete**, **setg**, **setge**, **setl**, **setle**, **seto**, **setp**, **setpe**, **setpo**, **sets**, **setz**, **setna**, **setnae**, **setnb**, **setnbe**, **setnc**, **setne**, **setng**, **setnge**, **setnl**, **setnle**, **setno**, **setnp**, **setns**, and **setnz**. They take the following generic forms (substituting the appropriate mnemonic for **seta**):

```
seta( Reg8 );
seta( mem );
```

5.18^: The Conditional Move Instructions

These instructions include **cmova**, **cmovae**, **cmovb**, **cmovbe**, **cmovc**, **cmove**, **cmovg**, **cmovge**, **cmovl**, **cmovle**, **cmovo**, **cmovp**, **cmovpe**, **cmovpo**, **cmovs**, **cmovz**, **cmovna**, **cmovnae**, **cmovnb**, **cmovnbe**, **cmovnc**, **cmovne**, **cmovng**, **cmovnge**, **cmovnl**, **cmovnle**, **cmovno**, **cmovnp**, **cmovns**, and **cmovnz**. They use the following general syntax:

```
CMOVcc( src, dest );
```

Allowable operands:

```
CMOVcc( reg16, reg16 );
CMOVcc( reg32, reg32 );
CMOVcc( mem16, reg16 );
CMOVcc( mem32, reg32 );
```

These instructions move the data if the specified condition is true (specified by the *cc* condition). If the condition is false, these instructions behave like a no-operation.

2.11.18 The Input and Output Instructions

The **in** and **out** instructions use the following syntax:

```
in( port, al )
in( port, ax )
in( port, eax )

in( dx, al )
in( dx, ax )
in( dx, eax )

out( al, port )
out( ax, port )
out( eax, port )

out( al, dx )
out( ax, dx )
out( eax, dx )
```

The "port" parameter must be an unsigned integer constant in the range 0..255. Note that these instructions may be privileged instructions when running under 32-bit operating systems. Their use may generate a fault in certain instances or when accessing certain ports.

2.11.19 The Interrupt Instruction

This instruction uses the syntax **int**(*constant*); where the constant operand is an unsigned integer value in the range 0..255.

2.11.20 Bound Instruction

This instruction takes the following form:

```
bound( Reg16/32, mem )
```

2.11.21 The Enter Instruction

The **enter** instruction uses the syntax:

```
enter( const, const );
```

The first constant operand is the number of bytes of local variables in a procedure; the second constant operand is the lex level of the procedure. As a rule, you should not use this instruction and the corresponding **leave** instruction. HLA procedures automatically construct the display and activation record for you (more efficiently than when using **enter**). See the HLA Reference Manual chapter on *HLA Procedures* for more details on building procedure activation records.

2.11.22 CMPXCHG Instruction

This instruction uses the following syntax:

```
cmpxchg( reg/mem, reg )
```

2.11.23 The XADD Instruction

The XADD instruction uses the following syntax:

```
xadd( source, dest );
```

2.11.24 BSF and BSR Instructions

The bit scan instructions use the following syntax:

```
bsr( source, dest );  
bsf( source, dest );
```

2.11.25 The BSWAP Instruction

This instruction takes the form:

```
bswap( reg32 );
```

It converts between little endian and big endian data formats in the specified 32-bit register.

2.11.26 Bit Test Instructions

This group of instructions includes BT, BTC, BTR, and BTS. They allow the following generic forms:

```
bt( BitNumber, Dest );
```

2.11.27 Floating Point Instructions

See the HLA Reference Manual chapter *The 80x86 Instruction Set in HLA* for a complete list of the floating-point instructions and their syntax.

2.11.28 MMX and SSE Instructions

See the HLA Reference Manual chapter *The 80x86 Instruction Set in HLA* for a complete list of the MMX and SSE instructions and their syntax.

2.12 Memory Addressing Modes in HLA

HLA supports all the 32-bit addressing modes of the Intel 80x86 instruction set². A memory address on the 80x86 may consist of one to three different components: a displacement (also called an offset), a base pointer, and a scaled index value. The following are the legal combinations of these components:

2. It does not support the 16-bit addressing modes since these are not very useful under Win32.

```

displacement
basePointer
displacement + basePointer
displacement + scaledIndex
basePointer + scaledIndex
displacement + basePointer + scaledIndex

```

Note that a scaled index value cannot exist by itself.

HLA's syntax for memory addressing modes takes the following forms:

```

staticVarName

staticVarName [ constant ]

staticVarName[ breg32 ]
staticVarName[ ireg32 ]
staticVarName[ ireg32*index ]

staticVarName[ breg32 + ireg32 ]
staticVarName[ breg32 + ireg32*index ]

staticVarName[ breg32 + constant ]
staticVarName[ ireg32 + constant ]

staticVarName[ ireg32*index + constant ]

staticVarName[ breg32 + ireg32 + constant ]
staticVarName[ breg32 + ireg32*index + constant ]

staticVarName[ breg32 - constant ]
staticVarName[ ireg32 - constant ]
staticVarName[ ireg32*index - constant ]

staticVarName[ breg32 + ireg32 - constant ]
staticVarName[ breg32 + ireg32*index - constant ]

[ breg32 ]

[ breg32 + ireg32 ]
[ breg32 + ireg32*index ]

[ breg32 + constant ]

[ breg32 + ireg32 + constant ]
[ breg32 + ireg32*index + constant ]

[ breg32 - constant ]

[ breg32 + ireg32 - constant ]

```

```
[ breg32 + ireg32*index - constant ]
```

"staticVarName" denotes any static variable currently in scope (local or global).

"basereg" denotes any general purpose 32-bit register.

"breg₃₂" denotes a base register and can be any general purpose 32-bit register.

"ireg₃₂" denotes an index register and may also be any general purpose register, even the same register as the base register in the address expression.

"index" denotes one of the four constants "1", "2", "4", or "8". In those address expression that have an index register without an index constant, "*1" is the default index.

Those memory addressing modes that do not have a variable name preceding them are known as "anonymous memory locations." Anonymous memory locations do not have a data type associated with them and in many instances you must use the type coercion operator in order to keep HLA happy.

Those memory addressing modes that do have a variable name attached to them inherit the base type of the variable. Read the next section for more details on data typing in HLA.

HLA allows another way to specify addition of the various addressing mode components in an address expression - by putting the components in separate brackets and concatenating them together. The following examples demonstrate the standard syntax and the alternate syntax:

```
[ebx+2]           [ebx] [2]
[ebx+ecx*4+8]    [ebx] [ecx] [8]
lbl [ebp-2]      lbl [ebp] [-2]
```

The reason for allowing the extended syntax is because you might want to construct these addressing modes inside a macro from the individual pieces and it's much easier to concatenate two operands already surrounded by brackets than it is to pick the expressions apart and construct the standard addressing mode.

2.13 Type Coercion in HLA

While an assembly language can never really be a strongly typed language, HLA is much more strongly typed than most other assembly languages.

Strong typing in an assembly language can be very frustrating. Therefore, HLA makes certain concessions to prevent the type system from interfering with the typical assembly language programmer. Within an 80x86 machine instruction, the only checking that takes place is a verification that the sizes of the operands are compatible.

Despite HLA playing fast and loose with machine instructions, there are many times when you will need to coerce the type of some operand. HLA uses the following syntax to coerce the type of a memory location or register operand:

```
(type typeId memOrRegOperand)
```

There are two instances where type coercion is especially important: (1) when you need to assign a type other than byte, word, or dword to a register³; (2) when you need to assign an anonymous memory location a type.

3. Probably the most common case is treating a register as a signed integer in one of HLA's high level language statements. See the section on HLA High Level Language statements for more details.

3 Installing HLA

3.1 Installing HLA Under Windows

3.1.1 New Easy Installation:

You can find a program titled "hsetup.exe" on Webster. Running this application automatically installs HLA on your system. That's all there is to it. Those who wish to exercise complete control over the placement of the HLA executables can still choose to manually install HLA, but this is not recommended for first-time users.

3.1.2 Manual Installation under Windows

HLA can operate in one of several modes. In the standard mode, it converts an HLA source file directly into an object file like most assemblers. In other modes, it has the ability to translate HLA source code into another source form that is compatible with several other assemblers, such as MASM, TASM, FASM, NASM, and Gas. A separate assembler, such as MASM, can compile that low-level intermediate source code to produce an object code file. Strictly speaking, this step (converting to a low-level assembler format and assembling via MASM/FASM/GASM/Gas is not necessary, but there are some times when it's advantageous to work in this manner. Finally, you must link the object code output from the assembler using a linker program. Typically, you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.lib or hlalib_safe.lib) and, possibly, several operating system specific library files (e.g., kernel32.lib under Win32). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your system.

First, you will need an HLA distribution for your particular Operating System. These instructions describe installation under Windows; see the next section if you're using Linux, FreeBSD, or Mac OSX. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. It also includes copies of the Pelles C librarian and linker, and some other tools. These tools will let you produce executable files under Windows. However, it's still a good idea for Windows' users to grab a copy of the Microsoft linker. The easiest way to get all the Microsoft files you need is to download the *Visual C++ Express Edition* package from Microsoft's web site. As I was writing this, this package could be found at <http://www.microsoft.com/Express/vc/>.

- Here are the steps I went through to install HLA on my system:
- If you haven't already done so, download the HLA executables file from Webster at <http://webster.cs.ucr.edu>. On Webster, you can download several different ZIP files associated with HLA from the HLA download page. The "Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the HLA v2.2 "hla.zip" file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my "C:\" root directory.
- After downloading hla.zip to my C: drive, I double-clicked on the icon to run WinZip. I selected "Extract" and told WinZip to extract all the files to my C:\ directory. This created an "HLA" subdirectory in my root on C: with two subdirectories (include and lib) and two EXE files (HLA.EXE and HLPARSE.EXE. The HLA program is a "shell" program that runs the HLA compiler (HLPARSE.EXE), MASM (ML.EXE), the linker (LINK.EXE), and other programs. You can think of HLA.EXE as the "HLA Compiler".

- Next, I set some environment variables:

```
path=c:\hla;%path%
set hlalib=c:\hla\hlalib
set hlainc=c:\hla\include
```

- HLA is a Win32 Console Window program. To run HLA you must open up a console window. If you've downloaded the Visual C++ Expression Edition package, you should run the command prompt program from Visual C++ (generally found at Start→Programs→Microsoft Visual C++ Express Edition→Visual Studio Tools. If you've not installed the Visual C++ Expression Edition tools, you can usually find the command prompt program in places like Start→Programs→Accessories→Command Prompt (under Windows 2000) You might find it in another location. You can also start the command prompt processor by selecting Start→Run and entering "cmd".
- At this point, HLA should be properly installed and ready to run. Try typing "HLA -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Next, let's verify the correct operation of the linker. Type "link /?" (if you've installed the Microsoft linker as part of the Visual C++ Express Edition) or "polink /?" and verify that the linker program runs. Again, you can ignore the help screen that appears. You don't need to know about this stuff.

3.1.2.1 What You've Just Done

Before describing how to write, compile, and run your very first HLA program, it's probably worthwhile to take a quick step back and carefully consider what we've accomplished in the previous section so it's easier to troubleshoot problems that may come up. This section describes how the steps you went through in the previous section affect the execution of HLA.

Note: This section is of interest to all HLA users; whether you've installed HLA via the HLASETUP.EXE program, or you've manually installed HLA.

In order to execute HLA.EXE, HLPARSE.EXE, and various other programs needed to compile and run an HLA program, the operating system needs to be able to find all of the executable files that HLA needs. In theory, the executable files that HLA needs could be spread out all over your system as long as you tell the OS where to find every file. In practice, however, this makes troubleshooting the setup a lot more difficult if something goes wrong. Therefore, it's best to put all the necessary executables in the same directory.

Note that Windows doesn't know anything at all about the "C:\HLA" subdirectory; it's not going to automatically look for the HLA executables in this subdirectory, you have to tell Windows about this directory. This is done using the "PATH" environment variable. Whenever you tell Windows to run a program from a command prompt window, the OS first looks for a program with the given name in the current directory. If it finds the program (with a ".EXE", ".COM", or ".BAT" suffix), it will run that program from the current directory. If it does not find a program with an allowable suffix in the current directory, then the OS will use the PATH environment variable to determine which subdirectories to search through in order to find the executable program. The PATH environment variable is a (possibly empty) string that takes the following form:

pathToDirectory; pathToDirectory; pathToDirectory;...

Each *pathToDirectory* item in the list above generally represents a full path to some directory in the system. Examples include "c:\hla", "c:\windows", and "c:\bin"; these are always paths to directories, not to individual files. A PATH environment string may contain zero or more such paths; if there are two or more paths, each subdirectory path is separated from the others by a semicolon (the ellipses ["..."] above signify that you may have additional paths in the string, you don't actually place three periods at the end of the list).

When Windows fails to find an executable program you specify on the command line in the current directory, it tries searching for the executable file in each of the directory paths found in the PATH environment variable. Windows searches for the executable file in the subdirectories in the order that they appear in the environment string. That is, it searches for the executable in the first directory path first; if it doesn't find the executable in that directory, it tries the second path in the

environment string; then the third, then the fourth, etc. If Windows exhausts the list of directory paths without finding the executable file, it displays an error message. For example, suppose your PATH environment variable contains the following:

```
c:\hla; c:\windows; c:\bin
```

If you type the command "xyz file1" at the command line prompt, Windows will first search for program "xyz.exe", "xyz.com", or "xyz.bat" in the current directory. Failing to find the program there, Windows will search for "xyz" in the "C:\HLA" subdirectory. If it's not there, then Windows tries the "C:\WINDOWS" subdirectory. If this still fails, Windows tries the "C:\BIN" subdirectory. If that fails, then Windows prints an error message and returns control to the command line prompt. If Windows finds the "xyz" program somewhere along the way, then Windows runs the program and the process stops at the first subdirectory containing the "xyz" program.

The search order through the PATH environment string is very important. Windows will execute the first program whose name matches the command you supply on the command line prompt. This is why the previous section had you put "c:\hla;" at the beginning of the environment string; this causes Windows to run programs like HLA.EXE, HLPARSE.EXE, and LINK.EXE from the "C:\HLA" subdirectory rather than some other directory. This is very important! For example, there are many versions of the "LINK.EXE" program and not all of them work with HLA (and chances are pretty good that you might find an incompatible version of LINK.EXE on your system). Were you to place the "C:\HLA" directory path at the end of the PATH environment string, the system might execute an incompatible version of the linker when attempting to compile and link an HLA program. This generally causes the HLA compilation process to abort with an error. Placing the "C:\HLA" directory path first in the PATH environment variable helps avoid this problem¹.

By specifying the PATH environment variable, you tell Windows where it can find the executable files that HLA needs in order to compile your HLA programs. However, HLA and LINK also need to be able to find certain files. You may specify the location of these files explicitly when you compile a program, but this is a lot of work. Fortunately, both HLA.EXE and LINK.EXE also look at some environment variable strings in order to find their files, so you can specify these paths just once and not have to reenter them every time you run HLA.

Most HLA programs are not stand-alone projects. Generally, an HLA program will make use of routines found in the HLA Standard Library. The HLA Standard Library contains many conversion routines, input/output routines, string functions, and so on. Calling HLA Standard Library routines saves a considerable amount of effort when writing assembly language code. However, the HLA compiler isn't automatically aware of all the routines you can call in the HLA Standard Library. You have to explicitly tell the compiler to make these routines available to your programs by including one or more header files during the compilation process. This is accomplished using the HLA `#include` and `#includeonce` directives. For example, the following statement tells the HLA compiler to include all the definitions found in the "stdlib.hhf" header file (and all the header files that "stdlib.hhf" includes):

```
#include( "stdlib.hhf" )
```

When HLA encounters a `#include` or `#includeonce` directive in the source file, it substitutes the content of the specified file in place of the `#include` or `#includeonce`. The question is "where does this file come from?" If the string you specify is a full pathname, HLA will attempt to include the file from the location you specify; if it cannot find the file in the specified directory, the HLA will report an error. E.g.,

```
#include( "c:\myproject\myheader.hhf" )
```

In this example, HLA will look for the file "myheader.hhf" in the "c:\myproject" directory. If HLA fails to find the file, it will generate an error during compilation.

If you specify a plain filename as the `#include` or `#includeonce` argument, then HLA will first attempt to find the file in the current directory (the one you're in when you issued the HLA command at the command line prompt). If HLA finds the file, it substitutes the file's contents for the include directive and compilation continues. If HLA does not find the file, then it checks out the "HLAINC" environment variable, whose definition takes the following form:

```
hlainc=c:\hla\include
```

Unlike the PATH environment variable, the HLAINC environment variable allows only a single directory path as an operand. This is the path to the HLA include files directory which is "c:\hla\include" (assuming you've followed the directions in the previous section). Since HLA header files usually come in two varieties: headers associated with library routines and header files associated with the current project, the fact that HLA only provides one include path is not much of a limitation. You should keep all project-related header files in the same directory as your other source files for the project; the HLA library header files (and header files for any generic library modules you write) belong in the "C:\HLA\INCLUDE" directory. Note that if you want to place header files in some directory other than the current directory or the directory that the HLAINC environment variable specifies, you will have to specify the path to the include file in the *#include* or *#includeonce* statement.

If HLA cannot find the specified header file in the current directory or in the directory specified by the HLAINC environment variable, then you'll get an error to the effect that HLA cannot find the specified include file. Needless to say, most assemblies will fail if HLA cannot find the appropriate header files.

Since most HLA programs will use one or more of the HLA Standard Library header files, chances are good that the assembly won't be successful if the HLAINC environment string is not set up properly. Conversely, if HLAINC does contain the appropriate string, then HLA will be able to successfully compile your code to assembly and produce an object file. The last step, converting the object file to an executable, introduces another possible source of problems. The LINK program combines the object file associated with your code with HLA Standard Library and Windows library modules. Even the most trivial HLA program will need to link with (at least) one or more Windows module. (The most trivial HLA program is probably the one that immediately returns to the OS; such a program needs to call the Windows ExitProcess API in order to return to the OS, so at the very least you'll need to link with the Windows' kernel32.lib module to be able to call ExitProcess.) Once again, however, the library modules that your program needs could be anywhere on the disk. You'll have to use some environment variables to tell HLA and the linker where it can find the library modules. You accomplish this using two environment variables: one for HLA and one for the linker. We'll discuss the HLALIB environment variable first, since it's the easiest to understand.

The HLA Standard Library contains thousands of small little routines that have been combined into a single file: either "hlalib.lib" or "hlalib_safe.lib" ("hlalib_safe.lib" is a *thread-safe* version of the library). Whenever you call a particular standard library routine, the linker extracts the specific routine you call from the "hlalib.lib" library module and links combines this code with your program. Once again, the linker program and HLA don't know where to find these files, you have to tell them where to find it. This is done using the HLALIB environment variable; this environment variable contains a single pathname to the directory containing the HLALIB.LIB and HLALIB_SAFE.LIB files. I.e.,

```
set hlalib=c:\hla\hlalib
```

The important thing to note in this example is that the path is the directory containing the HLALIB.LIB and HLALIB_SAFE.LIB files. This is in direct contrast to the PATH and HLAINC environment objects where you specify only the subdirectory containing the desired files.

In addition to the "hlalib.lib" or "hlalib_safe.lib" library file, you'll also need to link your HLA programs against various Windows library modules. Basic HLA programs will need to link against the Windows' kernel32.lib, user32.lib, and gdi32.lib library modules. These library modules (plus several other Windows related library modules) are found in the Visual C++ Express Edition package; though in the previous section you should have copied these three library modules to the "c:\hla\hlalib" subdirectory. You'll need to tell the LINK.EXE program where it can find these files; this is done with the LIB environment variable. The LIB environment variable's syntax is very similar to that for the PATH environment variable. You get to specify a list of directories in the LIB environment string and LINK.EXE will automatically search for missing library files in each of the paths you specify, in the order you specify, until it finds a matching filename. By prepending "c:\hla\hlalib;" to this environment string, you tell LINK.EXE to search for library modules like KERNEL32.LIB, USER32.LIB, and GDI32.LIB in the "C:\hla\hlalib" subdirectory if it doesn't find them in the current subdirectory.

Note: in the future, if you make other Win32 API calls you may need to copy additional .LIB files from the Visual C++ Express Edition package to the "c:\hla\hllib" directory. However, for most basic HLA programs (and certainly, all console mode programs) you won't need to do this. Another alternative would be to add the path to the Visual C++ Express Edition's libraries directory to the LIB environment string (generally, installing the Visual C++ Express Edition package does this for you).

Okay, with this explanation out of the way, it's time to write, compile, and run, our first HLA program!

3.1.2.2 Running HLA

Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. A long running convention is to write a "Hello World" programs as the very first program in any language. This document will continue that cherished convention.

When writing HLA programs, the best approach is to create a single directory for each project you write. I'd suggest creating a subdirectory "c:\hla\projects" and then create a new subdirectory inside "c:\hla\projects" for each HLA project you write. For the example in this section, you might create the directory "c:\hla\projects\hw" (for "Hello World").

There are many different ways to create project directories. The most common way to do this under Windows is in the Windows Explorer (right click on a window and select "New→Folder"). However, since HLA is a command-line based tool, it's probably best to describe how to do this from the command line just to introduce some (possibly) new command line commands.

The first step is to bring up a command prompt window. Generally, this is done by selecting "Start→Programs→Accessories→Command Prompt" from the Windows Start menu. You may also select "Start→Run" and type "cmd" to bring up a command prompt shell. If you wind up writing a lot of HLA code, you'll want a faster and easier way to run the command prompt, so I recommend creating a shortcut to the command prompt program on your desktop. To do this, click on Start (and release the mouse button) and then move the mouse cursor to select "Programs→Accessories→Command Prompt". Now right-click on the "Command Prompt" menu item and drag it onto your desktop. Select "Create Shortcut(s) Here" from the resulting pop-up menu.

If you're using Visual C++ Express Edition, then select Start→Programs→Microsoft Visual C++ Express Edition→Visual Studio Tools and right-click on that menu item. You should now have a shortcut to the command prompt shell program on your desktop and you can easily run the shell by double-clicking on its icon.

After creating a shortcut, there are some things you can do to make HLA development a little easier. Right click on the shortcut you've just created and select properties. In the window that pops up, you'll probably want to change the "Start In:" string to "c:\hla\projects". This tells the command prompt program to make the specified path the current directory whenever you run the command prompt program by double clicking on this icon. By starting off inside the c:\hla\projects" subdirectory, you'll find that you save some typing (assuming, of course, you wind up putting most of your projects in the "c:\hla\projects" directory; use a different path here if this is not the case). Next, select the "layout" tab in the Command Prompt Properties window. Under "Screen Buffer Size" you'll probably want to make the value larger (the default is 300 on my system). I've found that 3000 is a good number here. This number specifies the number of lines of text that the system will save when data scrolls off the top of the screen. This lets you view up to 3,000 lines of text from program execution (including your program's output, HLA error messages, etc.). If you have a large monitor, you might also want to change the Window Size values as well. The default of 80 columns is probably fine, though you may want to expand the height to 50 lines (or whatever your monitor allows). Once you've set up the command window properties, double-click on the command prompt icon to start it running.

The first step, before doing anything else, is to verify that you've properly set up the environment variables. To check out their values, simply type "set" followed by ENTER. The "set" command without any parameters tells the command shell to dump the current values of all the environment variables active in the command prompt window. Scan through this list (scrolling back using the scroll bar if there are too many definitions to all fit in the window at one time) and search for the PATH, LIB, HLAINC, and HLALIB environment variables. Verify that they are present and have the correct values (that you've entered in the previous sections). If they're not present or the values are incorrect, HLA will not execute properly and you need to fix this problem first (Win95/98 users, did you remember to reboot after changing the autoexec.bat file?). If the environment variables are present and correct, then you're ready to try writing and running your first HLA program.

Switch to the "C:\HLA" subdirectory by using the following DOS (command line prompt) command⁵:

```
cd c:\hla
```

"cd" stands for "Change Directory". This command expects a single command line parameter that is the path of the directory that you want to make the "current directory." If you ever want to know what the current directory is, simply type "cd" without any parameters and the Command Shell will display the current directory⁶.

If you haven't done so already, it's time to create the "projects" directory where you will place the HLA projects you create. Do this with the following command:

```
mkdir projects
```

"mkdir" stands for "make directory" and this command requires a single argument - the name of the directory you want to create. If you do not specify a full pathname (as is the case in this example), the command shell creates the directory within the current directory. Verify that you've properly created the "projects" subdirectory by issuing the following DOS command:

```
dir
```

"dir" stands for "directory" and tells the command shell to display all the files in the current directory. This should list the projects directory you just created (plus all the other files and directories in the "c:\hla" directory). Note that you can also use the Windows Explorer to create directories and view the contents of directories. However, since HLA is a command prompt based application, it's useful to learn a few commands that will prove useful.

Now, switch to the projects subdirectory by using the following command:

```
cd projects
```

At this point, it's a good idea to create a new subdirectory for the "Hello World" project. Do this by using the following command:

```
mkdir hw
```

CD into this directory once you've created it. Now you're ready to begin work on the "Hello World" program.

Leaving the command prompt window open for the time being, run the editor of your choice (e.g., NOTEPAD.EXE, if you don't have any other preferences). Enter the following HLA program into the editor:

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end HelloWorld;
```

Scan over the program you've entered and verify that you've entered it exactly as written above. Save the file from your editor as hw.hla in the c:\hla\projects\hw subdirectory (generally using the File>Save or File>Save As menu item in the editor). Switch over to the command prompt window and verify that the file is present by issuing a "DIR" command; this should list the hw.hla file. If it's not present, try saving the file again and be sure to browse to the "c:\hla\projects\hw" before actually saving the file.

WARNING: NOTEPAD.EXE has a habit of tacking a ".txt" to the end of filenames you save to disk. Default installations of Windows do not display file suffixes of known file types (and ".txt" is a known type). Therefore, a directory window may show a program name like "hw.hla" when, in fact, the filename is "hw.hla.txt". Probably the number one problem people have when testing out their HLA installation is that the compiler claims it cannot find the file "hw.hla" when the user first attempts to compile the file. This is because it's really named "hla.hw.txt". You must change the filename to "hw.hla" before HLA will accept it. This is the number one HLA installation problem. Watch out for this!

Make sure you're in the same directory containing the HW.HLA file and type the following command at the "C:>" prompt: "HLA -v HW". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should output similar to the following (this changes all the time, so don't expect it to be exact):

```
HLA (High Level Assembler)
Use '-license' to see licensing information.
Version 2.0 build 407 (prototype)
Win32 COFF output
OBJ output using HLA Back Engine
-test active

HLA Lib Path:      g:\hla\hlalib\hlalib.lib
HLA include path: g:\hla\hlalibsrc\trunk\hlainc
HLA temp path:
Linker Lib Path:  C:\Program Files\Microsoft Visual Studio
9.0\VC\LIB;C:\Program Files\Microsoft SDK
s\Windows\v6.0A\lib;g:\hla\hlalib;g:\hla\hlalib

Files:
1: hw.hla

Compiling 'hw.hla' to 'hw.obj'
using command line:
[hlaparse -WIN32 -level=high -v -ccoff -test "hw.hla"]

-----
HLA (High Level Assembler) Parser
use '-license' to view license information
Version 2.0 build 406 (prototype)
-t active
File: hw.hla
Output Path: ""
hlainc Path: "g:\hla\hlalibsrc\trunk\hlainc"
hlauxinc Path: "(null)"
Compiler running under Windows OS
Back-end assembler: HLABE
Language Level: high

Compiling "hw.hla" to "hw.obj"
Compilation complete, 18758 lines,    0.454 seconds,    41317 lines/second
-----
HLA Back Engine Object code formatter

HLABE compiling 'hw.hla' to 'hw.obj'
Optimization passes: 3+2
-----
Linking via [link @"hw.link._.link"]
```

```
Microsoft (R) Incremental Linker Version 9.00.30729.01
Copyright (C) Microsoft Corporation. All rights reserved.
```

```
-heap:0x1000000,0x1000000
-stack:0x1000000,0x1000000
-base:0x4000000
-entry:HLAMain
-section:.text,ER
-section:.data,RW
-section:.bss,RW
kernel32.lib
user32.lib
gdi32.lib
-subsystem:console
-out:hw.exe
g:\hla\hlalib\hlalib.lib
hw.obj
ctl32.dll
```

If you get output that is similar to the above, you're in business.

Manually installing HLA is a complex and slightly involved process. Fortunately, the `hlasetup.exe` program automates almost everything so that you don't have to worry about changing registry settings and things like that. If you're a first-time HLA user, you definitely want to use this method to install HLA. Manual installation is really intended for upgrades as new versions of HLA appear. You do not have to change the environment variables to install a new version of HLA, simply extract the executable files over the top of your existing installation and everything will work fine.

The most common problems people have running HLA involve the location of the Win32 library files, the choice of linker, and appropriately setting the `hlalib`, `halib_safe`, and `hlainc` environment variables. During the linking phase, HLA (well, `link.exe` actually) requires the `kernel32.lib`, `user32.lib`, and `gdi32.lib` library files. These must be present in the pathname(s) specified by the LIB environment variable. If, during the linker phase, HLA complains about missing object modules, make sure that the LIB path specifies the directory containing these files. If you're a Microsoft Visual C++ user, installation of VC++ should have set up the LIB path for you. If not, then locate these files and copy them to the `HLA\HLALIB` directory. If these files are not present on your system, you should download the Microsoft Visual C++ Express edition to obtain them.

Another common problem with running HLA is the use of the wrong `link.exe` program. Microsoft has distributed several different versions of `link.exe`; in particular, there are 16-bit linkers and 32-bit linkers. You must use a 32-bit segmented linker with HLA. If you get complaints about "stack size exceeded" or other errors during the linker phase, this is a good indication that you're using a 16-bit version of the linker. Obtain and use a 32-bit version and things will work. Don't forget that the 32-bit linker must appear in the execution path (specified by the PATH environment variable) before the 16-bit linker. HLA ships with a copy of the Pelles linker (`polink.exe`) that you can use if you've not downloaded Microsoft's Visual C++ Express edition.

3.1.3 Standard Configurations Under Windows

The "standard" HLA configuration under Windows consists of `HLA.EXE`, `HLAPARSE.EXE`, `PORC.EXE`, and `POLINK.EXE`. This standard configuration generates object files directly, compiles any resource files using the Pelles C `PORC.EXE` resource compiler, and links the object modules together using the Pelles C linker (`POLINK`). The Pelles C tools were chosen for the standard configuration under Windows because they are freely distributable (unlike the Microsoft tools). For those who care about such things, `HLAPARSE.EXE` produces object modules directly. HLA's object code generator produces slightly more optimal output code than `FASM`, `MASM`, or `TASM`.

HLA also provides the ability to produce assembly language source files, in a `MASM`, `TASM`, `FASM`, `NASM`, or `Gas` format that can be assembled to object code using one of these assemblers. So why would anyone want to have HLA produce assembly language output to be run through a different assembler (much like GCC does)? For common applications, there is no need to do this.

However, in some specialized situations having this facility is quite useful. For example, rather than using the HLA back-engine native code generator, you may elect to have HLA generate FASM source code to be processed by the FASM assembler. There are three reasons for doing this:

- You want to see how HLA would translate the HLA program (in HLA syntax) to a lower-level assembly language (in FASM syntax); this is great, for example, for seeing how macros expand or how HLA processes high-level control constructs.
- If there is a defect in the internal HLA back engine that prevents HLA from directly generating an object code file, you can probably produce a source file and successfully compile your program using the external version of FASM.

Another configuration is to have HLA produce a MASM compatible output file and use Microsoft's MASM to translate that output source file into an object file. There are several reasons why you might want to use MASM:

- MASM can inject additional symbolic debugging information (usable by Visual Studio's debugger) into the object file, making it easier to debug HLA applications using Visual Studio.
- FASM and HLABE may have some code generation defect that you can't work around.
- The HLA back engine's output might not be completely compatible with some other object module tool you're using.
- You want to take HLA output and merge it with some MASM projects you have.

Although MASM is not a freely distributable program (and, therefore, is not included in the HLA download), you may download a free copy from the Microsoft Web site or obtain a copy as part of the Visual C++ Express Edition package.

Another configuration is to have HLA produce a NASM compatible output file and use the Netwide Assembler (NASM) to translate that output source file into an object file. There are a couple of reasons why you might want to use NASM:

- HLA's back engine may have some code generation defect that you can't work around.
- HLA's back engine output might not be completely compatible with some other object module tool you're using and NASM's output is compatible.
- You want to take HLA output and merge it with some NASM projects you have.
- You want to compile the code on an operating system that supports NASM but doesn't directly support HLA.

One last assembler choice under Windows is Borland's Turbo Assembler (TASM). There is one main reason why you would want to use TASM to process HLA output - you want to link HLA output with a Borland Delphi project. Delphi is very particular about the object files it will link against. Effectively, you can only use TASM-generated output files when linking with Delphi code. Therefore, if you want to link your HLA modules into a Delphi application, you'll need to use the TASM output mode. Like MASM, TASM is not a freely distributable product and cannot be included as part of the HLA download. However, Borland will provide a free copy as part of their free C++ download on their website (registration required). Note that TASM support in HLA has been deprecated and stop functioning as time passes.

Under Windows, you may use either the freely distributable Pelle's C linker (Polink) or the Microsoft linker to process the object code output from the HLA system. Polink is provided with the HLA download (subject, of course, to the Pelles C license agreement). Microsoft's linker is a commercial product (and as such, it is not included as part of the HLA download), but it is available as a free download from Microsoft's web site and as part of the Visual C++ express edition package. HLA will use either linker as the final stage in producing an executable. The Microsoft linker has been around longer and has, arguably, fewer bugs than Polink, but the choice is yours. Another possible linker option is the Borland Turbo linker (TLINK). Just note that HLA.EXE will not automatically run TLINK; you will have to run it manually after producing an OMF object file with HLA. Note that only MASM and TASM are capable of producing OMF files. FASM and HLA's internal code generator do not generate OMF object code files, so you cannot use TLINK with their output.

To produce libraries, you may optionally employ a librarian such as Microsoft's LIB.EXE, the Pelle's C POLIB.EXE, or Borland's Turbo Librarian (TLIB.EXE). The HLA.EXE program does not automatically run these programs; you will have to run them manually to create a .LIB file from your object files. Please see the documentation for these products for details on their use. The HLA download includes the POLIB.EXE program and the HLA standard library source code includes a make file option that will use any of these three librarians to produce the HLA hlalib.lib library file.

Note that it is possible to mix and match modules in the HLA system, within certain reasonable limitations. For example, you could use the FASM assembler and the Microsoft linker, the TASM assembler and the POLINK linker, or even the MASM assembler the TLINK linker. In general, FASM output works fine with the Microsoft linker and librarian or the Pelle's C linker and librarian, MASM output works best with Microsoft's linker and librarian, and Turbo assembler works best with the Borland tools or the Microsoft tools.

Under Windows, the default configuration is to generate an MSCOFF object file directly and use the POLINK linker to process the resulting object file(s). See the section on "Customizing HLA" for details on changing the default configuration.

3.2 Installing HLA Under Linux, Mac OSX, or FreeBSD (*NIX)

HLA is a compiler that translates source code into either object code or a lower-level assembly language that Gas (GNU's *as* assembler) must process. After compilation, you must link the object code output using a linker program such as the GNU ld linker. Typically, you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.a). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your system.

These instructions assume that you are using the BASH command-line shell program. If you are using a different command-line shell interpreter, you may need to adjust some of the following instructions accordingly. Note that you can run the BASH interpreter from just about any command-line shell by typing "bash" at the command line.

Mac OSX users note: the terminal window, by default, does not run the BASH shell command interpreter. You should explicitly run BASH by typing "bash" at the command-line prompt when you open up a terminal window.

First, you will need an HLA distribution for Linux, Mac OSX, or FreeBSD. Please see Webster or the previous section if you're attempting to install HLA on a different OS such as Windows. The latest version of HLA is always available on Webster at <http://webster.cs.ucr.edu>. You should go there and download the latest version if you do not already possess it.

Under Linux, Mac OSX, and FreeBSD, HLA will produce a low-level assembly language output file that you can assemble using the Free Software Foundation's Gas assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program want Gas to process it, you'll need to make sure you have a reasonable version of Gas available (Gas is available on most *NIX distributions, so this shouldn't be a problem). Note that the HLA Gas output can only be assembled by Gas v2.10 or later (so you will need the 2.10 or later binutils distribution).

Here's the steps I went through to install HLA on my Linux, Mac OSX, and FreeBSD systems:

- First, if you haven't already done so, download the HLA executables file (for Linux, Mac OSX, or FreeBSD) from Webster at <http://webster.cs.ucr.edu>. On Webster you can download several different tar.gz files associated with HLA from the HLA download page. The "Linux Executables", "Mac Executables", or "FreeBSD executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the linux.hla.tar.gz (for Linux), mac.hla.tar.gz (for Mac OSX), orbsd.hla.tar.gz (for FreeBSD) file for HLA v2.2 while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my "/usr/hla" directory; you can put the file wherever you like, though this documentation assumes that all HLA files wind up in the "/usr/hla/..."

directory tree. Note: the .tar.gz file downloads into /usr/hla. If you want the files placed somewhere else, unpack them to this directory and then move them.

- After downloading linux.hla.tar.gz, mac.hla.tar.gz, or bsd.hla.tar.gz to my root directory, I executed the following shell command: "gzip -d linux.hla.tar.gz" ("gzip -d bsd.hla.tar.gz" under FreeBSD; "gzip -d mac.hla.tar.gz" for Mac OSX). Once decompression was complete, I extracted the individual files using the command "tar xvf linux.hla.tar" ("tar xvf bsd.hla.tar" under FreeBSD, "tar xvf mac.hla.tar" under Mac OSX). This extracted several executable files (e.g., "hla" and "hlaparse") along with three subdirectories (include, hlalib, and hlalibsrc). The HLA program is a "shell" program that runs the HLA compiler (hlaparse), gas (as), the linker (ld), and other programs. You can think of hla as the "HLA Compiler". It would be a real good idea, at this point, to set the permissions on "hla" and "hlaparse" so that everyone can read and execute them. You should also set read and execute permissions on the two subdirectories and read permissions on all the files within the directories (if this isn't the default state). Do a "man chmod" from the Linux/Mac OSX/FreeBSD command-line if you don't know how to change permissions.
- If you prefer a more "Unix-like" environment, you could copy the hla and hlaparse (and other executable) files to the "/usr/bin" or "/usr/local/bin" subdirectory. This step, however, is optional
- Next, (logged in as a plain user rather than root or the super-user), I edited the ".bashrc" file in my home directory ("/home/rhyde" in my particular case, this will probably be different for you). I found the line that defined the "path" variable, it originally looked like this on my system:

```
PATH=$DBROOT/bin:$DBROOT/pgm:$PATH
```

I edited this line to add the path to the HLA directory, producing the following:

```
PATH=$DBROOT/bin:$DBROOT/pgm:/usr/hla:$PATH
```

Without this modification, *NIX will probably not find HLA when you attempt to execute it unless you type a full path (e.g., "/usr/hla/hla") when running the program. Since this is a pain, you'll definitely want to add "/usr/hla" to your path. Of course, if you've chosen to copy hla and hlaparse to the "/usr/bin" or "/usr/local/bin" directory, chances are good you won't have to change the path as it already contains these directories.

- Next, I added the following four lines to ".bashrc" (note that *NIX filenames beginning with a period don't normally show up in directory listings unless you supply the "-a" option to ls):

```
hlalib=/usr/hla/hlalib
export hlalib
hlainc=/usr/hla/include
export hlainc
```

These four lines define (and export) environment variables that HLA needs during compilation. Without these environment variables, HLA will probably complain about not being able to find include files, or the linker (ld) will complain about strange undefined symbols when you attempt to compile your programs. Note that this step is optional if you leave the library and include files installed in the /usr/hla directory subtree.

- Optionally, you can add the following two lines to the .bashrc file (but make sure you've created the /tmp directory if you do this):

```
hlatemp=/tmp
```

```
export hlatemp
```

After saving the ".bashrc" shell, you can tell *NIX to make the changes to the system by using the command:

```
source .bashrc
```

Note: this discussion only applies to users who run the BASH shell. If you are using a different shell (like the C-Shell or the Korn Shell), then the directions for setting the path and environment variables differs slightly. Please see the documentation for your particular shell if you don't know how to do this.

- At this point, HLA should be properly installed and ready to run. Try typing "HLA -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA. Enter it into a text editor and save it using the filename "hw.hla":

```
program HelloWorld;
#include( "stdlib.hhf" )
begin HelloWorld;

    stdout.put( "Hello, World of Assembly Language", nl );

end HelloWorld;
```

- Make sure you're in the same directory containing the "hw.hla" file and type the following command at the prompt: "hla -v hw". The "-v" option tells HLA to produce VERBOSE output during compilation. This is helpful for determining what went wrong if the system fails somewhere along the line. This command should produce output like the following:

```
HLA (High Level Assembler)
Use '-license' to see licensing information.
Version 2.0 build 411 (prototype)
ELF output
GAS output
-test active

HLA Lib Path:      /usr/hla/hlalib/hlalib.a
HLA include path: /usr/hla/include
HLA temp path:
Files:
1: hw.hla

Compiling 'hw.hla' to 'hw.asm'
using command line:
[hlaparse -LINUX -level=high -v -sg -test "hw.hla"]

-----
HLA (High Level Assembler) Parser
use '-license' to view license information
Version 2.0 build 411 (prototype)
-t active
File: hw.hla
Output Path: ""
```

```

hlainc Path: "/usr/hla/include"
hlaauxinc Path: "(null)"
Compiler running under Linux OS
Back-end assembler: GAS
Language Level: high

Compiling "hw.hla" to "hw.asm"
Compilation complete, 25444 lines, 0.122 seconds, 208557 lines/second
-----

Assembling "hw.asm" to "hw.o" via [as --32 -o hw.o "hw.asm"]
Linking via [ld -o "hw" "hw.o" "/usr/hla/hlalib/hlalib.a"]

```

Versions of HLA may appear for other Operating Systems (beyond Windows, Linux, FreeBSD, and Mac OSX) as well. Check out Webster to see if any progress has been made in this direction. Note a unique thing about HLA: Carefully written (console) applications will compile and run on all supported operating systems without change. This is unheard of for assembly language! Therefore, if you are using multiple operating systems supported by HLA, you'll probably want to download files for all supported OSes.

3.2.1 Standard Configurations under Linux/FreeBSD/Mac OSX

HLA supports fewer configurations under Linux, FreeBSD, and Mac OSX than under Windows but this is primarily because the main tools available for *NIX (Linux/FreeBSD/MacOSX) are all freely distributable and there is no need to support commercial tools. There are three different ways to generate object code files and only one linker and one librarian option available under Linux. There is no resource compiler (that HLA would automatically use).

HLA can generate object files in one of two different ways under *NIX:

- The hlaparse program can generate an ELF object file directly.
- The hlaparse program can generate a Gas-compatible source file that the FSF Gas assembler can convert to an ELF file.

As this was being written, HLA under Mac OS X only generates Gas-compatible source files that the Gas assembler converts to Mach-o object files. Direct output of mach-o object files should appear in HLA v2.3.

Under *NIX you don't get a choice of linkers. Everyone uses the FSF/GNU ld (load) program as the standard system linker. The HLA package also uses ld. In a similar vein, your only librarian choice is the FSF/GNU ar (archive) program. These tools work great and they're freely distributable, so they're the perfect back ends to the HLA system.

3.3 Non-Standard Configurations under Windows and Linux

It is possible, though uncommon, to use HLA in ways that aren't 100% compatible with the underlying operating system. For example, under Windows you can use HLA to produce a Gas-compatible assembly language source file. Likewise, you can use HLA under Linux to produce a MASM or TASM compatible assembly language source file. However, note that when HLA produces a Gas file, it includes certain start-up code that is only appropriate for Linux; this is true even if you do this under Windows. Similarly, producing a MASM or TASM source file includes start-up code that is only appropriate for Windows, even if the file is produced under Linux. So even if it were possible to run these products under the "wrong" operating system (e.g., MASM under Linux), the resulting object files would not be in a format acceptable to the OS and the code emitted by the HLA compiler wouldn't run properly. Nevertheless, if you just want to view the assembly language file that HLA produces, it doesn't really matter what operating system you're running under, so you may as well pick an output format with which you are most comfortable.

3.4 Customizing HLA

With environment variables you can create a customized version of HLA that suits your particular needs. The following subsections describe different ways you can optimize HLA for your personal use.

3.4.1 Changing the Location of HLA

To simplify installation and reduce installation problems, this manual suggests that you install HLA under Windows in the C:\hla subdirectory and install HLA under *NIX in the /usr/hla subdirectory. If you would prefer to put the HLA system somewhere else, it's easy to do as long as you tell the system what you're doing. This is typically accomplished by setting up a couple of environment variables.

First, to be able to run the HLA compiler and associated tools, the hla.exe/hla, hlaparse.exe/hlaparse, back-end assembler (if applicable), and linker all have to be in directories in the execution path. You may either move the HLA executables to some existing directory in the OS' execution path, or you can tell the OS to include the directory containing these files in the execution path. The standard HLA installation instructions, for example, opt for this latter case.

Under *NIX, for example, it's not uncommon for someone to put executables in the /usr/bin or /usr/local/bin directories. These directories are always in the execution path, so placing all the HLA executables in one of these directories under *NIX would spare you having to add the /usr/hla subdirectory to your execution path.

Under Windows, there is no special directory where everyone dumps their little executable files (like /usr/local/bin under *NIX). You could find an existing directory that's in the execution path and dump the HLA executables in there, however it's almost always a better idea to simply change the path environment variable so that it includes the HLA directory that contains the executables. If you've installed HLA via the HLA installation program, the install program automatically sets this up for you. However, if you want to move HLA to a different directory in the future, you will need to remove the old path to HLA from your PATH environment variable and add the path to the new HLA executables to the PATH.

Changing the execution path isn't your only concern if you decide to move HLA around. The HLA compiler will also need to know where it can find the HLA include files and the hlalib.lib standard library files. Under Windows, the linker might also want to know where the hlalib.lib file can be found. If you haven't told it otherwise, HLA under *NIX assumes that the include subdirectory and the hlalib subdirectory can be found in the /usr/hla subdirectory. Under Windows, HLA will first look in the same directory containing the HLA executables and, failing to find the include and hlalib directories there, it will then look in the C:\HLA subdirectory. If you've moved the HLA include and hlalib directories somewhere else, then you will need to set up environment variables to tell HLA where it can find these directories (technically, you could specify the paths to these directories on the HLA command-line, but that's so painful that you would never consider it for anything other than a temporary solution). The "hlainc" and "hlalib" environment variables serve this purpose.

Windows:

```
set hlainc=path_to_include_directory
```

Linux/FreeBSD/Mac OS (using BASH shell interpreter):

```
set hlainc=path_to_include_directory
export hlainc
```

Under Windows or *NIX you can use the *set* command to set the hlainc environment variable to the path where HLA can find the HLA include subdirectory. For example, if you're using Windows and you've moved the HLA include files to the C:\tools\hla\hlainc subdirectory, you could use the following command to tell HLA where it can find the include file:

```
set hlainc=c:\tools\hla\hlainc
```

The hlalib and hlalib_safe environment variables specify the *complete path* to the hlalib.lib/hlalib.a and hlalib_safe.lib/hlalib_safe.a files. Unlike the hlainc environment variable, this is not the path to the directory containing the library file, but the full path to the file itself. The reason this is a path to the library file rather than a path to the subdirectory containing the file is very simple: it's possible to have two or more library modules (in the same directory) and you might want to choose the most appropriate one for the job at hand. For example, you might have a debugging version of the library, an OMF version of the library, and a standard version of the library all in one directory. In any case, suppose the hlalib.a and hlalib_safe.a files (archive files) under *NIX are located at /usr/home/rhyde/hla/hlalib/hlalib.a and /usr/home/rhyde/hla/hlalib/hlalib_safe.a; you could tell Linux/FreeBSD/Mac OSX about this using BASH commands like the following:

```
set hlalib=/usr/home/rhyde/hla/hlalib/hlalib.a
export hlalib
set hlalib_safe=/usr/home/rhyde/hla/hlalib/hlalib_safe.a
export hlalib_safe
```

(export is a bash command that tells it to make the environment variable available to the invoking shell.)

3.4.2 Setting Auxiliary Paths

When assembling HLA source files using a back-end assembler such as MASM, FASM, Gas, or NASM, it emits a couple intermediate files for use by these back-end assemblers and the linkers. Specifically, the compilation process produces a ".asm" file for the assembler and a ".link" file for the linker. Some HLA users feel that these auxiliary files clutter up their project directory and would prefer not to see them. Fortunately, there are a couple of different ways to tell HLA to put these files in some other location besides the current project directory.

The first way to do this (which isn't really the subject of this section) is to use the "-p:<path>" command-line option to provide a temporary path for HLA to use. The advantage to using this command-line parameter is that you can set a different temporary path for each compilation. The disadvantage to this approach is that it can be a real pain to constantly set the path (if you're typing command lines manually).

A more comprehensive solution is to define the hlatmp environment variable. When HLA runs, it checks this environment variable and, if defined, uses its value to determine the path to the directory where HLA will store all temporary files. This spares you from having to place an explicit path on each command line. For example, the following command line will tell HLA to use the C:\temp (under Windows) subdirectory to hold all temporary files:

```
set hlatmp=c:\temp
```

Do take care when using the hlatmp environment variable. If you compile multiple source files with the same name (presumably from different directories), then the intermediate files they produce may create conflicts. In other words, don't use the hlatmp environment variable to specify a temporary path when doing several compilations in a batch operation. Use an explicit "-p:<path>" command-line option in those cases.

3.4.3 Setting the Default Back-End Assembler

By default, the "HLA.EXE" (Windows) or "hla" (*NIX) programs use the HLA back engine to directly produce an object file from the translation of the input HLA source file. For reasons explained earlier, you might want to override this default selection and use one of the back-end assemblers that HLA supports (MASM, TASM, NASM, or FASM under Windows, or GAS under *NIX). There are two ways to do this: via command-line parameters or by the "hlaopt" environment variable.

As described earlier, the -hlabe, -masm, -fasm, -nasm, -tasm, -gas, and -gasx command-line options let you specify which assembly language syntax and back-end assembler HLA will use to produce an object code file. The default is "-hlabe" which uses the internal HLA back engine to directly produce an object code file without using an intermediate assembly language file. The other options all produce an intermediate assembly language source file and use the associated assembler (if possible under the current operating system) to translate that assembly language source file into an object code file.

If you would like to change the default so you don't have to specify a command-line option all the time, you can use the "hlaopt" environment variable to automatically supply that command-line parameter for you. For example:

```
set hlaopt=-masm

or

set hlaopt=-gas
export hlaopt
```

4 Using HLA with the HIDE Integrated Development Environment

This chapter describes two IDEs (Integrated Development Environments) for HLA: HIDE and RadASM.

4.1 The HLA Integrated Development Environment (HIDE)

Sevag has written a nice HLA-specified integrated development environment for HLA called HIDE (HLA IDE). This one is a bit easier to install, set up, and use than RadASM (at the cost of being a little less flexible). HIDE is great for beginners who want to get up and running with a minimal amount of fuss. You can find HIDE at the HIDE home page:

<http://sites.google.com/site/highlevelassembly/downloads/hide>

Contact: sevag.krikorian@gmail.com

Note: the following documentation was provided by Sevag. Thanks Sevag!

4.1.1 Description

HIDE is an integrated development environment for use with Randall Hyde's HLA (High Level Assembler). The HIDE package contains various 3rd party programs and tools to provide for a complete environment that requires no files external to the package. Designed for a system-friendly interface, HIDE makes no changes to your system registry and requires no global environment variables to function. The only exception is ResEd (a 3rd party Resource Editor written by Ketil.O) which saves its window position into the registry.

4.1.2 Operation

HIDE is an integrated development environment for use with Randall Hyde's HLA (High Level Assembler). The HIDE package contains various 3rd party programs and tools to provide for a complete environment that requires no files external to the package. Designed for a system-friendly interface, HIDE makes no changes to your system registry and requires no global environment variables to function. The only exception is ResEd (a 3rd party Resource Editor written by Ketil.O) which saves its window position into the registry.

4.1.3 First Execution

The first time you run HIDE you may see 1 to 4 windows open, depending on the initial setup of your current version. At the very least, the main window (the one with the menu-bar) will be open. The visibility of other windows may be altered by selections in the View menu. The windows may be in either floating or docked mode. Floating windows reside outside the main window, while docked windows will be within the main window itself.

Whatever changes you make to the window positions and status will be saved and restored the next time you start HIDE.

4.1.4 The Windows

HIDE is subdivided into several Windows. A collapsable side panel contains several tool windows and an output window below displays execution details.

4.1.4.1 Editor

The HIDE editor uses KetilO's powerful RAEDIT.dll This window contains 3 buttons along the horizontal scroll bar: From left to right, Show/Hide Line Numbers, Expand All, Collapse All. Expand and Collapse work on folding text which makes navigating large documents easier. HIDE is setup to fold procedures and declaration sections.

There will be a '+' and '-' buttons in the margin to indicate text blocks that can be folded/expanded.

Near the bottom scrollbar, there may be up to three extra buttons, one expands the margin to show line numbers, the other two expand/collapse all blocks in the current window.

When a bookmark is set, you will see a blue square in the margin. Clicking on the square removes the bookmark.

The editor window also has a Splitting Bar along the top of the vertical scroll bar. This allows you to split the view into 2 windows.

4.1.4.2 Output

This window has two modes selectable by two buttons on the title bar; Notes and Output

While in Output mode, the window displays information on programs that are launched from HIDE. This includes error reports from HLA.

If the Output window is hidden when information needs to be displayed, it is opened temporarily and then closed. To cancel the automatic closing timer, click anywhere in the Output window.

While in Note mode, the window displays an edit control used for saving project notes. These notes are saved directly in the project file (.hpr)

4.1.4.3 Tool Bar

Contains various buttons for quick access. All of these also have menu-bar counterparts.

4.1.4.4 Tab Bar

When more than 1 file is open, The Tab Bar contains the filenames of all the open files allowing quick navigation.

4.1.4.5 Status Bar

Status bar contains various information on HIDE modes and files.

From the left:

Line number, number of lines

INS/OVR, insert/overwrite mode

No Project/Project, indicates if a project is currently active

Release/Debug, indicates if "Debug" or "Release" is selected in Compiler Settings

Info, if the pointer/cursor hovers over a recognized property, the property information is displayed here.

4.1.4.6 Panel

The collapsable panel on the right side hosts several windows.

Each window has a Pin button and a Close button.

To undock a window from the Panel, simply click on the title bar and drag the window out.

To dock a window back into the Panel, click its tab on the Panel.

If you wish to anchor an undocked window so that it does not dock when the its Panel tab is clicked, click on the Pin button. The icon will change to indicate window is pinned.

Click the Pin button again to deactivate anchor.

The close button hides the window and docks it back to the Panel.

4.1.4.7 Project Panel

If a project is open, this window will contain a treeview display of all the Jobs and files in the project. Double-click on a file to open it in the editor or Hex editor (depending on the file-type).

There are also other options available if you right-click the name. A menu will open offering different options are available depending on the file-type.

Also, when a file has been altered by an external program, an asterisc '*' will appear before the name.

Different file/job types have special icons for easy recognition.

A target/ghosted target indicate a target job type.

A hammer/ghosted hammer indicates a build-type job.

Ghosted indicates a held job.

A circle with a 'P' indicates a Program (or entry) source.

A square with an 'U' indicates a unit source.

A blue square with an 'RC' indicates a resource-type file.

A square with line-dashes indicates a misc or include type file.

A green circle with a 'B' indicates a binary-type file.

Items that may appear in the Right-click menu:

-Project Manager

Opens the Project Manager. This option is available with all types.

-Dedicate to wscan

Selecting this file will reserve it for being used with "Consolidate Windows Header" Project menu.

This option is available with all include file-types.

-Set Auto Open

Selects this file to open automatically when the project is loaded. This option is available with all text file-types.

-Reopen File

Reloads the file from disc. Especially useful if the file has been altered externally. This option is available with all text file-types.

-Open With Resed

Opens the file with ResEd. This option is available with resource file-types.

-Rebuild File

Forces a rebuild of the selected file. This option is available with Source and Resource file-types

-Rebuild Job

Forces a rebuild of the selected job. This option is available with Build Job-types.

-Run, Run With Debug

Runs any output for the selected job. These options are available with Program Job-types.

-Execute Script

Executes the selected kMake script. This option available with Target job-types.

-Open With kHelp

Opens the selected file with kHelp. This option available with kHelp file-types.

-View Dependents

Only available if Autodependencies is selected in HIDE Setting and the current source has dependents. This item lists all the dependents in the Output window.

4.1.4.8 Properties

If a project is open, this window will contain information on the sections of a project, including procedures, and variables. A tool bar allows selection of display mode: all project files or current open file.

4.1.5 Compiling Simple Programs

When no projects are loaded, HIDE will operate on No-Project mode, this mode is displayed on the status bar. In this mode, you may compile simple programs. This mode does not allow use of units, linking additional libraries or resources.

To use Non-Project mode, simply load an HLA sample program or write one. Use menu item File -> Open to open a new editor window. When the program is typed, you may save it then compile. By default, HIDE allows you to compile and run a sample program without saving it first, HIDE will automatically save it in the HIDE/TEMP folder as "temp.hla" The executable will also end up in the temp folder.

This default mode may be removed from the HIDE Settings dialog (in Options menu). To compile, use menu item Make -> Build, Build & Run or Rebuild.

4.1.6 Menus

When no projects are loaded, HIDE will operate on No-Project mode, this mode is displayed on the status bar. In this mode, you may compile simple programs. This mode does not allow use of units, linking additional libraries or resources.

To use Non-Project mode, simply load an HLA sample program or write one. Use menu item File -> Open to open a new editor window. When the program is typed, you may save it then compile. By default, HIDE allows you to compile and run a sample program without saving it first, HIDE will automatically save it in the HIDE/TEMP folder as "temp.hla" The executable will also end up in the temp folder.

This default mode may be removed from the HIDE Settings dialog (in Options menu). To compile, use menu item Make -> Build, Build & Run or Rebuild.

4.1.6.1 Edit

Standard: Standard cut, copy, paste options available.

Indentation:

Indent adds the tab character at the beginning of each selected line, while Outdent works the inverse.

Commentation:

Comment and Uncomment have 2 modes. With HLA files (sources ending with .hla or .hhf), the comment adds a double slash at the beginning of each selected line. Uncomment works the inverse.

Bookmarks:

Toggle bookmark adds a bookmark at the current line. The margin displays a filled circle at the location. Next/Previous Bookmarks cycle between saved bookmarks. Clear All Bookmarks removes all saved bookmarks.

These bookmarks are only good for the current session, they will not be remembered if you exit HIDE.

Source Bookmarks:

Regular bookmarks only last one session, to have persistent bookmarks, use a source bookmark. These will add a commented bookmark into the source which the HIDE properties scanner will pick up and display in the "bookmarks" view of the Properties panel. Remove a bookmark by deleting the comment.

"Insert Source Bookmark" will open a dialog asking for a label. Enter a single word to describe this bookmark. HIDE will add a comment at the current cursor position that looks like:

```
//bm=label
```

This can be entered manually as well. Next property update, the bookmark will be displayed in the "bookmarks" view and the bookmark will be saved in the source.

Block Selection:

Mark Set/End sets begin and end points for block selection. This is an alternative to shift-scrolling to select large blocks of text.

Quick Navigation to Labels:

Find/Return Declare tries to find the location in the sources where the current label under the caret is declared. The middle mouse button does the same thing except for the word currently under the caret position. There are up to 5 saved position, 4 through menus and 1 with middle mouse button. Selecting a menu again returns to the original spot.

Note: When the cursor hovers over a declared label, the property information of the label (if any) is displayed on the status bar.

Goto Program Begin jumps to the main program begin location of the Entry file.

4.1.6.2 View

Shows or hides, docks or undocks various HIDE windows:

Treeview: contains a file-list of the current project.

Toolbar: some of the commonly used menuitems are on the toolbar

Statusbar: contains some useful at-a-glance information about mode.

Output: compiling output is displayed here.

Toggle Windows: closes open Panels, opens closed Panels

HIDE Windows: hides all open Panels

Panel: Hides/shows side panel

Output: Hides/shows output panel

Dock Output: docks/undocks Output panel from main window

Cycle Panel: changes views in side panel

4.1.6.3 Project

New Project:

Opens a dialog to get a file name for creating a new project.

Enter a project name, options include using a template or starting with the Project Manager.

There are some quick templates included for convenience.

Projects will be saved in their own folder located at HIDE(default), or at the current user-specified projects folder (Options -> Set Paths). The current projects folder is displayed in the dialog.

Once a project is created, any currently open project is closed. If no templates are used, the Project Manager opens. Otherwise the template project is created and opened.

See the description of Project Manager on how to add new jobs and files to the project.

Open Project:

Opens a previously created project. If there is an active project, it is closed.

Close Project:

Closes current project.

Project Manager:

Opens the Project Manager dialog. See the Project Manager topic for more information.

Consolidate Windows Header

Selecting this option activates wscan.exe which scans your project sources for any Window header labels (namespace w). It then scans the main w.hhf header files and creates a smaller subset of this file using only the labels (direct and indirect) declared in your sources. This automatically creates a "win_inc.hhf" header file and adds it to your project. To use this file, you must #include ("win_inc.hhf") with any unit that needs them (see the wscan documents for more information).

Empty Temp Folder

This will delete the contents of the current temp folder. The location of the temp folder will vary depending on project settings. See Help->Show Environment menu for current temp folder.

4.1.6.4 Make

Build Active Job:

If the current file belongs to a job, that job is built. If it's a non-project file, the file is built.

Build Project:

This option builds all the jobs in the project, in the order displayed

in the Project Panel.

ReBuild Project:

Forces a rebuild of all the jobs without checking file-dates.

Rebuild Active File:

Forces a rebuild of the current file without checking file-dates.

Rebuild Active Job:

Forces a rebuild of the current active job without checking file-dates.

Run: Runs the executable of the current active job, if any.

Build Project & Run:

First Builds the project then runs the produced executable of the active job

Run With Debug:

Runs a previously built program of the active job with a specified Debugger.

See Options menu Set Paths for more info.

Clean:

Deletes compiled object and resource files

Test Build -Commands only:

Shows the commandlines HIDE will execute under normal execution of Make->Build Project
No actual execution will take place.

Source HLABE:

Converts the file in the active window into hla's internal HLABE format, opens a new window to display the contents.

This format is useful for seeing the high level functionality of hla unrolled into low level and debugging macros.

Source -> MASM, FASM, NASM, GAS

Converts the file in the active window into hla's translation of the various supported languages.
This format is useful for seeing how hla code appears in other assembly languages.

Some of these options are also available in the right-click menu of the Project Panel.

4.1.6.5 Tools

These tool selections are hard coded into HIDE, included in the bin folder or operating system and always available

4.1.6.5.1 Debug Window

A simple window that can display run-time output sent by your programs. Operation of Debug Window requires several steps.

1. Include the dbgwin header file (located in the hlaincfolder).
`#includeonce ("hide.hhf")`

2. Link with debug.lib This is done automatically by setting the Output Version (in Compiler Settings dialog, see Options) to Debug (debug mode).

3. Add a global compile time variable called debug to your source. This is done automatically by setting the Output Version to Debug (debug mode).

Displaying text to the debug window is done by a series of macros all of which use the "dbg" namespace.

Current functions:

`dbg.put(arg1,[arg2],[arg...]);`

Used somewhat like `stdout.put`, but less powerful.

Eg: `dbg.put("The variable MyVar contains the value: ",MyVar);`

`dbg.cls;`

Clears the debug window display of any text.

Eg: `dbg.cls;`

`dbg.putz (addr);`

Displays a zero-terminated string.

`dbg.separator;`

Draws a separating dashed line.

Eg: `dbg.separator;`

`dbg.dumpmem(address,length);`

Displays an arbitrary memory location at address passed in address parameter and of length passed in length parameter.

`dbg.dumpregs;`

Displays the contents of the registers.

`dbg.trace;`

Starts HLA trace, displayse current file/linenumber for most instructions.

`dbg.endtrace;`

Ends the trace

`dbg.timer;`

Starts a timer for measuring performance of code.

`dbg.endtimer;`

Ends the timer and displays the time.

4.1.6.5.2 Resource Editor

Launches Ketil.O's ResEd.

If there is a project open, it checks the current job and tries to find a resource file. It will open that file in ResEd.

To make sure ResEd opens a particular file, right-click the resource in the Project Panel and select "Open With ResEd"

Note: HIDE loads all files into memory. ResEd modifies external files. Although HIDE reopens the current resource file after exiting from ResEd, at times a user will have ResEd produce an additional file (such as rsrc.hhf) which will need to be manually reopened. An asterisk beside the file name in the Project view indicates a file has been modified externally and needs to be reopened, do so by right-clicking on the filename and selecting "Reopen File".

4.1.6.5.3 ASCII Table

Runs asciitbl.exe which outputs a basic text Ascii Table to the output window.

4.1.6.5.4 Calculator

Launches system calculator, calc.exe

4.1.6.5.5 Color Picker

Opens a color selection dialog. Clicking OK on the dialgo will send the current color to the main edit window at the carot postion as an HLA hex number.

4.1.6.5.6 Open Console

Runs the program attached to the "ComSpec" environment variable.

4.1.6.5.7 Run Program

Opens a dialog for input. Attempts to execute the input. You may use HIDE macros. The dialog shows the last text entered.

4.1.6.6 Options

Control all customizable aspects of HIDE, from font/color settings to output types.

4.1.6.6.1 Code Editor Font

Customize the font used in the editor and output windows.

4.1.6.6.2 Line Number Font

Customize the font used in the margin.

4.1.6.6.3 Colors & Keywords

Allows customization of highlight colors and highlight key-words. It contains 10 user key-lists which come predefined with HLA directive and assembler opcodes. There are also 4 HIDE specific lists, two of which contain keywords for HLA standard library routines and two contain keywords for Windows API structures, constants and functions.

Here you will also be able to use and save Themes. Several themes are included and your own themes may be saved. Saved themes are stored in data.ini. Only themes you save may be deleted.

4.1.6.7 HIDE Settings

This dialog allows you to change the editor behavior settings.

4.1.6.7.1 Tabs:

Tab Size: If a project is open, the new tab settings will only apply to the project. Otherwise, it will apply to default and all new projects.

Expand tabs: change tabs to spaces.

Auto indent: pressing enter will automatically tab to the previous indent.

4.1.6.7.2 Backups:

Instructs HIDE to make backups of every project file modified, up to the number specified. Once the limit is reached, the backups cycle back through the lower numbers. The files are saved either in the BAK folder of your project (if one was created during project creation), or in the HIDEfolder.

The backup naming convention is thus:

[n]<filename>where n is the backup number

If Backups is set to 0, no backups will be created.

4.1.6.7.3 Options

Hilite lines: Hilites the current cursor line

Hilite Comments: hilites commented code

Hide Divider Lines: does not show divider lines

Show Line #s: Automatically opens the line number panel for every open file

Auto HLA Structures: automatically completes or assists when HLA keywords are used

Auto Parenthesis:

Automatically completes parenthesis or '[', "
and '('. Also if the closing parenthesis is used
at the end of a word, the opening parenthesis is
automatically inserted at the beginning of the word

Keep Temp Files:

Saves temporary files in either project/temp folder
or HIDE/temp folder

Use Debug Lib: links in debug.lib and sends -ddebug ctl variable

Verbose Output: more verbose information during compiling

PE GUI: links output as Windows PE GUI

Send Console output to Output Window:

Redirects standard out of console programs executed to the output window.
Note: this does not allow input.

Auto Dependencies:

This activates the dependency checking feature of HIDE. Dependencies are considered only if the file is included and exists as a project file. To determine which dependencies HIDE will consider, you may right click on a file name in the Project View. If there are dependents, a menu item "View Dependents" will be visible. Clicking this will list dependents in Output window.

4.1.6.7.4 HIDE Global Settings

Open Last Project: Attempts to open the last project worked on

Autosave [Untitled]:

Automatically save [Untitled] as HIDE.hla *
When on No Project mode, this allows you to compile
a program without saving it first. The program is
saved in the HIDE folder with the name temp.hla,
the executable after a build will be in that folder
as well.

*Note, if there is a project open and it has 'Use Temp Folder'
active, the file will be saved in the projects's temp folder.

Use Standard Templates:

Everytime a new file is opened, depending on what kind
of file, HIDE will fill in code according to the files
located at HIDE. If you do
not like these, feel free to update them to your taste,
but create backups before upgrading HIDE, as the files
may be overwritten.

Scan for Properties:

Properties scanning is optional. Turn this off on low end systems that exhibit slowdowns during symbol scanning.

Top Window: Makes HIDE a top level window.

4.1.6.7.5 HLA Level:

This determines the level at which HLA will attempt to compile sources. 4 options available from left to right:
high, medium, low, machine

4.1.6.7.6 Global Link Settings

Allows you to edit link settings for non-project files. There must not be a project open for this to edit global settings.

4.1.6.8 SetPaths

Opens a dialog where certain HIDE and HLA paths can be set.

The fields can accept legal physical or relative paths. All HIDE macros are available for use in the fields and it is suggested that HIDE-relative macros be used for changing environment paths.

4.1.6.8.1 User Paths

Use this to extend the system search path used by HIDE

4.1.6.8.2 Project Folder

The default project folder is located at HIDE. Use this field to change the location to another folder.

4.1.6.8.3 Debugger

In this field, enter the path and executable of the debugger that will be launched when the user selects "Run With Debug"

4.1.6.8.4 Help (F1)

In this field, enter the path and name of a help file that will be used for Windows API documentation

4.1.6.8.5 hlalib

Use this field to change the "hlalib" environment path. This path must always point to a valid hlalib library.

4.1.6.8.6 hlainc

Use this field to change the "hlainc" environment path.

4.1.6.8.7 hlaopt

Use this field to change the default compiling options set by HIDE. Possible uses could be to use an external assembler.

Beware, changing this could make HIDE inoperable so only change it if you know what you're doing! You can revert to the default by deleting the "hlaopt" line in data.ini

4.1.6.9 User

The user menu is not shown unless there are user menus defined in the Data.ini file.

To define a user menu, add a [User Menu] section to HIDE.ini

Each line under "[User Menu]" has the format:

menu id, command to execute w/arguments

the command execution line may also contain HIDE macros (see below).

Eg:

```
[User Menu]
calc, calc.exe
Alpha, alpha.exe %s
```

See HIDE Macros section for more details on macros.

HIDE will also use appropriate programs to launch for opening documents from the user menu. It will determine this based on the document extension provided.

One can use this feature to open files using HIDE.

eg:

```
[User Menu]
My Help, "c:documents.pdf"
```

The above will attempt to open mypdf.pdf with default pdf viewer, if one is found.

To open files using the text/hex editor in HIDE, place the '<' (less than) char right before the filename

eg:

```
[User Menu]
HIDE.ini,<%h.ini
```

This will open "hide.ini" located in the HIDE folder using the current instance of HIDE.

Use the '>' (greater than) char right before the filename to open as HEX.

The '<' and '>' chars must appear before quotes, if quotes are used.

4.1.6.10 Help

Win32 API

Before this feature can be used, it has to be configured with the location of your Windows help file.

To do so, use Option -> Set Paths menu and add the location of your win32.hlp (or other) file to the "Help F1" field.

Once configured, selecting this menu will open win32.hlp (or other selected program). Selecting it while cursor is on a word will try to locate the keyword in win32.hlp and open that topic.

The Tutorials and manuals may need to be downloaded separately. These files are the kHelp versions of the documents. When downloaded, the tutorials go in HIDEfolder, the rest go in HIDE. When selected, kHelp is launched to display the document.

Show Environment:

Displays the current HIDE environment and macros associated with the paths.

Show Files List:

Dumps a list of all the project files, their Job segment and their locations in the Output window.

About

Opens a dialog with credits, info and other...

4.1.7 HIDE Macros

Here is a list of HIDE Macros that may be used in HIDE control boxes for paths and output redirection and in Target scripts.

%%\$	current directory, usually folder of current open project
%%h	HIDE hompath
%%p	projects folder path (not current open project)
%%c	folder for current open project
%%i	hlainc folder path
%%l	lib folder path
%%t	temp folder path
%%s	active edit source filename
%%x	template folder path

See Help-> Show Environment for the actual paths that will be substituted for some of these macros.

4.1.8 Project Manager

The Project Manager is a new feature added in HIDE 1.23.00+

From here, the details of the project are handled; adding/removing/moving/renaming jobs, files, folders, linked libraries; selecting compiling options; selecting linker options; selecting auto-maintained folder operation...

The main operation of the Project Manager is broken down into 4 panels from left to right:

1. Jobs Build/Held Queue + combobox (referred to as Jobs Combo)
2. Folders
3. Files + combobox (referred to as Files Combo)
4. Linked Objects

Most of the buttons are only active when they are useful.

On the extreme left, the up/down buttons are only active if there are two or more jobs in the project. You may select a Job and move them up or down the build Queue using these arrows.

1. Jobs List. This first listbox has two modes, Build Queue and Held. Select modes from the two buttons on top of the list.

To begin adding jobs to the project, select a job type from the Jobs Combo. There are several to choose from.

HLA Program

For small, mono-source HLA programs.

Modular Program

For large unit-based programs.

A Build Target

For kMake scripts

DLL

For modular Dynamic Linked Libraries

Library

For modular Libraries

Misc

Special type. This does not build anything.

Buttons:

Add Job - activated when a Job type is selected in the Jobs Combo.

Opens a dialog for getting the Job name.

Delete Job - deletes selected job. Only works if there are no files in the job.

Rename Job - renames selected job.

Hold Job - sends selected job into the Held list.

2. Folders List

When a project is created, this will already have the project folder listed here.

Add Folder - adds a new folder to the project

Delete Folder - deletes selected folder. Only works if the folder is empty. Does not delete the project folder.

Rename Folder - renames selected folder.

Move File - only active if there is also a file selected in the Files List. Moves the file to the folder.

3. Files List

Files Combo - only active when a job is selected. Allows you to add file-types to the job.

Note: some file-types are only available in certain jobs.

Select a file type to activate add button.

Entry File - only available in Program and DLL type jobs. Only one entry file may be created per job. The entry file is usually the one that contains the "Program.." heading.

Include File - for includes/headers

RC File - for resources

kHelp File - for kHelp documents/manuals

Binary File - for opening with Hex editor

Misc File - for general non-source text files

Unit File - only available in modular type jobs. For separately linked units.

Definition File - only available in DLLs, only one may be created.

kMake Script - only available in Target type jobs. Adds a kMake script. Only one per target.

Buttons:

Add File - opens a dialog to get name. Creates a new file. If "Use Standard Templates" is selected, the file is pre-filled with code found in the Datafolder based on type of file created.

To create a file in a folder different from the Project root, select a folder before clicking "Add File"

Delete File - removes a file from project and deletes it from disc. File is gone forever.

Rename File - give a file a new name.

Import File - Import an existing file into the project.

To move a file, select the file in the Files list, select a new folder in the Folders list and the "Move File" button will activate. Click to move.

4. Linked Objects List

This will contain a list of files linked in with the final executable/library. You may find some default libraries already listed. These are added when a new Job is created. To see the list of default libraries, or to make changes to the default list, see Data.ini

Buttons:

Add Library - add a library for linking.

Remove Library - remove a library from linking - does not delete library.

Project Options

Several buttons available:

Rename Project - renames the folder and project file.

Use Standard Templates - if selected, new files created will contain initial code taken from Data, depending on the type of file.

Use Temp Folder - creates a "temp" folder and uses that to store all temporary files.

Use Units Folder - creates a "units" folder and stores all the object files there.

Use Bak Folder - creates a "bak" folder and stores all backup files there.

Use Debug Lib - links debug.lib with all compiled jobs.

Keep Temp Files - does not delete temporary files. These files will be saved either in the current project temp folder or in the HIDE/temp folder, depending on the use of temp folder.

Verbose Compile - shows more compiling data.

Job Options:

Only available when significant jobs types are selected.

Link As GUI - links with -w option.

Link Options - opens a dialog allowing the editing of finder linker details. For advanced users only.

Output:

Shows how the final output file (if any) will be named and in which directory it will be created. You may use standard HIDE macros here for output redirection.

Done:

Closes the dialog and saves changes to project file.

4.1.9 Auto Completion

Bernd Kastenholz has written an autocompletion module for HIDE.

Autocompletion listbox is opened by pressing ctrl-space

Activation in HIDE-editor: Key Ctrl-Space

Opens a dialog beneath the cursor. The listbox contains all strings of file 'HideHomepath.txt'.

Settings: Key F1 (does only work when autocompletion dialog is visible)

Opens the settings dialog. Only the sort property of the listbox can be changed.

The settings are stored in file 'HIDE.ini'

Controlling the listbox:

Arrow up: Scroll up in the listbox

Arrow down: Scroll down

Arrow left: Increase the selection to the left

Arrow right: Decrease the selection to the right

Backspace: Increase the selection to the left

Return: Inserts the selected word and closes the dialog

Db1Clk with left MB in listbox: Inserts the selected word and closes the dialog

Scrolling with mousewheel.

ESC: Just closing the dialog and deletes the selection

HOME: Scroll to first item

END: Scroll to last item

PGDN: Scroll down one page

PGUP: Scroll up one page

Updates:

8/2/2006

- fixed a bug (while inserting chars)
- now the word list will be destroyed when changing the sort property

8/1/2006
v1.0.1

- added controlling the listbox with HOME, END, PGDN and PGUP
- fixed deleting of text
- fixed decreasing/increasing bug
- fixed backspace bug. Deletes now the selection
- cleaning the source code

7/31/2006
v1.0.0

first release of autocompletion

4.1.10 CommandLine Tools

For more in-depth documentation of these tools, see the HIDE Tools manual.

4.1.10.1 kMake

kMake is used to build target jobs. To edit a target job write in the optional [BUILD] section to avoid warnings from kMake.

eg:
[BUILD]
; commands to run

For those unfamiliar with kMake, the manual is included in Help -> HIDE Tools

For a quick summary, a script may launch any program or MS-DOS command that are available on your path. If you need to add more paths to HIDE, use Option -> Set Paths menu and add as many paths as you like to "User Paths"

The script may also contain HIDE macros (see Help -> Show Environment for an available list of macros and what they expand to).

4.1.11 Project File Format

Note: this format has changed significantly in HIDE 1.23.00 +
The original format is no longer supported and an automatic converter is provided to help in converting older projects to the newer format.

Here is a sample HPR file. I'll walk through the sections one at a time.

```
[HPR Settings]
mainfile=Src.hla
tab=6
backups=0
options=199
Project Version=10
usetemp=false
useunits=true
useback=false
findscope=1
findflags=0
```

```
[HPR Jobs]
cCalc
```

```
[HPR Folders]
units
res
src
```

```
[cCalc]
console=false
output=cCalc.exe
type=modular
main=cCalc.hla
```

```
[cCalc.link]
-heap:0xF4240,0xF4240
-stack:0xF4240,0xF4240
-base:0x4000000
-entry:HLAMain
-section:.data,RW
-section:.text,ER
-machine:ix86
```

```
[cCalc.files]
kernel32.lib,,extlinked
user32.lib,,extlinked
hlalib.lib,,extlinked
hidelib.lib,,extlinked
cCalc.hla,src,hlaprogram
cCalc.rc,res,resource
cCalc.hhf,src,include
cCalc.txt,,misc
```

```
[HPR Settings]
mainfile=Src.hla
tab=6
backups=0
options=199
Project Version=10
usetemp=false
useunits=true
useback=false
findscope=1
findflags=0
```

This section gives HIDE some details on how to treat this project

mainfile

HIDE opens this file when the project is first loaded. Any file can be set as mainfile by right-clicking the name in the Project Panel and selecting "Set Auto Open"

tab

Number of spaces in a tab

backups

Maximum number of backups reserved for this project

options

Maintains a bit-map of options

Project Version

Used for automatically updating changes to the project file format.

A version of <10 is the older project format which is no longer supported.

usetemp

useunits

useback

Some folders are auto-maintained. These show true or false whether this project makes use of these facilities.

```
usetemp = temp folder"temp"
```

```
useunits= units folder "units"
```

```
useback = backups folder"bak"
```

findscope

findflags

These maintain the find options for this project.

[HPR Jobs]

```
cCalc
```

This section lists the jobs queue. Every item here is executed in

order from first to last. There may also be a [Held Jobs] section which are jobs removed from the build queue.

[HPR Folders]

units

res

src

This section lists all the folders recognized by the project.

[cCalc]

console=false

output=cCalc.exe

type=modular

main=cCalc.hla

Each job listed in [HPR Jobs] and [Held Jobs] will have its own corresponding section which describes what the job is all about.

console

Indicates if this job will build as a console or PE GUI

output

Indicates the path/name of the output produced by this job

type

Corresponds to the type of job as selected when job was created

main

If this job has an entry (ie: it's an executable or DLL) then this will indicate which file is the entry.

[cCalc.link]

-heap:0xF4240,0xF4240

-stack:0xF4240,0xF4240

-base:0x4000000

-entry:HLAMain

-section:.data,RW

-section:.text,ER

-machine:ix86

Each job will have its own link section. This passes information to the linker. These options may be edited from the Project Manager

[cCalc.files]

kernel32.lib,,extlinked

user32.lib,,extlinked
hlalib.lib,,extlinked
hidelib.lib,,extlinked
cCalc.hla,src,hlaprogram
cCalc.rc,res,resource
cCalc.hhf,src,include
cCalc.txt,,misc

Each job will have its own files section. Every file is listed, along with its folder and type.

4.1.12 Licences

HIDE package contains tools released in various licences.

4.1.12.1 HIDE

HIDE

Copyright (c)2006 Sevag Krikorian

This licence applies to all versions of HIDE including any future released version, unless a new licence is applied.

HIDE consists of HIDE.exe as well as the package "HIDE." The package consists of other 3rd party tools and any licences associated with these tools are provided separately.

This program is free for commercial or private use as long as the following conditions are respected.

1. Redistribution

Redistribution in source or binary form is permitted.

The copyright notice, disclaimer and this licence (as well as the licences of all 3rd party software that is redistributed) must be retained.

2. Modification

The source may be modified and redistributed.

Redistribution of modified code also falls under the conditions of redistribution (see #1).

Modified source cannot be copied and placed under a different licence, including the GNU Public Licence.

3. Disclaimer

This software is provided "as is" and any express or implied warranties, including merchantability and fitness for a particular purpose are disclaimed.

The author and contributors will in no case be liable for the use or misuse of this software, including, but not limited to direct, indirect, incidental, special, exemplary or consequential damages.

4.1.12.2 PellesC

This software is provided 'as-is', without any expressed or implied warranty. In no event will the author be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to redistribute it freely, subject to the following restrictions:

- 1.The origin of this software must not be misrepresented; you must not claim that you wrote the original software.
- 2.Including Pelles C in a CDROM or in other products sold for profit needs explicit agreement from the author.
- 3.This notice may not be removed or altered from any distribution.

Pelle Orinius
Stockholm, Sweden

4.1.12.3 HLA

Use the `-license` command-line parameter to see the current license for the HLA system.

4.2 The RadASM/HLA Integrated Development Environment

Note: the HIDE integrated development environment is standard IDE for HLA. However, RadAsm, an older system that is compatible with several different assemblers, also supports HLA. I wrote this documentation for an older version of RadAsm (before HIDE appeared). As many HLA users prefer RadAsm (because they grew up with it or because they use several different assemblers and prefer using a single IDE) I've included this documentation. You may need to make appropriate mental adjustments if you're using a newer version of RadAsm than this document describes.

Last I checked, the official RadAsm development site was dead, but if you Google RadAsm you can find mirror sites to download it from. Last time I did this, found found the following sites:

http://www.oby.ro/rad_asm/

<http://radasm.cherrytree.at/radasm/>

4.2.1 Integrated Development Environments

An integrated development environment (IDE) traditionally incorporates a text editor, a compiler/assembler, a linker, a debugger, a project manager, and other development tools under the control of a single main application. *Integrated* doesn't necessarily mean that a single program provides all these functions. However, the IDE does automatically run each of these applications as needed. An application developer sees a single "user interface" to all these tools and doesn't have to learn different sets of commands for each of the components needed to build an application.

The central component of most IDEs is the editor and project manager. A *project* in a typical IDE is a related collection of files that contain information needed to build a complete application. This could, for example, include assembly language source files, header files, object files, libraries, resource files, and binary data files. The point of an IDE project is to collect and manage these files to make it easy to keep track of them.

Most IDEs manage the files specific to a given project by placing those files in a single subdirectory. Shared files (such as library and shared object code files) may appear elsewhere but the files that are only used for the project generally appear within the project directory. This makes manipulation of the project as a whole a bit easier.

RadASM, created by Ketil Olsen, is a relatively generic integrated development environment. Many IDEs only work with a single language or a single compiler. RadASM, though designed specifically for assembly language development, works with a fair number of different assemblers. The nice thing about this approach is that you may continue to use RadASM when you switch from one assembler to another. This spares you the effort of having to learn a completely new IDE should you want to switch from one assembler to another (e.g., switching between FASM, MASM, NASM, TASM, and HLA is relatively painless because RadASM supports all of these assemblers. RadASM is extremely customizable, allowing you to easily set it up with different assemblers/compilers or even modify it according to your own personal tastes. Although the version of RadASM that ships with HLA has been specifically set up to work seamlessly with RadAsm, it's nice to know that you can customize RadASM however you choose..

4.2.2 HLA Project Organization

A RadASM/HLA project is a collection of all the files specific to a given executable program. This includes project-specific source files, resource files, object files, header files, makefiles, and so on. Share library, object, and header files are logically a part of an HLA project, though they generally are not physically present in the set of files that comprise a project (i.e., you don't make copies of these files for each project you produce). Whether a given file is physically a part of the project or just logically a part of a project, a RadASM/HLA project cannot compile correctly without all the files that make up the project.

This document will adopt the (reasonable) convention of placing each HLA project in its own subdirectory. A given project directory will contain the following files and directories:

- All source files specific to the project (this includes make files, *.hla*, *.hhf*, *.rc*, *.rap* [RadASM project] and other files created specifically for this project, but does not include any standard library header or generic library files that all projects use).

- A “makefile” file to be processed by a “make” program or a batch file to compile and combine all the files in a project.
- A “Tmp” subdirectory where HLA can place temporary files it creates during compilation (normally these files wind up in the same directory as the HLA source files; placing them in the “Tmp” directory prevents clutter of the main project directory).
- A “Bak” subdirectory where backup files can be kept.

The RadASM IDE provides the ability to maintain projects directly. However, the combination of RadASM/HLA and a “make” program provides a superior solution to the standard RadASM project paradigm. Therefore, this document will assume that you’re using makefiles in your RadASM projects (the next section describes the “make” program, so if you’re not familiar with it, keep on reading...).

The drawback to using makefiles to maintain the project is that you’ve got to manually create the makefile; RadASM won’t do this for you automatically (as it does with its own projects). Fortunately, 90% of your makefile creations will simply be copying an existing makefile to your project’s directory, editing the file, and changing the filenames from the previous project to the current project (indeed, this operation is so common that you’ll find a generic makefile in the “snippets” RadASM directory accompanying the HLA download. You can easily create a copy of this generic makefile from RadASM’s “Tools > Snippets” menu, as you’ll see soon enough).

4.2.3 Using Makefiles

Although RadASM provides a true IDE for HLA that supports projects, browsing, and other nice features, the best way to manage your Win32 assembly projects (even within RadASM) is via a *makefile*. Since the use of *make* is going to be a fundamental assumption in this book (e.g., most examples will include a makefile), it’s probably wise to discuss the use of *make* here for those who may be unfamiliar with this program.

The main purpose of a program like *make* (or *nmake*, if you’re using Microsoft’s version of the program) is to automatically manage the compilation and linking of a multi-module project. Although it is theoretically possible to write a single, self-contained, assembly language source file that assembles directly to an executable file, in practice this is rarely done. Instead, programs are usually broken up into separate source files by logical function. In order to save time during development, you don’t always have to recompile every source file that makes up the application. Instead, you need only recompile those source files that have been changed (or depend upon changes in other source files). This can save a considerable amount of time during development if your project consists of many different source files that you’re compiling and linking together and you make a single change to one of these source files (because you will only have to recompile the file you’ve changed rather than all files in the system).

Note that you will have to obtain a *make* utility program in order to use *make* files. If you’ve got any Microsoft development tools, then you’ve probably got a copy of Microsoft’s *nmake.exe* program lying around. Ditto for Borland tools. The Free Software Foundation (FSF - the GNU folks) have their own version of *make* as well. If you don’t have a copy of a *make* utility, you can download Borland’s version as part of their C++ command line compiler package that they distribute free on their website (though you do have to register with Borland to receive this). Check out the C++Builder Downloads page at

http://www.borland.com/products/downloads/download_cbuilder.html

Click on the “compiler” link in order to download Borland’s command line C++ compiler (that includes the *make.exe* utility). If this link is broken, just visit <http://www.borland.com> and follow the downloads link.

Although separate compilation reduces assembly time and promotes code reuse and modularity, it is not without its own drawbacks. Suppose you have a program that consists of two modules: *pgma.hla* and *pgmb.hla*. Also suppose that you’ve already compiled both modules so that the files *pgma.obj* and *pgmb.obj* exist. Finally, you make changes to *pgma.hla* and *pgmb.hla* and compile the *pgma.hla* file *but forget to compile the pgmb.hla file*. Therefore, the *pgmb.obj* file will be *out of date* since this object file does not reflect the changes made to the *pgmb.hla* file. If you link the program’s modules together, the resulting executable file will only contain the changes to the *pgma.hla* file, it will not have the updated object code associated with *pgmb.hla*. As projects get larger they tend to have more modules associated with them, and as more programmers begin working on the project, it gets very difficult to keep track of which object modules are up to date.

This complexity would normally cause someone to recompile *all* modules in a project, even if many of the object files are up to date, simply because it might seem too difficult to keep track of which modules are up to date and which are not. Doing so, of course, would eliminate many of the

benefits that separate compilation offers. Fortunately, the make program can solve this problem for you. The make program, with a little help, can figure out which files need to be reassemble and which files have up to date *.OBJ* files. With a properly defined *make file*, you can easily assemble only those modules that absolutely must be assembled to generate a consistent program.

A make file is a text file that lists compile-time dependencies between files. An *.EXE* file, for example, is *dependent* on the source code whose assembly produce the executable. If you make any changes to the source code you will (probably) need to reassemble or recompile the source code to produce a new executable file¹.

Typical dependencies include the following:

- An executable file generally depends only on the set of object files that the linker combines to form the executable.
- A given object code file depends on the assembly language source files that were assembled to produce that object file. This includes the assembly language source files (*.HLA*) and any files included during that assembly (generally *.HHF* files).
- The source files and include files generally don't depend on anything.

A make file generally consists of a dependency statement followed by a set of commands to handle that dependency. A dependency statement takes the following form:

```
dependent-file : list of files
```

Example :

```
pgm.exe: pgma.obj pgmb.obj          --Windows make/nmake example
```

This statement says that *pgm.exe* is dependent upon *pgma.obj* and *pgmb.obj*. Any changes that occur to *pgma.obj* or *pgmb.obj* will require the generation of a new *pgm.exe* file.

The make program uses a *time/date stamp* to determine if a dependent file is out of date with respect to the files it depends upon. Any time you make a change to a file, the operating system will update a *modification time and date* associated with the file. The make program compares the modification date/time stamp of the dependent file against the modification date/time stamp of the files it depends upon. If the dependent file's modification date/time is earlier than one or more of the files it depends upon, or one of the files it depends upon is not present, then make assumes that some operation must be necessary to update the dependent file.

When an update is necessary, make executes the set of commands following the dependency statement. Presumably, these commands would do whatever is necessary to produce the updated file.

The dependency statement *must* begin in column one. Any commands that must execute to resolve the dependency must start on the line immediately following the dependency statement and each command must be indented one tabstop. The *pgm.exe* statement above would probably look something like the following:

```
pgm.exe: pgma.obj pgmb.obj
        hla -e:pgm.exe pgma.obj pgmb.obj
```

(The “-e:pgm.exe” option tells HLA to name the executable file *pgm.exe*.)

If you need to execute more than one command to resolve the dependencies, you can place several commands after the dependency statement in the appropriate order. Note that you must indent all commands one tab stop. The *make* program ignores any blank lines in a *make* file. Therefore, you can add blank lines, as appropriate, to make the file easier to read and understand.

There can be more than a single dependency statement in a *make* file. In the example above, for example, executable (*pgm.exe*) depends upon the object files (*pgma.obj* and *pgmb.obj*). Obviously, the object files depend upon the source files that generated them. Therefore, before attempting to resolve the dependencies for the executable, make will first check out the rest of the make file to see if the object files depend on anything. If they do, make will resolve those dependencies first. Consider the following make file:

1. Obviously, if you only change comments or other statements in the source file that do not affect the executable file, a recompile or reassembly will not be necessary. To be safe, though, we will assume *any* change to the source file will require a reassembly.

```
pgm.exe: pgma.obj pgmb.obj
    hla -e:pgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla
    hla -c pgma.hla

pgmb.obj: pgmb.hla
    hla -c pgmb.hla
```

The *make.exe* program will process the first dependency line it finds in the file. However, the files that *pgm.exe* depends upon themselves have dependency lines. Therefore, *make* will first ensure that *pgma.obj* and *pgmb.obj* are up to date before attempting to execute HLA to link these files together. Therefore, if the only change you've made has been to *pgmb.hla*, *make* takes the following steps (assuming *pgma.obj* exists and is up to date).

- The *make* program processes the first dependency statement. It notices that dependency lines for *pgma.obj* and *pgmb.obj* (the files on which *pgm.exe* depends) exist. So it processes those statements first.
- The *make* program processes the *pgma.obj* dependency line. It notices that the *pgma.obj* file is newer than the *pgma.hla* file, so it does *not* execute the command following this dependency statement.
- The *make* program processes the *pgmb.obj* dependency line. It notes that *pgmb.obj* is older than *pgmb.hla* (since we just changed the *pgmb.hla* source file). Therefore, *make* executes the command following on the next line. This generates a new *pgmb.obj* file that is now up to date.
- Having processed the *pgma.obj* and *pgmb.obj* dependencies, *make* now returns its attention to the first dependency line. Since *make* just created a new *pgmb.obj* file, its date/time stamp will be newer than *pgm.exe*'s. Therefore, *make* will execute the HLA command that links *pgma.obj* and *pgmb.obj* together to form the new *pgm.exe* file.

Note that a properly written *make* file will instruct the *make* program to assemble only those modules absolutely necessary to produce a consistent executable file. In the example above, *make* did not bother to assemble *pgma.hla* since its object file was already up to date.

There is one final thing to emphasize with respect to dependencies. Often, object files are dependent not only on the source file that produces the object file, but any files that the source file includes as well. In the previous example, there (apparently) were no such include files. Often, this is not the case. A more typical *make* file might look like the following:

```
pgm.exe: pgma.obj pgmb.obj
    hla -e:pgm.exe pgma.obj pgmb.obj

pgma.obj: pgma.hla pgm.hhf
    hla -c pgma.hla

pgmb.obj: pgmb.hla pgm.hhf
    hla -c pgmb.hla
```

Note that any changes to the *pgm.hhf* file will force the *make* program to recompile both *pgma.hla* and *pgmb.hla* since the *pgma.obj* and *pgmb.obj* files both depend upon the *pgm.hhf* include file. Leaving include files out of a dependency list is a common mistake programmers make that can produce inconsistent executable files.

Note that you would not normally need to specify the HLA Standard Library include files, the Standard Library ".lib" files, or any Windows library files (e.g., *kernel32.lib*) in the dependency list. True, your resulting executable file does depend on this code, but this code rarely changes, so you can safely leave it out of your dependency list. Should you make a modification to the Standard Library, simply delete any old executable and object files to force a reassembly of the entire system.

The *make* program, by default, assumes that it will be processing a *make* file named *makefile*. When you run the *make* program, it looks for *makefile* in the current directory. If it doesn't find this

file, it complains and terminates¹. Therefore, it is a good idea to collect the files for each project you work on into their own subdirectory and give each project its own *makefile*. Then to create an executable, you need only change into the appropriate subdirectory and run the make program.

The make program will only execute a single dependency in a make file, plus any other dependencies referenced by that one item (e.g., the *pgm.exe* dependency line in the previous example depends upon *pgma.obj* and *pgmb.obj*, both of which have their own dependencies). By default, the make program executes the first dependency it finds in the makefile plus any dependencies that are subservient to this first item. In particular, if a dependency line exists in the makefile that is not referenced (directly or indirectly) from the main dependency item, then make will ignore that dependency item unless you explicitly request its execution.

If you want to execute some dependency other than the first dependency in the make file, you can specify the dependency on the make command line when running make from the Windows' command prompt. For example, a common convention in make files is to create a "clean" dependency that cleans up all the files the compile creates. A typical "clean" dependency line for an HLA compilation might look like the following:

```
clean:
    del *.obj
    del *.inc
    del *.bak
```

The first thing you'll notice is that the "clean" item doesn't have a dependency list. When an item like "clean" appears without a dependency list, make will always execute the commands that follow. Another peculiarity to the "clean" dependency is that there (usually) isn't a file named *clean* in the current directory whose date/time stamp the make program can check. If a file doesn't exist, then make will assume that the file is always out of date. A common convention is to specify non-existent filenames (like *clean*) in a makefile as commands that someone would explicitly execute from within make. Of course, such usage (generally) assumes that you don't actually build a file named "clean" (or whatever name you choose to use).

Since, by default, you typically don't want to execute a command line "clean" when running make, you wouldn't usually place the *clean* dependency first in the make file (nor would you typically refer to *clean* within some other dependency list). Since make doesn't normally execute any dependency items that aren't "reachable" from the first dependency item in the make file, you might wonder how you'd tell make to execute the *clean* command. To specify the execution of some dependency other than the first (default) item in the make file, all you need to is specify the target you want to create (e.g., "clean") on the make command line. For example, to execute the *clean* command, you'd using a Windows command prompt statement like the following:

```
make clean
```

This command does not tell make to use a different make file. It will still open and use the file named *makefile* in the current directory²; however, instead of executing the first dependency it finds in *makefile*, it will search for the target "clean" and execute that dependency.

By convention, most programmers use the first dependency in a make file to build the executable based on the current build state of the program (that is, it will compile and link only those files necessary to create an up-to-date executable). Most programmers, by convention, will also include a "clean" target in their make file. The *clean* command deletes all object and intermediate files that the compiler generates; this ensures that the next build of the program will recompile every source file in the project, even if the original objects (and other targets) were up-to-date already. Doing a clean before building the application is useful when you've changed something that is not listed in the dependency lists but on which the final executable still depends (like the HLA Standard Library). Doing a *clean* is also a good way to do a sanity check when you're running into problems and you suspect that the dependency lists aren't completely correct.

-
1. The "-f*filename*" command line option that lets you specify the name of the makefile. See the manual for your version of make for details.
 2. You can tell make to use a different file by specifying the "-f" command line option. Check out make's documentation for more details.

Beyond *clean* there aren't too many "standard" target definitions you'll see programmers using in their make files, though it's common for different make files to have some additional commands beyond building the default target and cleaning up temporary compiler files. When using make with the RadASM/HLA package, however, there is an assumption that you've created the following dependencies in your make file:

build: This will be the default command (i.e., the first command appearing in the make file). It will build an executable by building any out-of-date files and linking everything together. A typical *build* dependency will look like this:

```
build: pgm.exe
```

This tells *make* to go execute the dependency for *pgm.exe* (which would normally be the default dependency in the file).

buildall: This command will rebuild the entire application. It begins by doing a clean, and then it does a build. This command generally takes the following form:

```
buildall: clean pgm.exe
```

compileRC: This command will compile any resource files into .RES files. Though the current example does not have any resource files, a typical entry in the make file might look like the following:

```
compileRC: pgm.rc
rc pgm.rc
```

syntax: This command will compile any HLA files into .ASM files just to check their syntax. Using the *pgma.hla/pgmb.hla* example given earlier, a typical compile dependency line might look like the following:

```
syntax:
hla -s pgma.hla pgmb.hla
```

run: This command will build the executable (if necessary) and then run it. The dependency line typically looks like the following:

```
run: pgm.exe
pgm <<any necessary command line parameters>>
```

clean: This is the command that deletes any compiler/assembler/linker produced temporary files, backup files, and the executable file. A typical clean command is

```
clean:
del *.obj
del *.inc
del *.bak
del tmp\*.asm
del tmp\*.inc
del pgm.exe
```

One nice feature that a standard *make* program provides is *variables*. The make program allows you to create textual variables in a make file using the following syntax:

```
identifier=<<text>>
```

All text beyond the equals sign (“=”) to the end of the physical line¹ is associated with the identifier and the make program will substitute that text whenever it encounters “\$(identifier)” in your text file. This behavior is quite similar to TEXT constants in the HLA language. As an example, consider the following *make* file fragment:

```
sources= pgma.hla pgmb.hla
executable= pgm.exe

$(executable): $(sources)
    hla -e:$(executable) $(sources)
```

Because of the textual substitution that takes place, this is equivalent to the following *make* file fragment:

```
pgm.exe: pgma.hla pgmb.hla
    hla -e:pgm.exe pgma.hla pgmb.hla
```

You can even assign variable names from the make command line using syntax like the following:

```
make executable=pgm.exe sources="pgma.hla pgmb.hla"
```

This is an important fact we’ll use because it allows us to create a generic makefile that RadASM can use to compile a given project by simply supplying the file names on the command line.

Although this section discusses the make program in sufficient detail to handle most RadASM projects you will be working on, keep in mind that the make program provides considerable functionality that this document does not discuss. For more details, consult the vendor’s documentation accompanying the version of make that you’re using. This document will assume that you’re using Borland’s *make* (version 4.0 or later) or some version of Microsoft’s *nmake*. Every make file in this book has been tested with both of these versions of make. These make files may work with other versions of *make* as well. If you don’t already have a copy of make, note that you can download Borland’s make as part of the Borland C++ 5.5 compiler (see the directions for downloading this file earlier in this section).

Because of the variations in the way different make programs work, the makefiles appearing in this document will be relatively simple, not taking advantage of too many special make features. The generic makefile we’ll usually start with looks like this:

```
build: $(hlafile).exe

buildall: clean $(hlafile).exe

compiler:
    echo No Resource Files to Process!

syntax:
    hla -s $(hlafile).hla

run: $(hlafile).exe
    $(hlafile)
    pause

clean:
    delete tmp
```

1. If you need more text than will physically fit on a single line, place a backslash at the end of the line to tell make that the line continues on the next physical line in the make file. The make program removes the new line characters between the two lines and continues processing.

```
delete *.exe
delete *.obj
delete *.link
delete *.inc
delete *.asm
delete *.map

$(hlafile).exe: $(hlafile).hla
    hla $(DEBUG) $(WINAPP) -p:tmp $(hlafile)
```

RadASM will fill in the `$(hlafile)` make variable with the project's (source file's) name. The `$(DEBUG)` variable will be filled in by RadASM (you'll see how later in this document) and will expand to an empty string if `$(DEBUG)` is not defined. The `$(WINAPP)` variable is another variable set by RadASM; it will contain the text `"-w"` if compiling a Windows GUI app, it will be the empty string if compiling a console application.

4.2.4 Installing RadASM

The easiest way to install RadASM/HLA is to run the `hla setup.exe` program found on Webster (HLA v1.58 or later). This program automatically installs HLA and RadASM, sets up appropriate environment variables, and modifies various RadASM ini files for proper use on your system. Just run `hla setup.exe`, answer a few questions about where you want the files placed, and you're in business.

For those who've already installed HLA and don't want to bother reinstalling everything, you can download the RadASM/HLA package from Webster, unzip that file, and install the code manually. The main thing you have to do is copy the RadASM directory into your `x:\hla` subdirectory and then execute the "PatchRadASM" application from within the `"x:\hla"` subdirectory. This goes in and patches all the `*hla.ini` files in the `"x:\hla\radasm"` subdirectory so that they know where the `"x:\hla"` subdirectory can be found. You may also edit these files manually and modify the line that says `"$A=C:\HLA"` so that it refers to the directory containing your HLA files and directories.

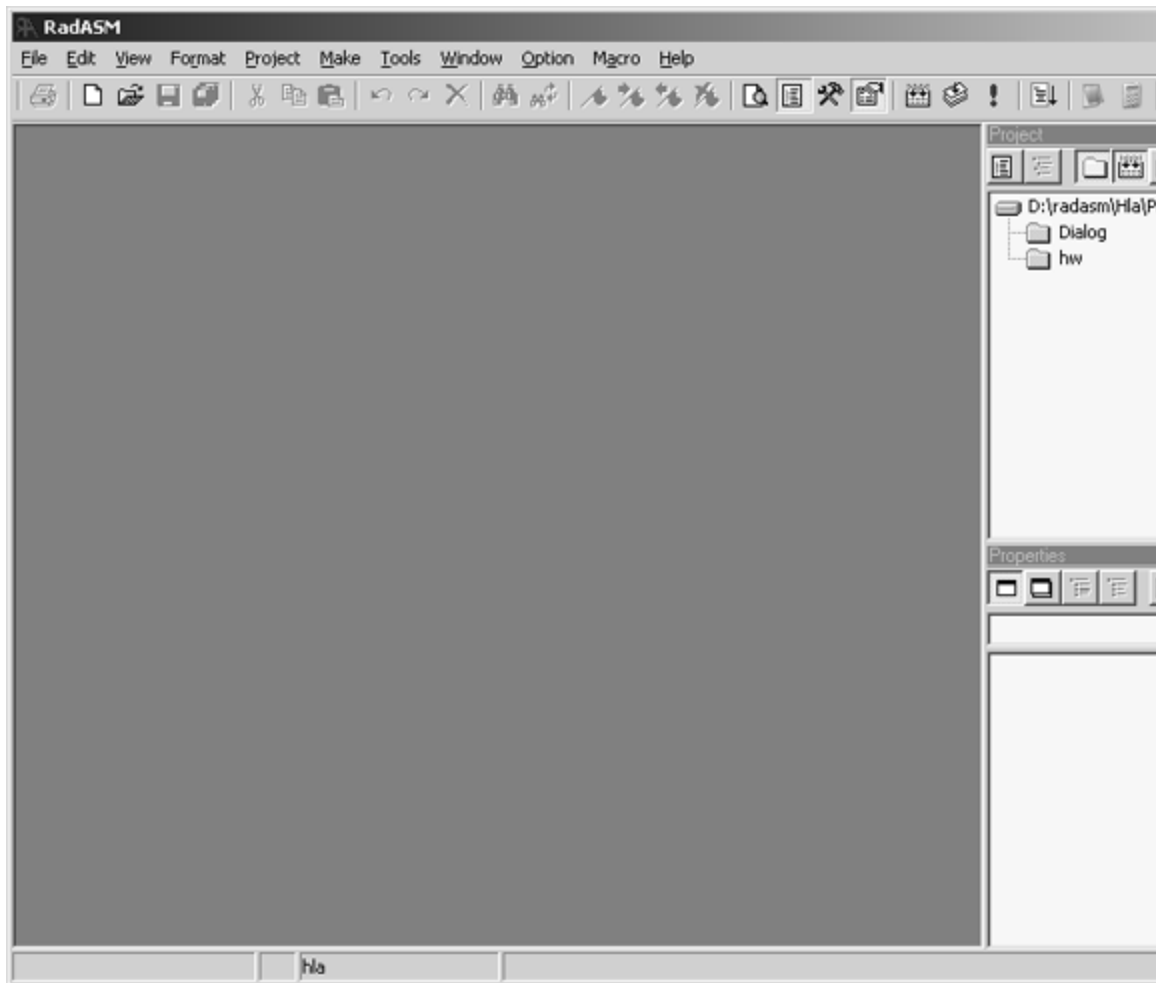
Note: Unless you're willing to learn how to customize RadASM and modify several files yourself, you *must* install the RadASM directory in the HLA subdirectory (wherever it is on the disk). If you're using RadASM with other assemblers and need to keep RadASM in some spot other than in the HLA subdirectory, please see the "RadASM customization" information at the end of this document and take a look at the `*hla.ini` files on Webster.

If you're an expert RadASM user and you only want to add HLA support to an existing RadASM setup, you can download the HLA-specific RadASM files directly from Webster and make the appropriate modifications yourself. This document will not describe how to do this; this is a task intended for advanced RadASM users only (for support, check out the RadASM forum at www.masmforum.com).

4.2.5 Running RadASM

Like most Windows applications, you can run RadASM by double-clicking on its icon or by double-clicking on a "RadASM Project" file (`".rap"` suffix). Simply double-clicking on the RadASM icon brings up a window similar to the one appearing in Figure .

RadASM Opening Screen



The main portion of the RadASM window is broken down into three panes. The larger of the three panes is where text editing takes place. The upper right hand pane is the “project management” window. The pane in the lower right hand corner lists the properties of the currently opened project.

4.2.6 The RadASM Project Management Window

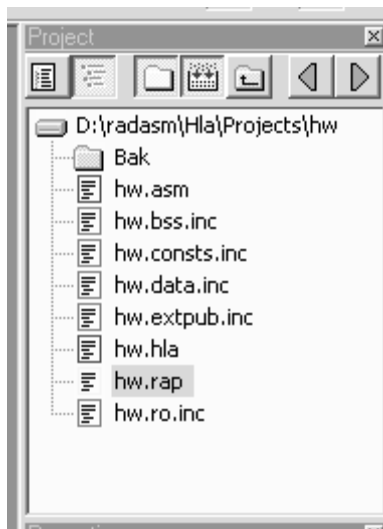
The project management window initially lists the project folders you’ve created; you can select an existing project by double-clicking on the project’s folder in this window. For example, RadASM ships with two sample projects, *Dialog* (that creates a small dialog box application) and *hw* (that creates a small “Hello World” console application). Assuming you’re running RadASM prior to creating any new projects beyond these two default projects, the Project pane will look something like Figure .

Default RadASM Project Pane



Double-clicking on the hw folder opens the folder containing that project. This changes the pane to look something like that appearing in Figure .

RadASM Project Pane With hw Folder Opened



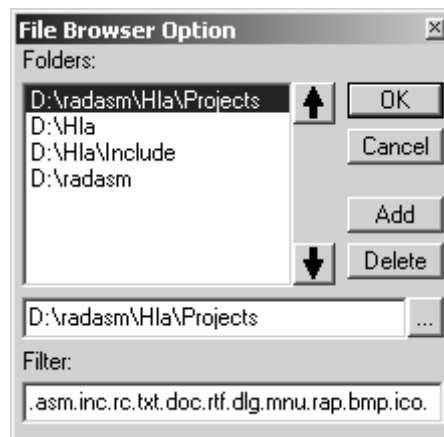
By default, RadASM does not show all the files present in the folder you've opened. Instead, RadASM filters out files that don't have a certain file suffix. By default, RadASM only displays files with the following suffixes:

- .asm
- .inc
- .rc
- .txt

- .doc
- .rtf
- .dlg
- .mnu
- .rap
- .bmp
- .ico
- .cur
- .hla
- .hhf

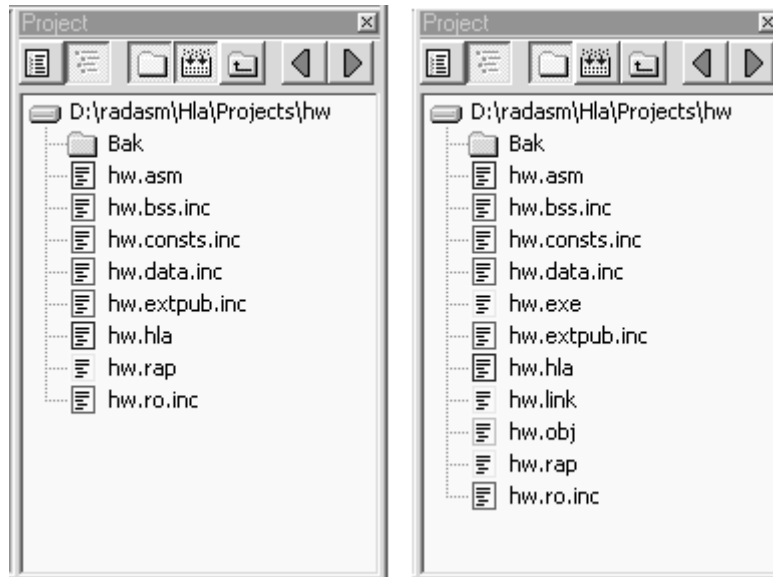
This list is actually designed to generically handle all file types for every assembler that RadASM works with. HLA users might actually want to drop “.asm” and “.inc” from this list as files with these suffixes are temporary files that HLA produces (much like “.obj” files, which don’t normally appear in this list). You can change the filter suffixes in one of two places. The first place is in the *radasm.ini* file. Search for the “[FileBrowser]” section and edit the line that begins with “Filter=...”. You can delete or add suffixes to your heart’s content on this line. The second way to change the default filters, arguably the easiest way, is within RadASM itself. From the application’s menu, select “Option>File Browser” (that is, select the “File Browser” menu item from the “Option” menu). This brings up the dialog box appearing in Figure . The text edit box at the bottom of this dialog window (labelled “Filter:”) lets you edit the suffixes that RadASM uses for filtering files in the Project window pane.

RadASM File Browser Options Dialog Box



By default, RadASM only displays those files whose file suffixes appears in the filter list. If, for some reason, you need to see all files that appear in a project subdirectory, you can turn the file filtering off. There is a toolbar button at the top of the Project window pane that lets you activate or deactivate file filtering (this is the button in the middle of the project pane, if you let the mouse cursor hover over it for a few seconds the tool-tip help displays “file filter”). Clicking on this button toggles the display mode. So clicking on this button once will deactivate file filtering, to display all the files in the directory, clicking on this button a second time reactivates file filtering. Figure shows the effects of clicking on this button.

File Filtering in RadASM's Project Pane



If you've descended into a subdirectory by double-clicking on its folder icon and you decide to return to an upper level directory, you can move to that upper level directory by clicking on the "Up One Level" button in the RadASM Project pane

The left and right arrow buttons allow you to quickly scan through several different directories in the system. By default, RadASM displays a couple of interesting (HLA-related) subdirectories in the Project pane when you scan through the list using the left and right arrows in the Project pane. In general, however, you'll want to customize the directories RadASM visits when you press these two arrow buttons. You can add (or change) directory paths in the "[FileBrowser]" section of the `radasm.ini` file, though it's probably easier to select the "Option>File Browser" menu item to open up the File Browser Option dialog box and make your changes there (see Figure). The "Folders:" list in the File Browser Option dialog box lists all the directories that RadASM will rotate through when you press the left and right buttons in the Project window pane. You can add, delete, edit, and rearrange the items in this list.

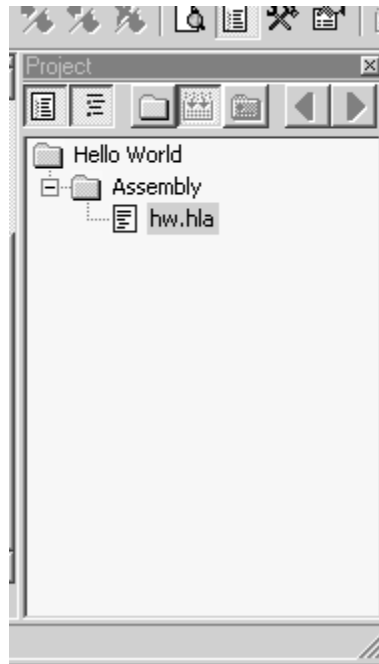
To edit an existing entry, click on that entry with the mouse and then edit the directory path appearing in the text edit box immediately below the "Folders:" list. You may either type in the path directly, or browse for the path by pressing the "browse" button immediately to the right of the text entry box.

To delete an entry from the File Browser Option list, select that item with the mouse and then press the "Delete" button appearing in the File Browser Option Window. To add a new entry to the list, press the "Add" button and then type the path into the text edit box (or use the browse button to locate the subdirectory you want to add). **Note:** do not type the new entry in and then press "Add". This sequence will change the currently selected item and then add a new, blank, entry. The correct sequence is to first press the "Add" button, and then edit the blank entry that RadASM creates.

The remaining buttons in the Project window are only applicable to open projects. Note that opening a project folder is not the same thing as opening a RadASM project. To open a RadASM project you must either create a new project or open an existing ".rap" file. For example, you can open the "Hello World" project in the `hw` directory by double-clicking on the `hw.rap` file that appears in the project window. Opening the `hw.rap` file does two things to the RadASM windows: first, it displays the `hw.hla` source file in the editor window and, second, it switches the Project window pane from "File Browser mode" to "Project Browser mode." In project browser mode RadASM displays only the files you've explicitly added to the project. Any incidental or generated files will not appear here (unless you explicitly add them). For example, whereas the "File Browser" mode displays several ".inc" and ".asm" files (assuming you've not removed these suffixes from the file filter), the "Project Browser mode" only displays the `hw.hla` file because this is the only file that was originally added to the project. Another difference between the file browser

and project browser modes is the fact that RadASM displays the files in “pseudo-directories” according to the file’s type. For example, it displays the *hw.hla* file under the sub-heading “Assembly” (see Figure). The *hw.rap* project is a relatively simple project, only having a single assembly file. The *Dialog.rap* project (that appears in the “Dialog” project folder) is a slightly more complex application, having a couple of resource files in addition to an assembly file (see Figure). Note that you can “flatten” RadASM’s view of these files by pressing the “Project Groups” button in the Project window pane (see Figure). Pressing this button a second time restores the project groups display (remember, you can always determine which button is which by letting the mouse cursor float above each button for a few seconds).

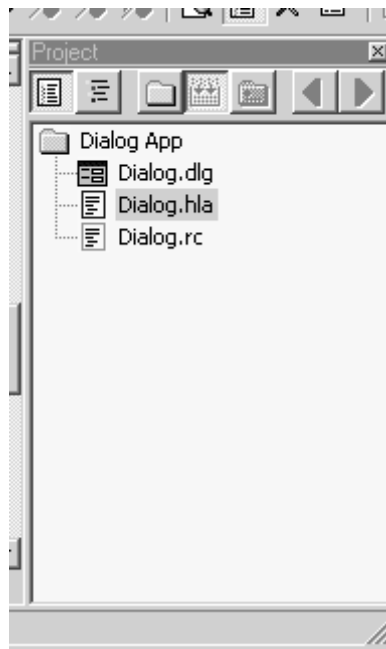
Project Window “Project Browser Mode”



Dialog.rap Project Browser Display

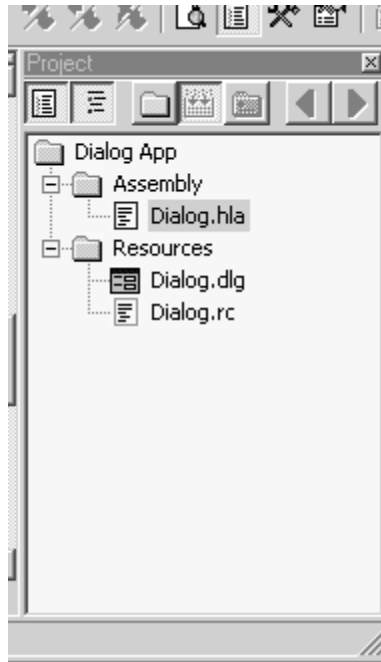


Effect of Pressing the “Project Groups” Button



When you've got a project loaded, RadASM displays the project view by default. By pressing the "File Browser" and "Project Browser" buttons in the Project window pane, you can switch between these two views of your files (see Figure).

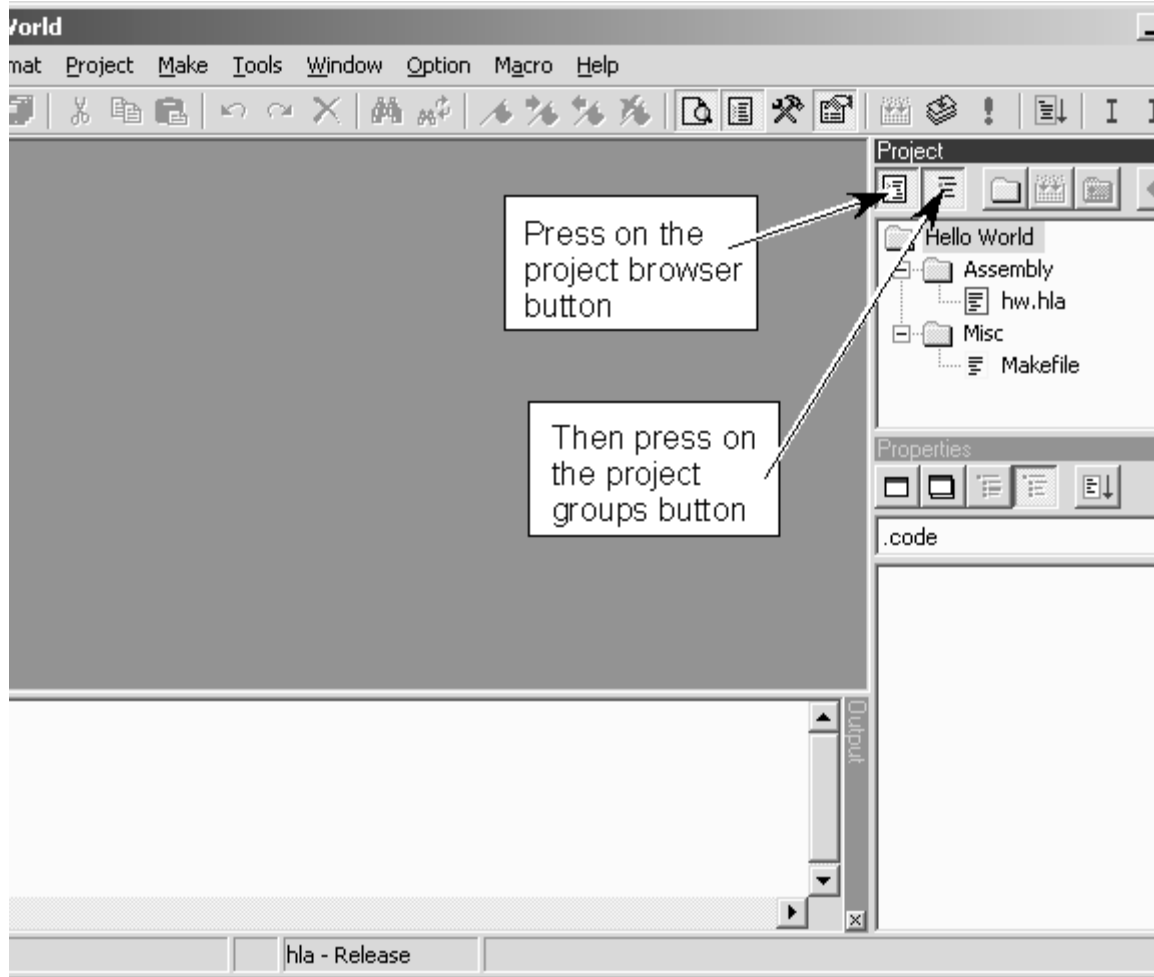
The Project Browser and File Browser Buttons



4.2.7 Compiling and Executing an Existing RadASM Project

To see how to use RadASM to compile and run a simple HLA program, begin by double-clicking on the *hw.rap* file. This is found in the ...Radasm\hla\projects\hw folder. When RadAsm opens up, you should see a display similar to Figure ; if not, then press the project browser and project groups buttons.

Selecting the HW.HLA Project



Just for fun, bring up the *hw.hla* program into the main editor by double-clicking on the *hw.hla* file icon in the project manager window. Here's what that file looks like:

```
program HelloWorld;
#include( "stdlib.hhf" )

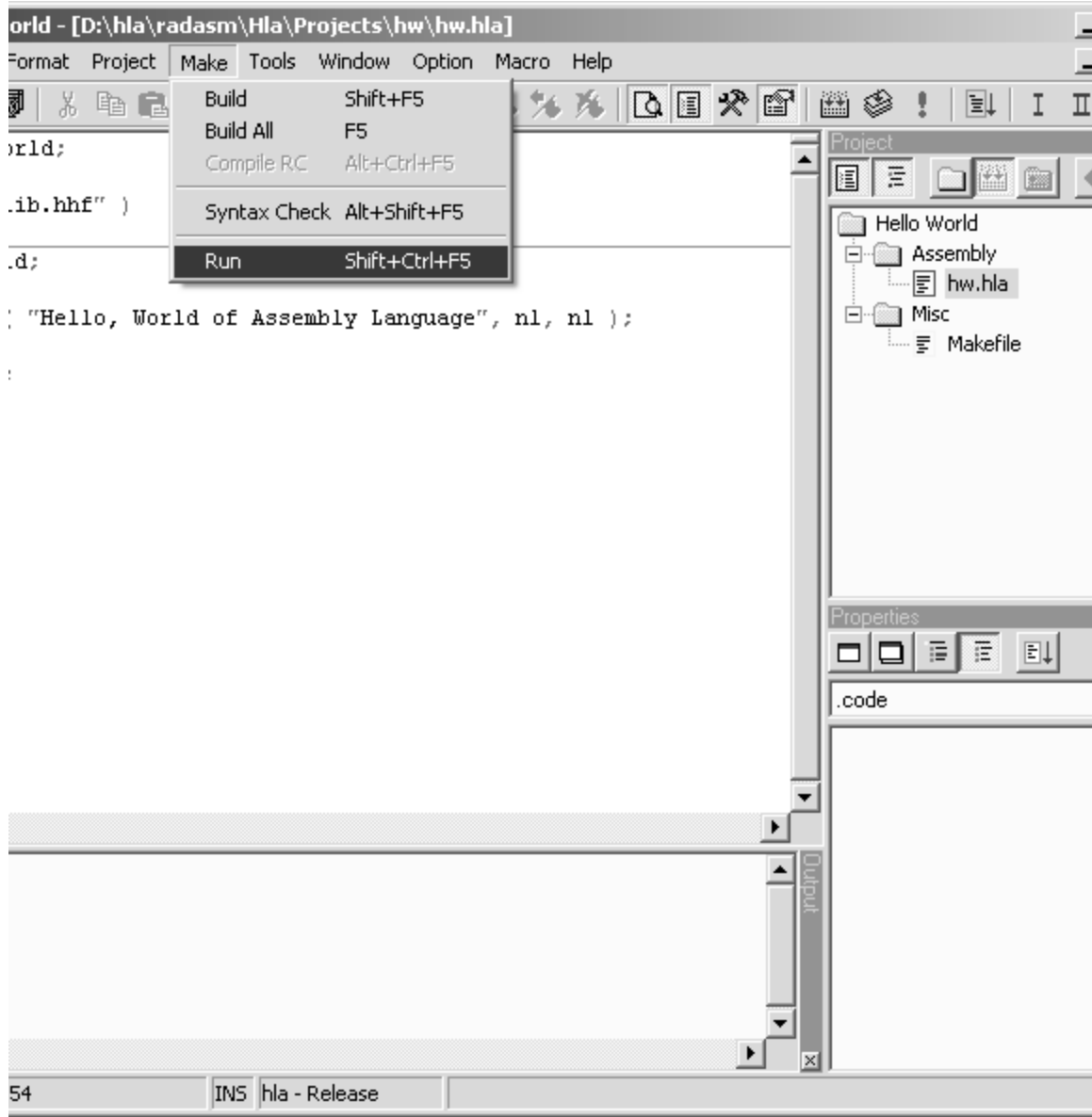
begin HelloWorld;

    stdout.put( "Hello, World of Assembly Language", nl, nl );

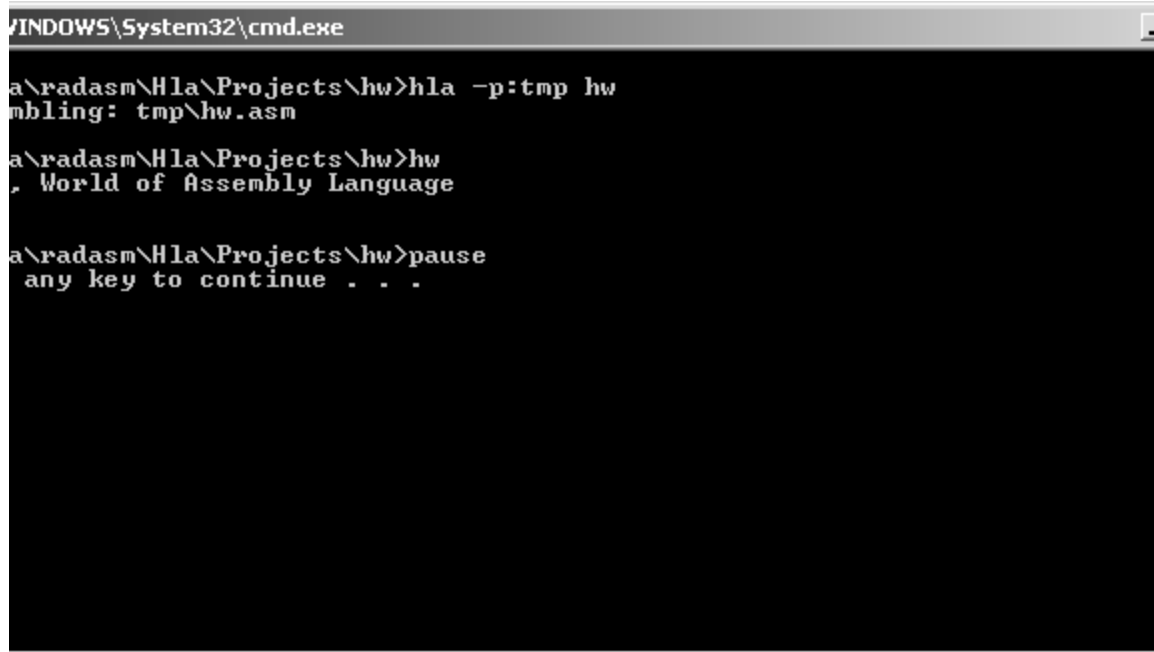
end HelloWorld;
```

To run the Hello World program from RadASM, simply select the “Run” entry from the Make menu (see Figure). This produces the program output found in Figure . When you press the enter key, the console window will close and control returns to RadASM.

Running the Hello World Program From RadASM



HW.HLA Program Output



```
WINDOWS\System32\cmd.exe
a\radasm\Hla\Projects\hw>hla -p:tmp hw
Linking: tmp\hw.asm

a\radasm\Hla\Projects\hw>hw
, World of Assembly Language

a\radasm\Hla\Projects\hw>pause
any key to continue . . .
```

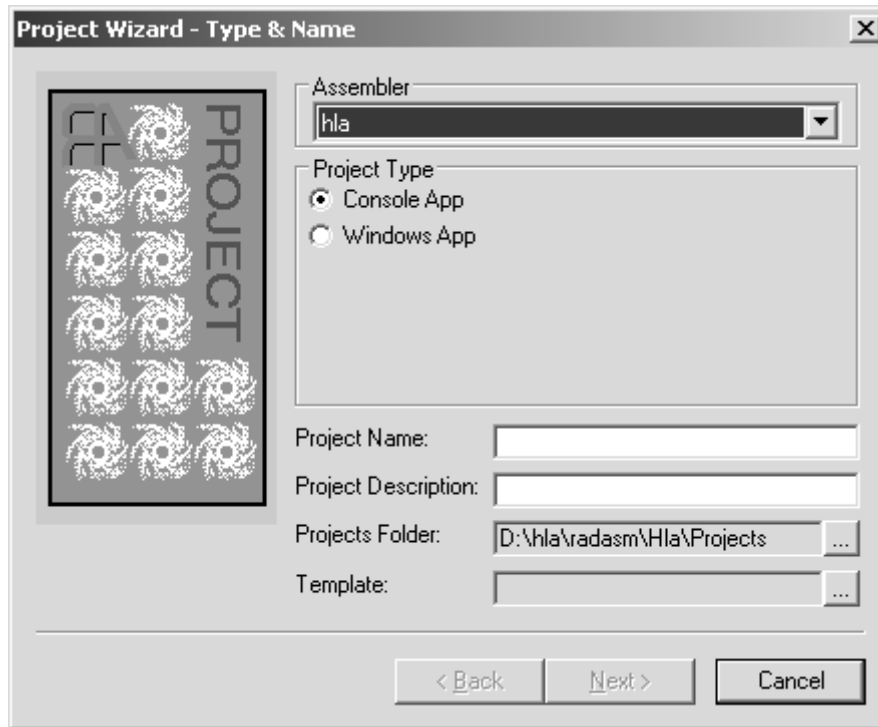
The other options in the RadASM “Make” menu have the following effect:

- Build- compiles the project; if you’re using makefiles in your RadASM projects, this option will only compile those files absolutely necessary to create the executable.
- Build All- cleans out all the old object and executable files and rebuilds the executable from scratch.
- Compile Resource- Used to compile any resource files associated with this project (note that Build and Build All will also compile the resource files, if necessary).
- Syntax Check - does a syntax compile on the HLA files, without actually building the whole application. Faster than building the whole project if you simply want to check for typos in your source code.
- Run - Runs the executable (if you’re using makefiles, this will also build the application if the executable is not current; if you’re not using makefiles, you must manually build the application before running it).

4.2.8 Creating a New Project in RadASM

While the two default projects that RadASM supplies are useful for demonstrating the RadASM Project window pane, you’re probably far more interested in creating your own RadASM/HLA projects. Creating your own project is a relatively straight-forward process using RadASM’s *project creation wizard*. To begin this process, select the “File>New Project” menu item. This opens the project wizard dialog box (see Figure).

RadASM Project Wizard Dialog Box



The “Assembler” pop-up menu list lets you select the assembler that you want to use for this project. Remember, RadASM supports a variety of different assemblers and the “rules” are different for each one. Because you’re probably using HLA (if you’re reading this document), you’ll want to select the HLA assembler from this list. HLA should be the default (in fact, only) assembler in this list. If you’re not using the *radasm.ini* file supplied on Webster, then you should make sure that HLA appears first in this list in the *radasm.ini* file.

The “Project Type” group is a set of radio buttons that let you select the type of project you’re creating. RadASM populates this list of radio buttons from the “[Project]” section of the *hla.ini* file. The “Type=...” statement in this section specifies the valid projects that RadASM will create. RadASM creates the radio button items in the order the project type names appear in the “Type=...” list; the first item in the list is the one that will have the default selection. If you’re going to be developing Windows’ GUI applications most of the time, you’ll probably want to change this list so that “Windows App” appears first in the list. This will slightly streamline the use of the Project Wizard because you won’t have to explicitly select “Windows App” every time you create a new Windows application. The standard default is a Console App because that’s the type of program most beginning HLA programmers create. You can actually add new project types to this list by modifying the *hla.ini* file. However, most HLA programmers will be creating either Win32 GUI apps or Win32 console apps, hence the standard release of RadASM/HLA supports these two application types. If you want to create your own project types, see the discussion on customizing RadASM later in this manual.

The “Project Name:” text entry box is where you specify the name of the project you’re creating. RadASM will create a folder by this name and any other default files it creates (within the project folder) will also have this name as their filename prefix. The text you enter at this point must be a valid Windows filename. Note that this should be a simple file name, not a path. You’ll supply the path to this file/directory in a moment. This name should be a *base* filename (that is, no extension). RadASM will create other filenames by attaching appropriate extensions to the name you supply here (e.g., “.hla” and “.exe”). So if you specify a name like “myProject” here, RadASM will create a directory named “myProject” to hold your files and it will also create a “myProject.hla” file (among other files). When you actually build your program, RadASM (by default) will create an executable named “myProject.exe”.

The “Project Description:” text entry box allows you to place a descriptive comment that describes the project. This is any arbitrary text you choose. It should be a brief (one-line) description of the project.

The “Projects Folder:” text entry box is where you select the path to the spot in the file system where RadASM will create the project folder. You can type the path in directly, or you can press the browse button to the right of this text entry box and use a Windows’ dialog box to select the subdirectory that will hold the project’s folder.

The “Template:” text entry box and browse button lets you select a template for your project. If you don’t select a template, then RadASM will create an empty project for you (i.e., the main “.hla” file will be empty). If you select one of the templates (e.g., the “.tpl” files found in the *RadASM\Hla\Templates* directory) then RadASM will create a “skeletal” project based on the project template you’ve chosen. Table lists some of the typical templates you will find.

RadASM/HLA Templates

Table 1:

Template Selection	Available if this project type is selected	Result
consApp.tpl	Console App	RadASM will create a simple console application. Builds are handled strictly by RadASM. Good for simple (one-file) HLA projects.
consAppBatch.tpl	Console App	RadASM will create a simple console application. Builds are handled by running one of several batch files (also created by this template) including build.bat, compilerc.bat, syntax.bat, and run.bat. By default, these batch files process a simple (one-source-file) project, but you can edit the batch files to handle more complex projects.
consAppMake.tpl	Console App	RadASM will create a simple console application. Builds are handled by running make.exe on a makefile that this template creates.
consAppNMake.tpl	Console App	Builds a project just like consAppMake.tpl except that it invokes Microsoft’s nmake.exe program rather than a generic make.exe program.
win32App.tpl	Windows App	RadASM will create a generic Win32 GUI project. Builds are handled strictly by RadASM. Good for simple (one-HLA-file) HLA projects.
win32AppBatch.tpl	Windows App	RadASM will create a generic Win32 GUI project. Builds are handled by running one of several batch files (also created by this template) including build.bat, compilerc.bat, syntax.bat, and run.bat. By default, these batch files process a simple (one-HLA-source-file) project, but you can edit the batch files to handle more complex projects.
win32AppMake.tpl	Windows App	RadASM will create a generic Win32 GUI project. Builds are handled by running make.exe on a makefile that this template creates.

Table 1:

Template Selection	Available if this project type is selected	Result
win32AppNMake.tpl	Windows App	Builds a project just like win32AppMake.tpl except that it invokes Microsoft's nmake.exe program rather than a generic make.exe program.
WPAApp.tpl	Windows App compatible with code from "Windows Programming in Assembly"	RadASM will create a Win32 GUI project based on the structure of the code described in "Windows Programming in Assembly". These projects use the "wpa.hhf" header file and the "winmain.lib" library module described in Randy Hyde's book "Windows Programming in Assembly Language" (found on Webster at http://webster.cs.ucr.edu). Builds are handled strictly by RadASM. Good for simple (one-HLA-file) HLA projects.
WPAAppBatch.tpl	Windows App compatible with code from "Windows Programming in Assembly"	RadASM will create a Win32 GUI project based on the structure of the code described in "Windows Programming in Assembly". These projects use the "wpa.hhf" header file and the "winmain.lib" library module described in Randy Hyde's book "Windows Programming in Assembly Language" (found on Webster at http://webster.cs.ucr.edu). Builds are handled by running one of several batch files (also created by this template) including build.bat, compilerc.bat, syntax.bat, and run.bat. By default, these batch files process a simple (one-HLA-source-file) project, but you can edit the batch files to handle more complex projects.
WPAAppMake.tpl	Windows App compatible with code from "Windows Programming in Assembly"	RadASM will create a Win32 GUI project based on the structure of the code described in "Windows Programming in Assembly". These projects use the "wpa.hhf" header file and the "winmain.lib" library module described in Randy Hyde's book "Windows Programming in Assembly Language" (found on Webster at http://webster.cs.ucr.edu). Builds are handled by running make.exe on a makefile that this template creates.
WPAAppNMAKE.tpl	Windows App compatible with code from "Windows Programming in Assembly"	Builds a project just like win32AppMake.tpl except that it invokes Microsoft's nmake.exe program rather than a generic make.exe program.
emptyWinApp.tpl	Windows App	RadASM will create an empty Win32 GUI project. Builds are handled strictly by RadASM. Good for simple (one-HLA-file) HLA projects.

Table 1:

Template Selection	Available if this project type is selected	Result
emptyWinAppBatch.tpl	Windows App	RadASM will create an empty Win32 GUI project. Builds are handled by running one of several batch files (also created by this template) including build.bat, compilerc.bat, syntax.bat, and run.bat. By default, these batch files process a simple (one-HLA-source-file) project, but you can edit the batch files to handle more complex projects.
emptyWinAppMake.tpl	Windows App	RadASM will create an empty Win32 GUI project. Builds are handled by running make.exe on a makefile that this template creates.
emptyWinAppNMake.tpl	Windows App	Builds a project just like emptyWinAppMake.tpl except that it invokes Microsoft's nmake.exe program rather than a generic make.exe program.

Generally, it's a good idea to select one of these templates when creating a new project. These templates automatically create any extraneous files a project needs (such as batch files and make files) and inserts these files into your new project. This spares you the effort of manually creating these files and inserting them into the project.

The RadASM/HLA package provides (at least) 16 different templates¹. There are four different template categories, each category containing four templates. Not all of these template files will be visible when you press the "template browse" button. The cons*.tpl files are only visible if you've selected the "Console App" radio button. The win32*.tpl, WPA*.tpl, and empty*.tpl files will only be visible if you've selected the "Windows App" radio button in the "Project Type" box.

Within a given template category (cons*, win32*, WPA*, empty*) there are four choices available to you. For example when selecting one of the console templates you could choose consApp.tpl, consAppBatch.tpl, consAppMake.tpl, or consNMake.tpl. The difference between these project types is how RadASM will build (compile/assemble) the project.

The *App.tpl template files tell RadASM to directly build your application (using commands found in the .tpl file). You can think of this as the "native" RadASM build mode. The only problem with this approach is that it is not very flexible (in terms of handling multi-file compilations) and it always rebuilds the entire project. As a result, projects that use the native RadASM build scheme are really suitable only for small (usually single-file) projects.

The *AppBatch.tpl template files tell RadASM to invoke various batch files when building the application. The template will actually create simple versions of these batch files for you: build.bat, compilerc.bat, syntax.bat, and run.bat. These batch files correspond to the items in the RadASM Make menu (note that the "Build" and "Build All" menu items both run the build.bat file). By default, these batch files only support a the creation of an application built around a single HLA source file (just like a native RadASM build). However, you can always edit these batch files to do a more sophisticated compilation. A later section will describe how to edit these batch files.

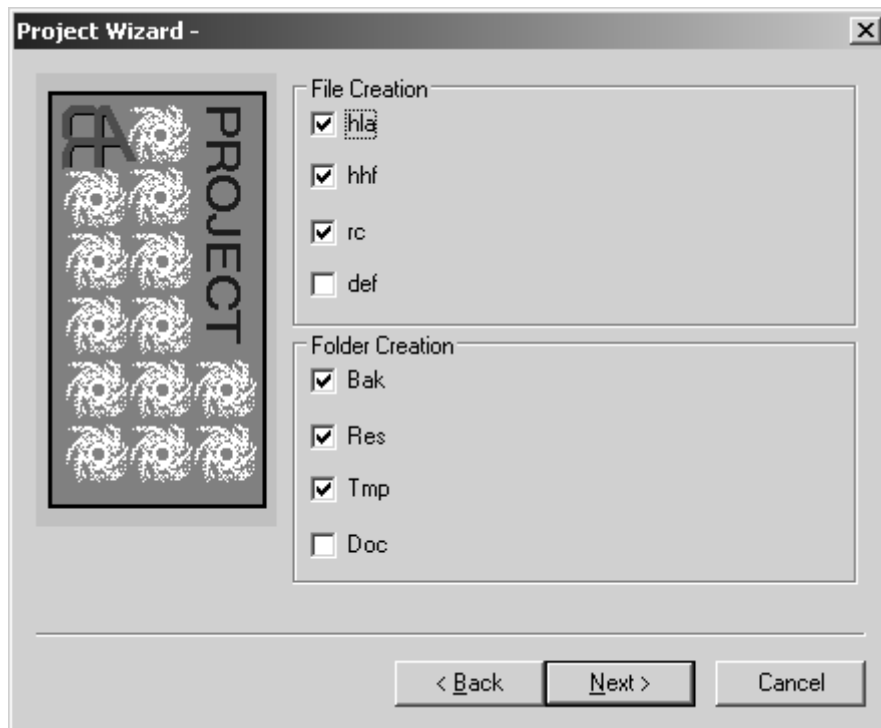
The *AppMake.tpl and *AppNMake.tpl template files tell RadASM to invoke a make utility (a generic make.exe program or Microsoft's nmake.exe utility, based on which template you select). The template creates a generic makefile for you (automatically) that handles all the menu items in the RadASM Make menu. Using make is, without question, the best way to use RadASM. Make is far more efficient for larger projects than using batch files or RadASM's built-in compilation capabilities. However, there are two drawbacks to using make: first, you have to have a copy of the make.exe (or nmake.exe) program (though this utility is available for free, see how to get a copy of

1. Actually, there may be more by the time you read this. The first 12 templates were operational when this manual was written. However, it's easy enough to add new templates to RadASM so there may be more by the time you read this.

this program in the section on make, earlier in this document); the second drawback is that you will have to edit the makefile that these templates create before you can build anything complex with them, i.e., if you want to create a sophisticated multi-file project, you'll need to make other changes to the makefile that the template creates. See the section on make earlier in this document for the details associated with the make language.

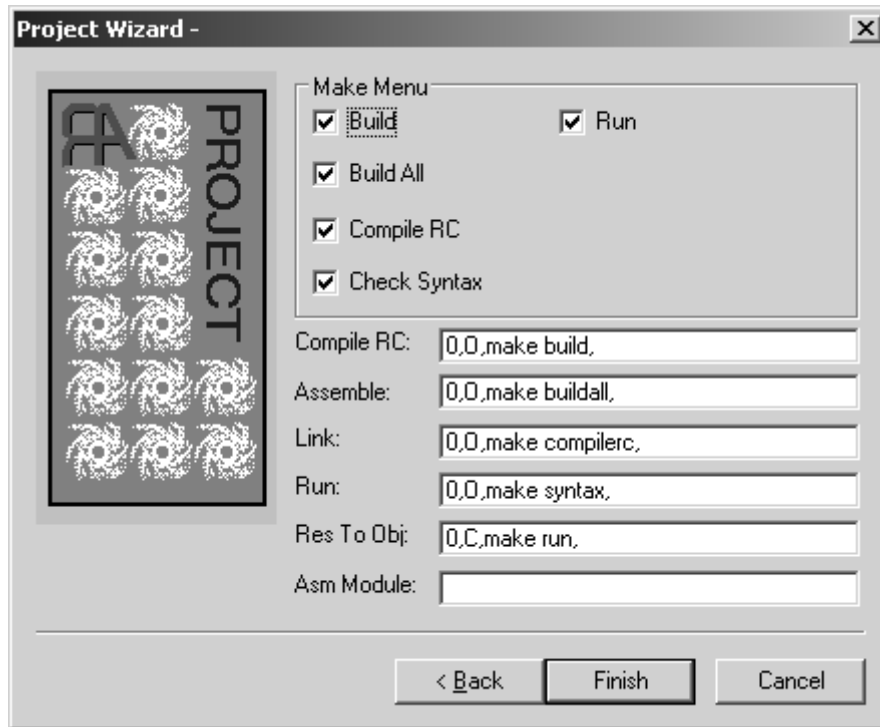
Once you've selected the assembler type, project type, entered the project name and description, and optionally selected the folder and a template, press the "Next>" button to move on to the next window of the Project Wizard dialog. This dialog box appears in Figure . In this dialog box you select the initial set of files and folders that RadASM will create in the project's folder for you. At the very least, you're going to want a ".hla" file and a "Tmp" subdirectory. It's probably a good idea to create a "BAK" subdirectory as well (RadASM will maintain backup files in that subdirectory, if it is present). More complex Windows applications will probably need a header file (".HHF") and if you're creating fancy GUI applications, you may need a resource file (".RC") as well. If you're creating a dynamically linked library (DLL), you'll probably want a definition file (".DEF") as well. If you plan on writing documentation, you might want to create a DOC subdirectory - the choice is yours. Check the files and folders you want to create and press the "Next >" button in the dialog box. Note that simple console applications (the type of applications most beginning HLA users create) require only a ".hla" file and a "Tmp" directory.

Project Wizard Dialog Box #2

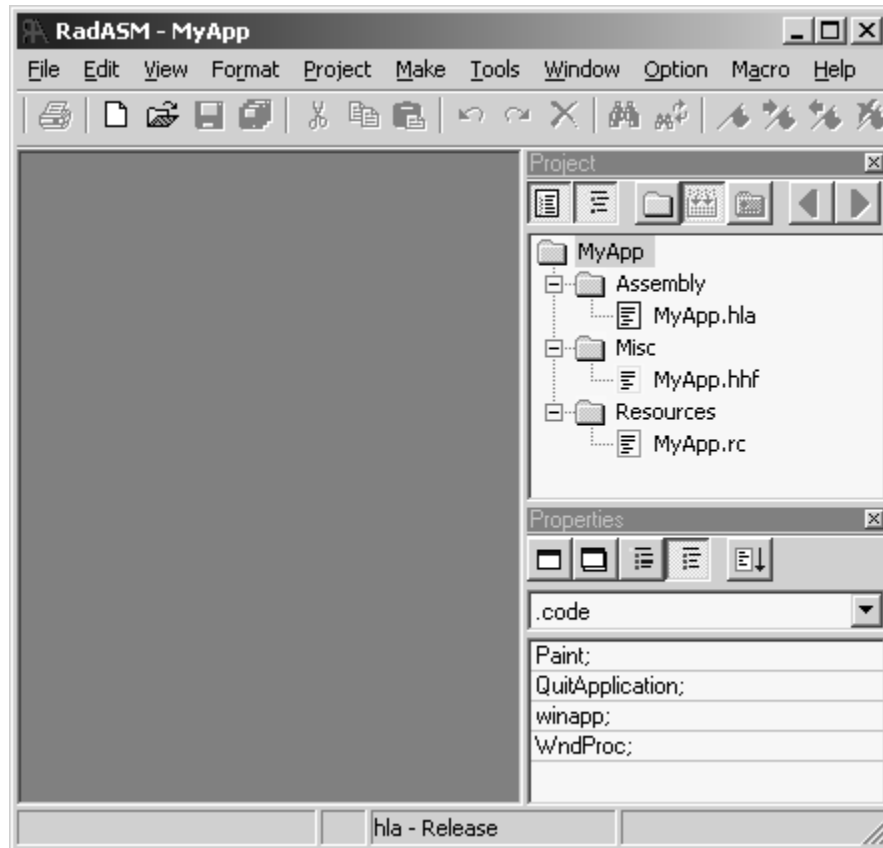


The last dialog box of the Project Wizard lets you specify the options present in the Make menu and the commands each of these options executes (see Figure). You should ignore all these options and just press the finish button. Generally, you will not customize this output; you will normally just hit the "finish" button to complete the construction of your project. If you do want to change these options, do it from the "Project>Project Options" menu item once you've created the project Figure shows what the RadASM window looks like after create a sample "Windows App" application based on the *win32app.tpl* template (this project was given the name "MyApp").

Project Wizard Dialog Box #3



Typical RadASM Window After Project Creation



4.2.9 Working With RadASM Projects

Of course, once you've created a RadASM project, you can open up that project and continue work on it at some later point. RadASM saves all the project information in a ".rap" (RadASM Project) file. This ".rap" file keeps track of all the files associated with the project, project-specific options, and so on. These project files are actually text files, you can load them into a text editor (e.g., RadASM's editor) if you want to see their format. As a general rule, however, you should not modify this file directly. Instead, let RadASM take care of this file's maintenance.

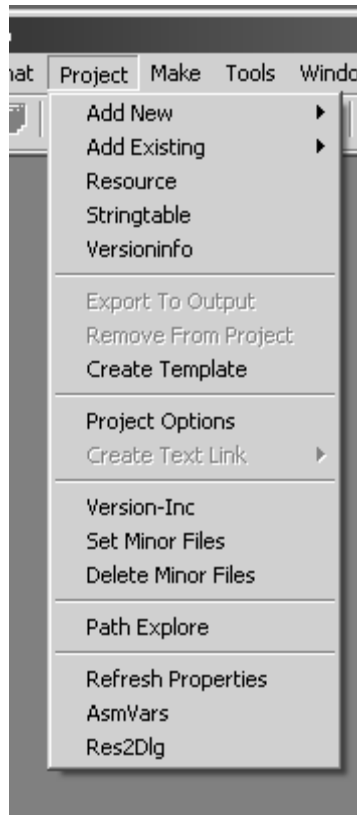
There are several ways to open an existing RadASM project file - you can double-click on the .rap file's icon within Windows and RadASM will begin running and automatically load that project. Another way to open a RadASM project is to select the "File>Open Project" menu item and open some ".rap" file via this open command. A third way to open a RadASM project is to use the File Browser to find a ".rap" file in one of your project directories and double-click on the project file's icon (the ".rap" file) that appears in the project browser. Any one of these schemes will open the project file you've specified.

RadASM only allows one open project at a time. If you have a currently open project and you open a second project, RadASM will first close the original project. You can also explicitly close a project, without concurrently opening another project, by selecting the "File>Close Project" menu item.

Once you've opened a RadASM project, RadASM's "Project" menu becomes a lot more interesting. When you create a project, RadASM gives you the option of adding certain "stock" files to the project (either empty files, or files with data if you select a template when creating the project). All of the files that RadASM creates bear the project's name (with differing suffixes). As a result, you can only create one ".hla" file (and likewise, only one ".hhf" file, only one ".rc" file, etc.). For smaller assembly projects, this is all you'll probably need. However, as you begin writing more complex applications, you'll probably want additional assembly source files (".hla" files),

additional header files (“.hhf”), and so on. RadASM’s Project menu is where you’ll handle these tasks (and many others). Figure shows the entries that are present in the Project menu.

The RadASM Project Menu



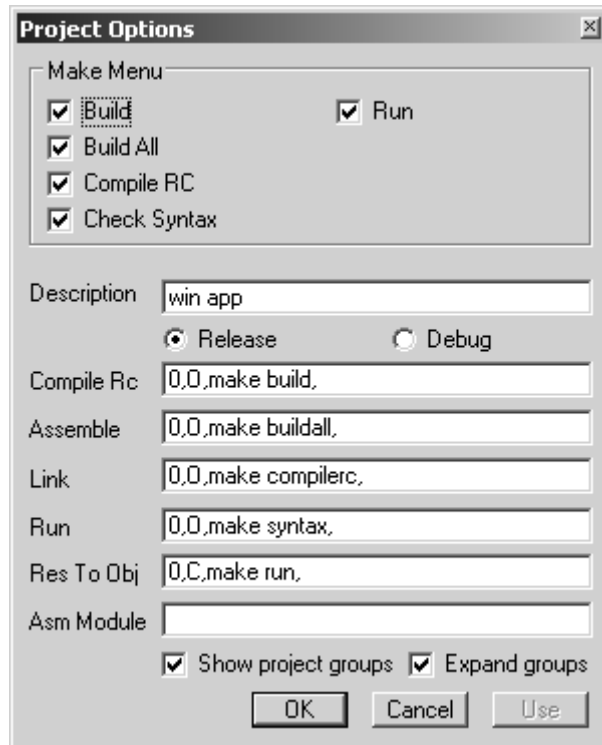
To add new, empty, files to a RadASM project, you use the “Project > Add New” menu item. This opens up a new submenu that lets you select an assembly file (“.hla” file), an include file (“.hhf”), a resource compiler file (“.rc”), a text file, and so on. Selecting one of these submenu items opens up an “Add New File” dialog box that lets you specify the filename for the file. Enter the filename and RadASM will create an empty text file with the name you’ve specified. Later on, you can edit this source file with RadASM and add whatever text is necessary to that file. Note that RadASM will automatically add that file to the appropriate group based on the file’s type (i.e., it’s suffix).

The “Project > Add Existing” sub-menu lets you add a pre-existing file to a project. This is a useful option for creating a RadASM project out of an existing HLA (non-RadASM) project or adding files from some other project (e.g., library routines) into the current project. Note that this option does not create a copy of the files you specify, it simply notes (in the “.rad” file) that the current project includes that file. To avoid problems, you should make a copy of the actual source file to the current project’s folder before adding it to the project; then add the version you’ve just copied to your project. It’s generally unwise to add the same source file to several different projects. If you change that source file in one project, the changes are automatically reflected in every other project that links this file in. Sometimes this is desirable, but most of the time programmers expect changes to a source file to be localized to the current project. That’s why it’s always best to make a copy of a source file when adding that file to a new project. In those cases where you do want the changes reflect to every application that includes the file, it’s better to build a library module project and link the resulting “.lib” file with your project rather than recompile the source file in.

The “Project > Project options” menu item opens up a “Project Options” dialog box that lets you modify certain project options (see Figure). This dialog box lets you change certain options that were set up when you first created the project using the “File > New Project” Project Wizard

dialogs. Most of the items in this dialog box should have been described earlier, but a few of the items do need a short explanation.

“Project > Project Options” Dialog Box



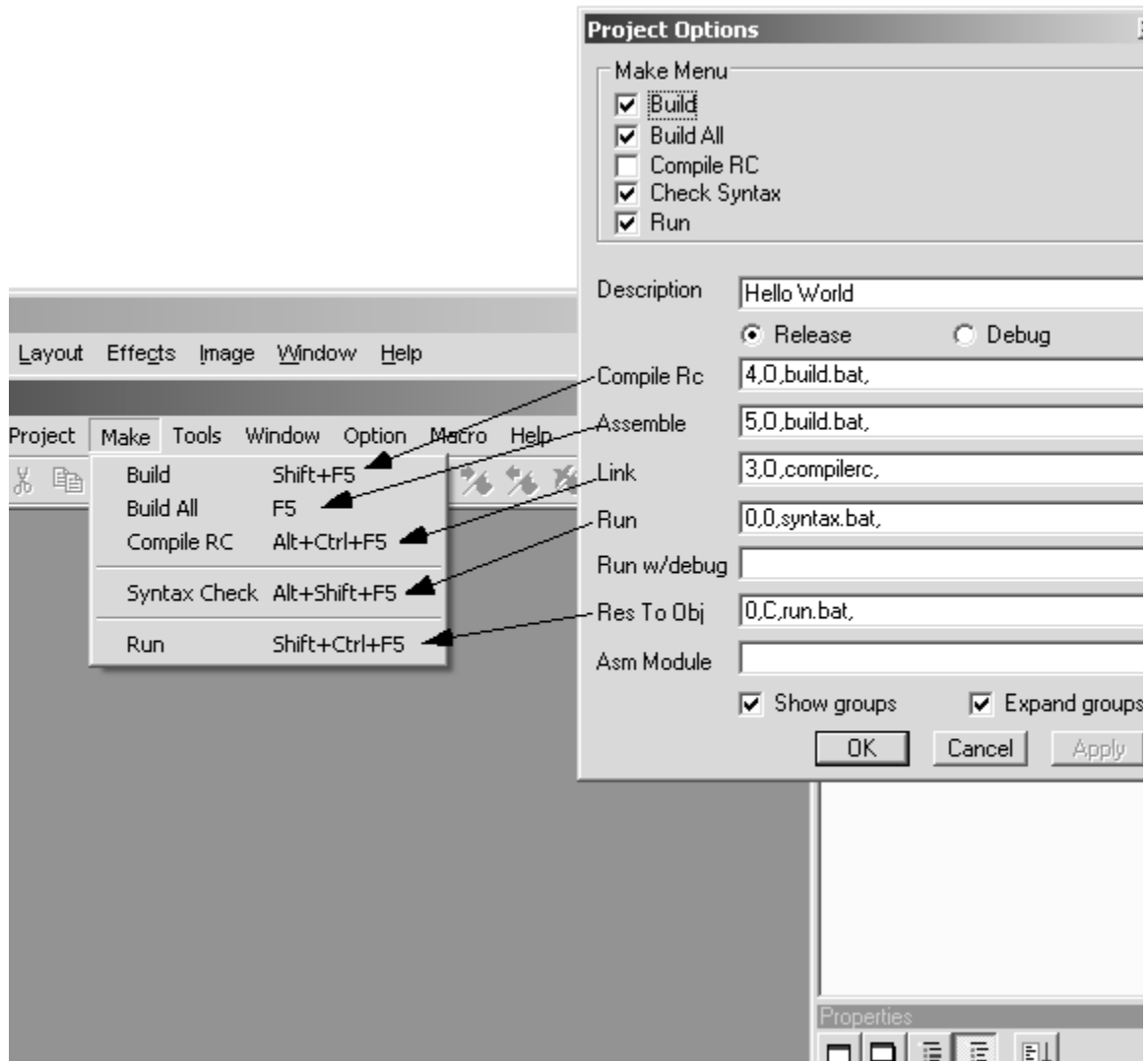
The Project Options dialog box provides two radio buttons that let you select whether RadASM will do a “debug build” or a “release build.” Be sure that the “Release” radio button is selected. The Debug option instructs HLA to insert certain debugging information into your executable file (e.g., for use by OllyDbg). We will not consider that option in this document.

4.2.10 Build Options with RadASM/HLA

Before discussing how to actually edit and compile programs using RadASM, we need to stop for a moment and discuss the internal operation of RadASM and how it controls programs like HLA. RadASM was created to be a very flexible system supporting multiple assemblers and different ways of building applications. In one respect, this flexibility is very good - it is exactly this flexibility that allows RadASM to work with HLA (rather than just with Microsoft’s assembler, for which RadASM was initially created). On the other hand, there is a down side to this flexibility - creating HLA projects is a little bit more involved than it has to be had RadASM been written specifically for HLA. In this section we’ll discuss the extra work involved with creating and maintaining RadASM projects.

Take another look at Figure . Beside the labels “Compile RC”, “Assemble”, “Link”, etc., you’ll find some editable strings. These strings are special RadASM commands that tell RadASM what to do whenever you select an item from RadASM’s “Make” menu. Originally, the labels next to each of these text edit boxes corresponded to menu items in the RadASM “Make” menu; HLA, however, has renamed the menu items in the “Make” menu, so they no longer correspond to the labels appearing in the “Project Options” dialog box (Figure). Figure shows the relationship between the labels in the Project Options dialog box and the Make menu.

Correspondence Between Project Options and Make Menu



The text appearing in the corresponding text edit box in the Project Options dialog box is a command that RadASM executes whenever you select the corresponding item from the Make menu. Here's the syntax for each of these entries:

```
DEL, OUT, CMD, FILE {, FILE, ...}
```

DEL is a numeric entry that specifies which files to delete prior to executing the command. Normally, this should be zero (which means "don't delete any files.").

OUT is either "O", "OT", or "C" meaning that the command's output goes to the RadASM output windows ("OT"), the command produces no output (and any output is ignored, "O"), or RadASM opens up a console (command-line) window and sends all output to that window ("C"). For most commands except "RUN", you'll probably want the command's output to go to the RadASM output window; when running the program you'll probably want the output to go to a console window (at least, if you're writing a console application).

CMD is the command (command-line prompt command) to execute in response to this Make menu selection. This includes the program's name and any command line parameters (though you don't usually specify the filenames to process here).

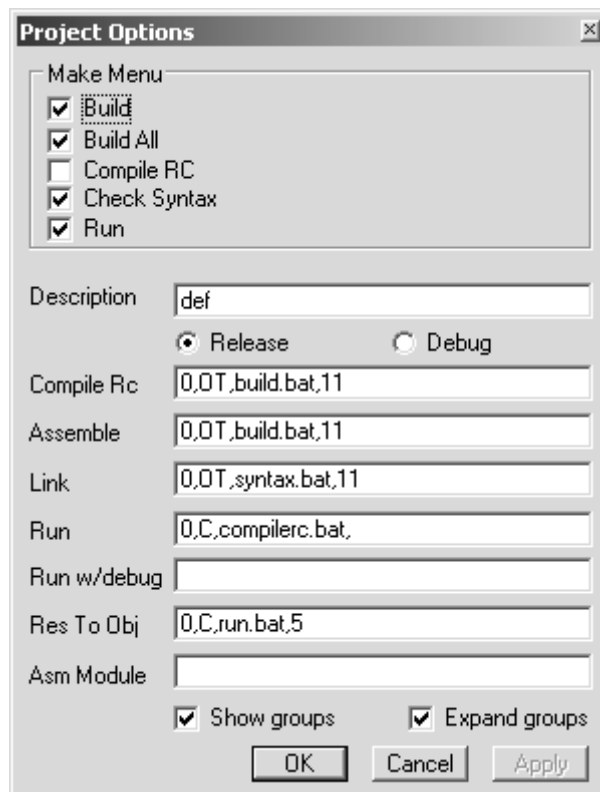
FILE is a special numeric designation (internal to RadASM) that specifies the file that the CMD is to process. We'll normally leave this blank, see the discussion on RadASM customization later in this document for more details on this entry.

There are three common ways people use to have RadASM run HLA to compile an HLA project: direct command execution, batch file execution, and make file execution. Each of these execution modes have their own set of advantages and disadvantages.

Direction command execution is the default mode for RadASM/HLA "out of the box." This mode has the advantage of being the easiest to use. For the most part, it doesn't require the creation of any special files in order to build a given project (though the "run" command works best if you create a batch file for it). There are several disadvantages to this approach. First, it doesn't work with HLA on all versions of Windows. Another disadvantage to this approach is that it's mainly useful for single-file projects (unless you're willing to delve deep into RadASM and learn all about customizing it for your own purposes). Yet another disadvantage is that you have to manually build each component of the project when using the direct command execution. In general, the disadvantages would outweigh the advantages of this execution mode were it not for the fact that the direct command approach works best for simple projects as it doesn't require the creation of any batch or makefiles. However, once you've created a few HLA projects and get comfortable with RadASM, you'll probably want to shift to one of the other RadASM operation modes. Note that running in this mode is equivalent to creating a project with one of the *App.tpl templates (also note that template settings always override the default settings).

Batch file execution is the second mode of operation that RadASM/HLA supports. In this mode of operation each of the RadASM commands in the Project Options dialog box executes a batch file and that batch file handles whatever set of tasks is necessary for the specified Make menu option. Figure shows the Project Options dialog box with the commands to execute when operating in batch mode. Note that each of the commands simply execute a batch file (build.bat, compilerc.bat, syntax.bat, and run.bat).

RadASM in Batch Execution Mode (Project Options)



The batch files specified in the Project Options dialog box must appear in the same directory as the other files for project (e.g., along with the source files). These batch files contain a list of command-prompt commands to execute whenever you select one of the menu items from the Make menu. Here are the contents for each of the generic batch files supplied with RadASM/HLA:

```
build.bat:

hla -p:tmp %1

compilerc.bat:

echo "No Resource Files to Compile!"
pause

syntax.bat:

hla -p:tmp -s %1

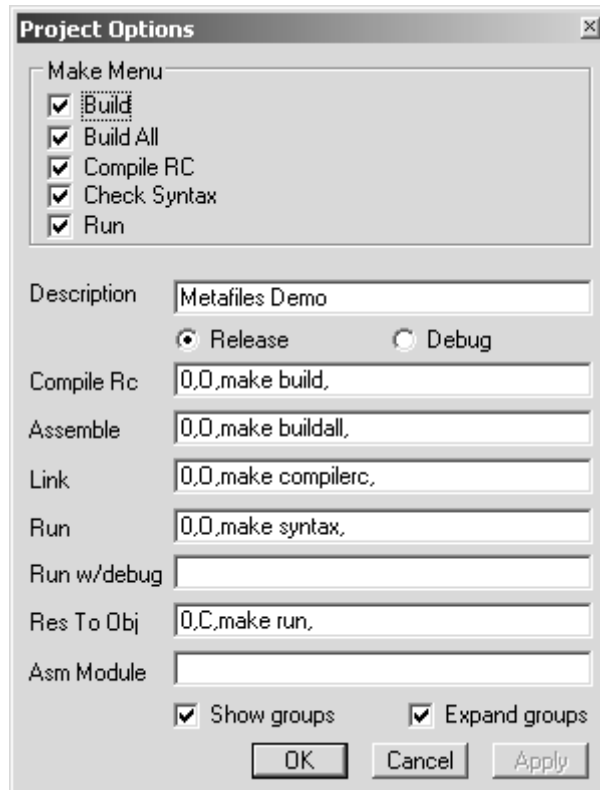
run.bat:

%1
pause
```

The advantage of the batch file execution scheme over the direct execution scheme is that you can execute several commands within a batch file (unlike the direct command execution scheme). This lets you compile multi-file projects and execute other command-line actions within the batch file. Also, the batch file scheme works with all versions of Windows. Furthermore, the batch file scheme doesn't require any additional utility programs to achieve this flexibility. Batch files have two main disadvantages. First, you have to write a set of batch files for every project you create (though for single-file projects, the generic batch files work fine; editing the batch files is only necessary for more sophisticated projects). The second problem with batch files is they force a rebuild of every file in a multi-file project, even if such work is unnecessary.

The makefile execution scheme is the most flexible of the three. Like the batch file scheme, you can execute multiple commands and this scheme works with all versions of Windows. A big advantage of makefiles over batch files is that you can easily handle large multi-file projects using makefiles and you can build the projects only recompiling the files that are necessary. Like batch files, one disadvantage to using makefiles is that you have to maintain a separate "makefile" that directs the compilation. Another disadvantage to the makefile scheme is that you have to have a separate "make" utility installed on your system (if you don't already have a copy of make, you can obtain one for free from Borland; see the section on "Make" for more details). Figure show the command set for RadASM when using the makefile execution scheme with Borland's "make.exe" program (note: to use Microsoft's "nmake.exe" program, simply change each occurrence of "make" to "nmake" in the dialog box).

Makefile Execution Scheme (Project Options)



The version of RadASM that ships with HLA includes several versions of the “hla.ini” initialization file that RadASM uses. These files are the following:

- hla.ini - this is the actual file that RadASM uses. As shipped, this is the same as hla_2000.ini (the direct execution mode file).
- hla_2000.ini - this is the version of the hla.ini file that supports direct command execution. If you ever change hla.ini and you want to restore the direct execution form, simply make a copy of this file and rename it to hla.ini.
- hla_bat.ini - this is the version of the hla.ini file that supports batch mode execution. If you want to use batch mode execution, make a copy of this file and rename the copy to “hla.ini”.
- hla_make.ini - this is the version of the hla.ini file that supports Borland’s make.exe application for the makefile execution mode (actually, this .ini file supports makefile execution using any make program named “make.exe”). If you want to use makefile mode execution, make a copy of this file and rename the copy to “hla.ini”.
- hla_nmake.ini - this is the version of the hla.ini file that supports Microsoft’s nmake.exe application for the makefile execution mode. If you want to use makefile mode execution, make a copy of this file and rename the copy to “hla.ini”.

Note that the execution mode specified by the “hla.ini” files is only available when you create a new project without using a template (template files override the settings in the hla.ini file). Each RadASM project file you create (the “.rap” file) maintains the execution mode as part of that project. Should you change the execution mode by copying some new file over the top of hla.ini, you do not change the execution modes for any pre-existing projects. If you want to change the execution mode of an existing project, you will have to select the “Project>Project Options” menu item and edit the entries in the project option dialog box.

The remainder of this document will assume that you’re using the flexible “makefile” execution mode and that you’re creating makefiles for each of your projects. Therefore, to follow along with the examples that appear in the remainder of this document, you should make a copy of

the `hla_make.ini` (or `hla_nmake.ini`) file and rename it to `hla.ini`. Another alternative is to always use one of the `*AppMake.tpl` or `*AppNMake.tpl` templates when creating new projects.

4.2.11 Editing HLA Source Files Within RadASM

The RadASM text editor is quite similar to most Windows based text editors you've used in the past (i.e., RadASM generally adheres to the Microsoft Common User Access (CUA) conventions. So the cursor keys, the mouse, and various control-key combinations (e.g., `ctrl-Z`, `ctrl-X`, and `ctrl-C`) behave exactly as you would expect in a Windows application. Because this is an advanced programming book, this chapter will assume that you've used a CUA-compliant editor (e.g., Visual Studio) in the past and we'll not waste time discussing mundane things like how to select text, cutting and pasting, and other stuff like that. Instead, this section will concentrate on the novel features you'll find in the RadASM editor.

Of course, the first file navigation aid to consider is the Project Browser pane. We've already discussed this RadASM feature in earlier sections of this document, but it's worth repeating that the Project Browser pane lets you quickly switch between the files you're editing in a RadASM project. Just double-click on the icon of the file you want to edit and that file will appear in the RadASM editor window pane.

Immediately below the Project Browser pane is the "Properties" pane (if this pane is not present, you can bring it up by selecting "View > Properties" from the RadASM View menu). This pane contains two main components: a pull down menu item that lets you select the information that RadASM displays in the lower half of this window. If not already selected, you should select the ".code" item from this list. The ".code" item tells RadASM to list all the sections of code that it recognizes as procedures (or the main program) in an HLA source file (see Figure).

The HLA Properties Window Pane



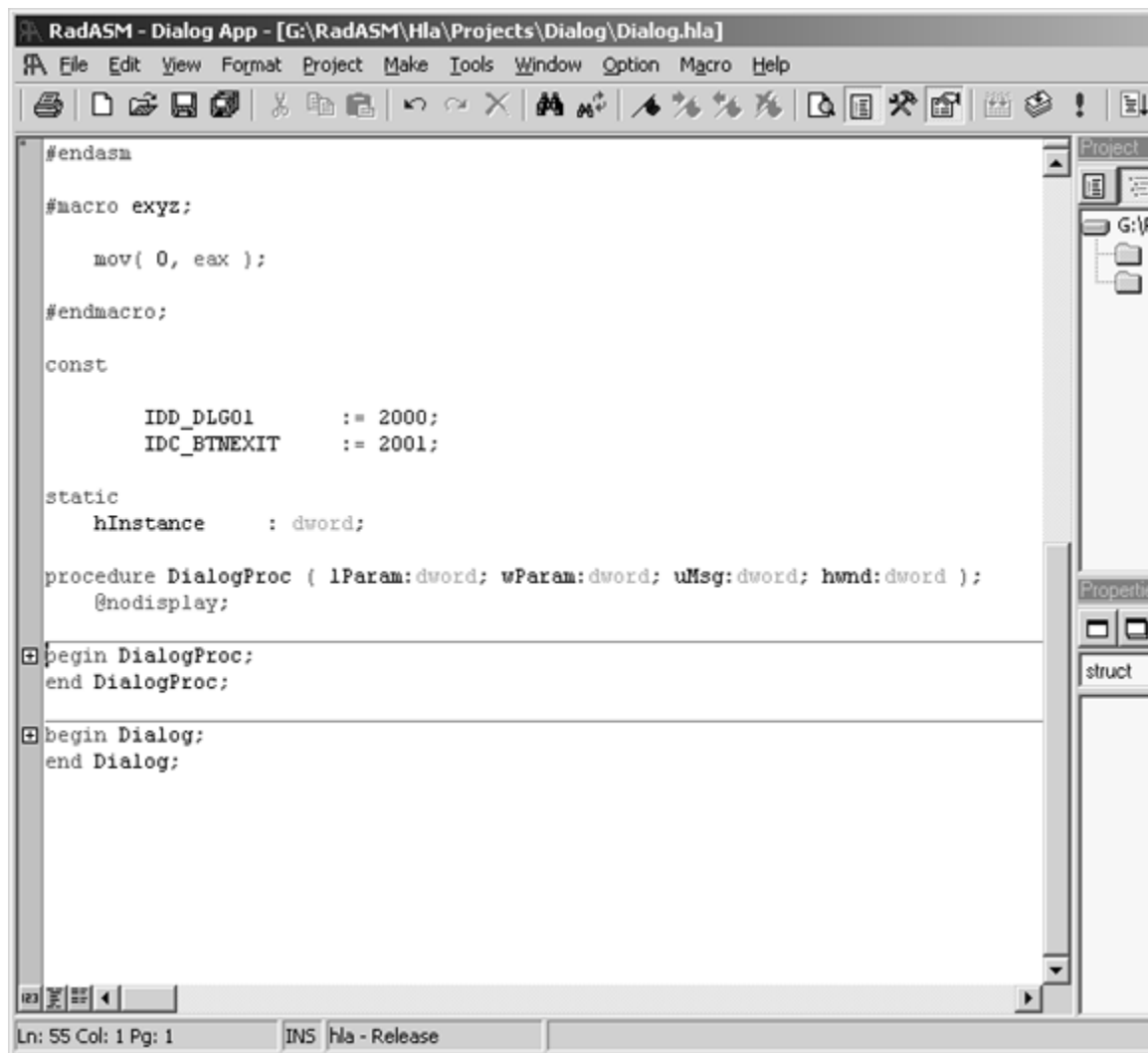
One very useful RadASM feature is that you can quickly jump to the start of a procedure's body (at the `begin` statement) by simply double-clicking on that procedure's name in the Properties Window pane. In the example appearing in Figure (this is the "Dialog" project supplied with RadASM for HLA), double-clicking on the "Dialog;" and "DialogProc;" lines in this list box automatically navigates to the start of the code for the selected procedure.

The pull-down menu in the Properties window lets you select the type of objects the assembler provides. For example, by selecting ".const" you can take a look at constant declarations in HLA. The "macro" selection lets you view the macro definitions that appear in the source file. As this chapter was first being written, the other property items weren't 100% functional; hopefully by the time you read this RadASM will have additional support for other types of HLA declarations.

Another neat feature that RadASM provides is an "outline" view of the source file. Looking back at Figure you'll notice that "`begin DialogProc;`" statement has a rectangle with a

minus sign in it just to the left of the source code line. Clicking on this box closes up all the code between the `begin` and the corresponding `end` in the source file. Figure shows what the source file looks like when the `Dialog` and `DialogProc` procedures are collapsed in outline mode. The neat thing about outline mode is that it lets you view the “big picture” without out the mind-numbing details of the source code for each procedure in the program. In outline view, you can quickly skim through the source file looking for important code and “drill down” to a greater level of detail by opening up the code for a procedure you’re interested in looking at. You can also rapidly collapse or expand all procedure levels by pressing the “expand” or “collapse” buttons appearing on the lower left hand corner of the text editor window (see Figure).

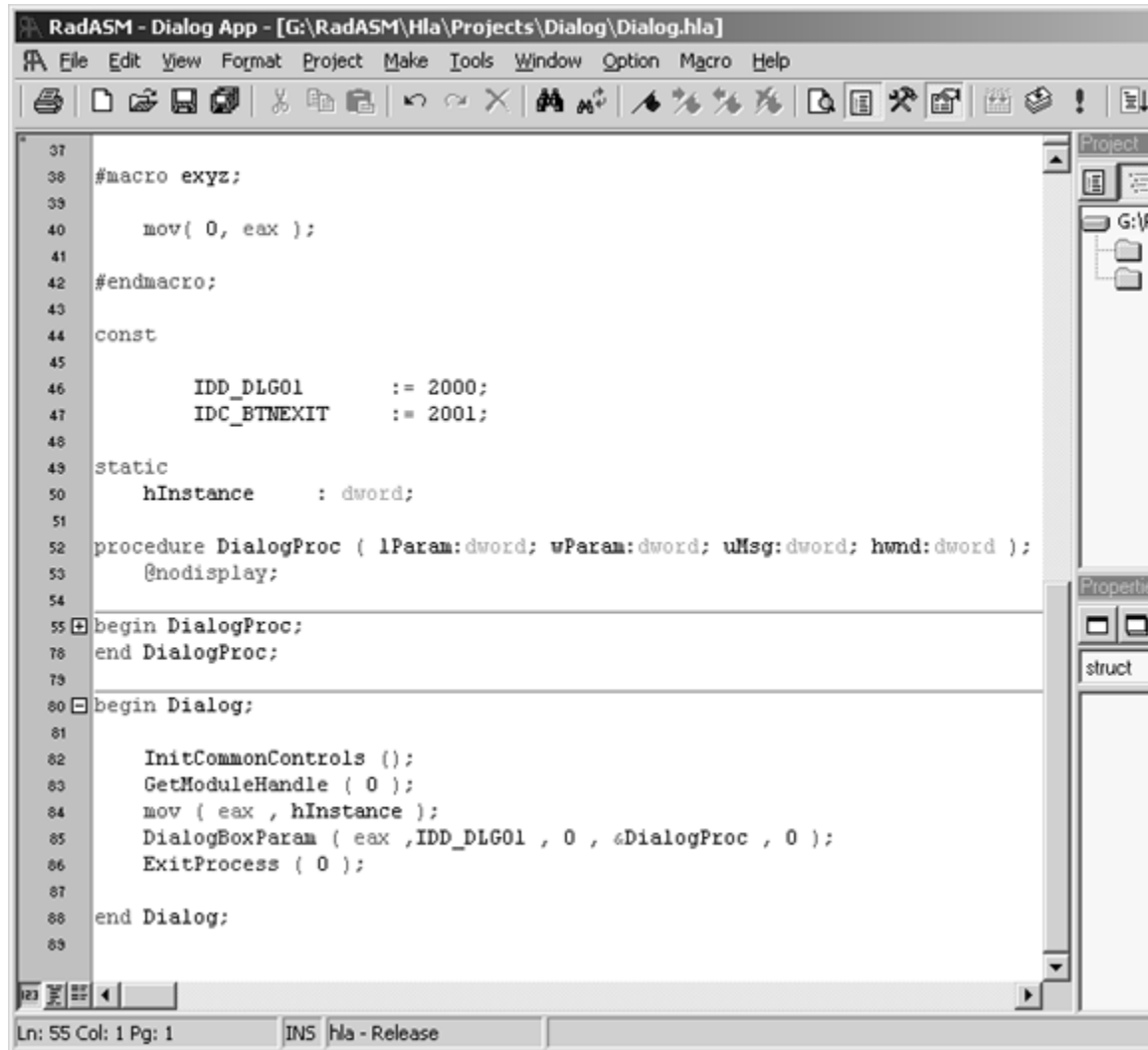
RadASM Outline View (with Collapsed Procedures)



Another useful feature RadASM provides is the ability to display line numbers with each line of source code. Pressing on the line number icon in the lower-left hand corner of the text editor window (the icon with the “123” in it) toggles the display of line numbers in the editor’s window. See Figure to see what the source file looks like with line numbers displayed. The line number display mode is quite useful when searching for a line containing a syntax error (as reported by

HLA). Note that you can also navigate to a given line number by pressing ctrl-G and entering the line number (you can also select “Edit > Goto line” from the “Edit” menu).

Displaying Line Numbers in RadASM's Editor



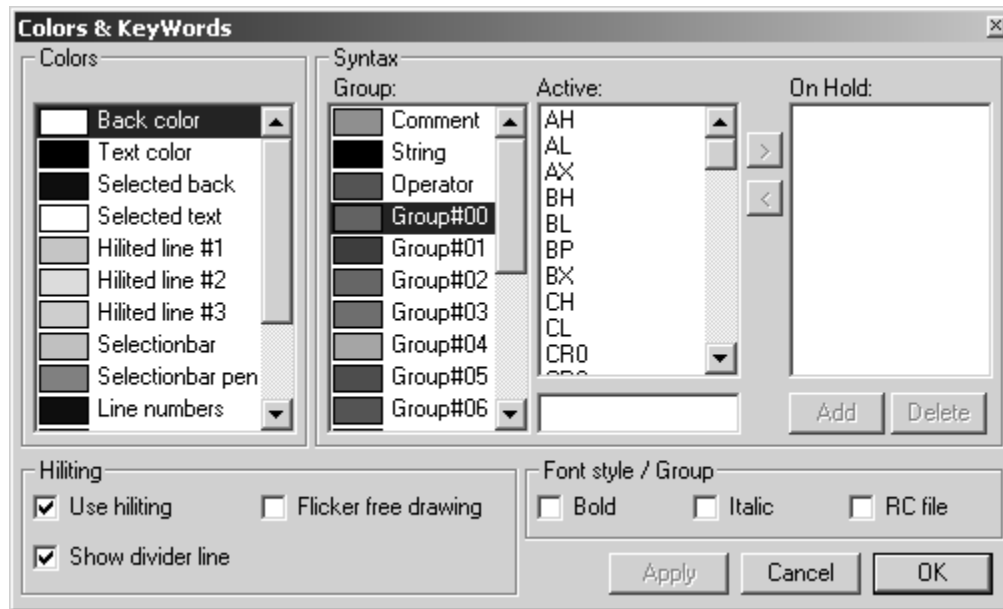
Another useful navigation feature in RadASM is support for *bookmarks*. A bookmark is just a point in the source file that you can mark. You can create a bookmark by selecting a line of text (by clicking the mouse on the gray bar next to the line) and selecting “Edit > Toggle BookMark” or by pressing shift-F8. You can navigate between the bookmarks by pressing F8 or ctrl-F8 (these move to the next or previous bookmarks in the source file). RadASM (by default) provides several icons on its toolbar to toggle bookmarks, navigate to the previous or next bookmark, or clear all the bookmarks. Which method (edit menu, function keys, or toolbar) is most convenient simply depends on where your hands and the mouse cursor currently sits.

The RadASM “Format” menu also provides some useful features for editing HLA programs. The “Format > Indent” and “Format > Outdent” items (also accessible by pressing F9 and ctrl-F9) move a selected block of text in or out four spaces (so you can indent text between an `if` and `endif`, for example). You can also convert tabs in a document to spaces (or vice versa) from the “Format > Convert > Spaces To Tab” and “Format > Convert > Tab To Spaces” menu selections.

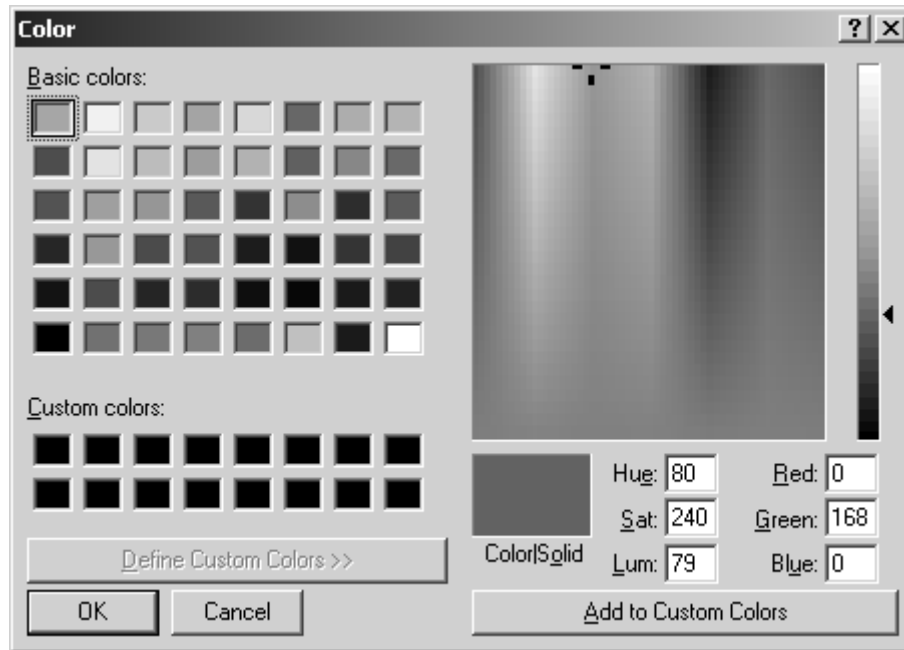
You’ll notice that RadASM provides syntax coloring in the editor window (that is, it sets the text color for various classes of reserved words and symbols to different colors, making them easy

to identify with a quick glance in the editor window). The *hla.ini* file accompanying the RadASM/HLA release contains a set of reasonable color definitions for HLA's different reserved word types. However, if you don't particularly agree with this color scheme, it's really easy to change the colors that RadASM uses for syntax highlighting. Just select the "Options > Colors & Keywords" menu item and select an item from the Syntax/Group list box (Figure shows what this dialog box looks like with the Group #00 item selected). By double-clicking on an item within the Group list box, you can change the color for all the items in that particular group (e.g., see Figure). RadASM automatically updates the *hla.ini* file to remember your choice of colors the next time you run RadASM.

Option>Colors & Keywords Dialog Box with Group#00 Selected



Color Selection Dialog Box



You can also set the display fonts to something you're happier with if the default font (Courier New, typically) isn't to your liking. This is also achievable from the RadASM "Option" menu.

4.2.12 Managing Complex Projects with RadASM

One of the main reasons for using a project-oriented integrated development environment like RadASM is to streamline the development of complex projects. For the sake of argument, we'll define a "simple" project as any HLA project consisting of a single ".hla" source file and, possibly, a header file. A complex project will be any application that requires multiple source files, object modules, library modules, and resource files that must be separately compiled and linked together to form a single executable file. Though an IDE such as RadASM is helpful when working on simple projects, a development environment is most effective when working on larger, complex projects.

Although it is possible to maintain certain complex projects using RadASM's native capabilities, by far the best solution for complex projects is to use a make utility or (if you don't have access to a make utility) batch files to control the compilation process. Though it requires a little additional labor to set up a set of batch files or a make file, the flexibility you gain by using this approach is well worth the small amount of additional effort (effort that will be repaid many times over during the project's development).

Before describing how to write batch files and makefiles to take over control of the compilation process from RadASM, perhaps it would be wise to offer a small justification for this approach. After all, RadASM has some very sophisticated schemes for building projects, why not stick with RadASM's native approach? Well, there are several reasons. First, although RadASM's general nature is a wonderful attribute of the system (e.g., it allows HLA to work with RadASM even though it was originally designed for MASM), sometimes a *specific* solution is more efficient or more powerful than a general solution. Second, although RadASM is a great development environment, sometimes it's just easier or more convenient to compile a project from the command line prompt; by using batch or make files in your RadASM projects, you can easily work from the command line *or* from within RadASM and know that you're building your project exactly the same way in both cases. Also, tools like the make utility have been around for quite some time and contain lots of features that you won't find in a less mature system like RadASM. Fortunately, RadASM is flexible enough to allow the use of batch and make files when working on a project, so you get the best of both worlds - the convenience of an integrated development environment, and the power and flexibility of make files.

4.2.13 Project Maintenance with Batch Files

The batch file approach is usable by those who do not have access to a make utility (or those who want to distribute RadASM projects to others who might not have a make utility available). Although batch files are more flexible than native RadASM builds, you should really attempt to use the make file approach unless there are some extenuating reasons why you would rather go with the batch file approach (e.g., the need to distribute RadASM/HLA projects to people who might not have a make utility).

A batch file is simple an ASCII text file that contains a sequence of command-line commands. The Windows command-line interpreter executes each line of text in a batch file just as though you'd typed those commands directly into a command window. By placing multiple commands in a batch file, you can execute as many commands as necessary to build your project. For example, suppose you have a little utility that increments a version number embedded in an HLA header file. You could execute a batch file that bumps up the version number and builds the HLA application using the following sequence of commands:

```
BumpVersion version.hhf
hla FileThatIncludesVersionFile.hla
```

Batch files also let you specify command-line parameters, e.g.,

```
someCmd parm1 parm2 parm3 ...
```

You may refer to these command-line parameters within the batch file using %1, %2, %3, etc. For example, the default build.bat file that RadASM will create for you if you specify the use of one of the *AppBatch.tpl template files is

```
hla -p:tmp %1
```

RadASM (by default) invokes this build.bat file using a command line like the following:

```
build filename.hla
```

The batch file processor substitutes "filename.hla" for the "%1" within the batch file.

The big problem with RadASM's native compilation facilities is that it doesn't particularly know what files you want to compile. It will supply the main project name (or, with appropriate customization, all the files in a given project), but it won't let you easily pick and choose which files you want to process. That's where batch files (and makefiles) are useful. In a project-specific batch file, you can easily specify any or all files that you want to compile and link together into a single executable. For example, if you have a project that combines two HLA files, a resource (.rc) file, and a specialized library, you could handle this compilation with the following command in a batch file:

```
hla myProj.hla subroutines.hla resources.rc speciallib.lib
```

Such a command line could not be built (automatically) from within RadASM. This is particularly true if some of the files are not present in the project's directory (e.g., common object and library files present in a separate subdirectory).

If you create a project with one of the *AppBatch.tpl templates, or create a generic project using the hla_batch.ini file as your hla.ini file, then RadASM will, by default, connect the following Make menu items to the following batch files.

RadASM/HLA Make Menu/Batch File Correspondence

Table 2:

Make Menu Item	Corresponding Batch File
Build	build.bat
Build All	build.bat
Compile Resource	compilerc.bat
Syntax Check	syntax.bat
Run	run.bat

Note that the Build and Build All menu items both invoke the same batch file. The Build menu item's intent is to build the application by compiling only those files that absolutely need to be compiled. This feature is generally available only if you're using make files. Therefore, if you choose the Build menu item when using batch files, it will generally recompile all files in the application. The Compile Resource and Syntax Check menu items in the Make menu will invoke their corresponding batch files that will contain commands to compile a resource file (if any) or run the HLA compiler in a "compile to assembly" mode (no object or executable output, i.e., a syntax check of the file).

The run.bat file is somewhat special. The default run.bat file takes the following form:

```
%1
pause
```

RadASM will pass a command line parameter of the form "*projectname.exe*" to the run.bat file. Assuming that your project has compiled the files to produce the executable "*projectname.exe*", this batch file will execute your application and then wait until you hit the enter key before it closes up the console window that executes the batch file (this gives you the opportunity to review any output produced by console applications).

If you would prefer to execute different batch commands when selecting items from RadASM's Make menu, you can specify the commands to execute by selecting the "Project>Project Options" menu item. This opens up a dialog box that lets you specify the command line parameters for each of the menu items. See the discussion elsewhere in this document for more details.

In general, batch files are not the most appropriate way to deal with complex projects. Make files are a much better solution. Therefore, unless you absolutely have to, you should avoid using the RadASM/HLA batch file compilation scheme.

4.2.14 Project Maintenance with Make Files

Makefiles provide the best way to build complex projects when using RadASM/HLA. They are more efficient, they are safer, and they give you more control over the compilation process than you will get with RadASM's native mode or when using batch files. For most projects, the make file build scheme is, by far, the best. There are, of course, a couple of disadvantages to using make files. Specifically, you need to have a make utility in order to use makefiles and you need to learn the "make language" in order to use make files. Fortunately, a decent version of make is available for free from Borland and learning make is not that difficult (see the discussion of make earlier in this document). However, the advantages of make files far outweigh the disadvantages, so you should give make files serious consideration if you're not sure which RadASM/HLA compilation scheme to use.

Here are the commands that RadASM executes whenever you select an item from RadASM's make menu when using make files to build your application:

```
Build menu item:           make build
```

```
Build All menu item:      make buildall
Syntax Check menu item:  make syntax
Compile Resource menu item: make compilerc
Run menu item:           make run
```

These commands all assume that there is a single file, “makefile” present in the project directory. These commands will execute the *build*, *buildall*, *syntax*, *compilerc*, or *run* dependencies in the makefile, respectively. Here’s what the default makefile (supplied with RadASM/HLA) looks like:

```
build: $(hlafile).exe

buildall: clean $(hlafile).exe

compilerc:
    echo No Resource Files to Process!

syntax:
    hla -s $(hlafile).hla

run: $(hlafile).exe
    $(hlafile)
    pause

clean:
    delete tmp
    delete *.exe
    delete *.obj
    delete *.link
    delete *.inc
    delete *.asm
    delete *.map

$(hlafile).exe: $(hlafile).hla
    hla $(DEBUG) -p:tmp $(hlafile)
```

For simple projects (i.e., projects consisting of a single HLA source file), you’ll be able to use the makefile as-is. RadASM automatically supplies the project’s name as the “hlafile” variable to build your project whenever you select an item from the RadASM “Make” menu. For more complex projects, you’re going to want to edit this makefile extensively to add additional dependencies and commands.

The *build* dependency in this make file executes whenever someone selects the “Make>Build” menu item in RadASM. The intent of this command is to build the application with as little processing as possible. That is, if several of the files needed to build the final executable have already been compiled into object files, this command should not recompile those files, it should use the up-to-date objects as-is and only recompile those files whose source files are newer than the object files.

The *buildall* dependency in the makefile executes whenever someone selects the “Make>Build All” menu item in RadASM. The intent of this command is to do a complete build of the system, ignoring any object files that are already up to date. The typical execution of this command involves deleting all temporary files (e.g., object files) by executing the “clean” operation, and then doing a build.

The *compilerc* dependency executes whenever you select the RadASM “Make>Compile Resource” menu item. In the default make file provided with RadASM/HLA, this command simply displays a brief diagnostic message. If your project has some resource files that you need to compile with Microsoft’s resource compiler, then you would normally specify the dependencies

and commands needed to process those resource files after the compiler dependency. For example, if your project includes a resource file named “myProject.rc”, you’d typically edit the makefile to add/modify the following:

```
build: $(hlafile).exe $(hlafile).res

buildall: clean $(hlafile).exe $(hlafile).res

compiler: $(hlafile).res

syntax: compiler
    hla -s $(hlafile).hla

$(hlafile).res: $(hlafile).rc
    rc -v $(hlafile).rc
```

Of course, once you start making major modifications to the makefile, you can probably drop the use of the \$(hlafile) variable and use the direct filenames (variables are great for generic makefiles; however, they tend to obscure makefiles created for a specific project). That is, for a project like “myProject” with a “myProject.hla” file and a “myProject.rc” file you’d probably just create a makefile like the following:

```
build: myProject.exe

buildall: clean myProject.exe

compiler: myProject.res

syntax:
    hla -s myProject.hla

run: myProject.exe
    myProject
    pause

clean:
    delete tmp
    delete *.exe
    delete *.obj
    delete *.link
    delete *.inc
    delete *.asm
    delete *.map
    delete *.res

myProject.exe: myProject.hla myProject.res
    hla $(DEBUG) -p:tmp myProject myProject.res

myProject.res: myProject.rc
    rc -v myProject.rc
```

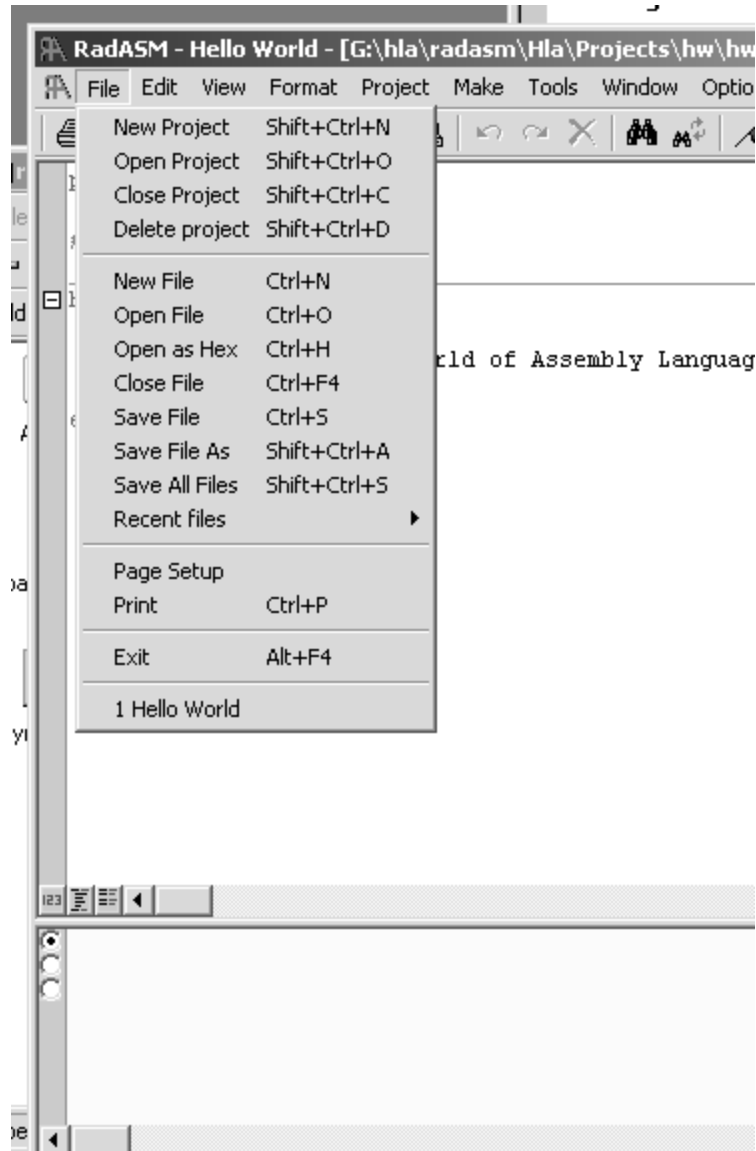
4.2.15 RadASM Menus

The following sections describe many of the RadASM menu items and their applicability to HLA software development. For a full discussion of each menu item, please see the on-line RadASM help file.

4.2.15.1 The RadASM File Menu

The RadASM file menu provides all the common file operations you'd expect in a Windows application, plus a few RadASM specific entries (see Figure).

RadASM File Menu



4.2.15.1.1 File>New Project

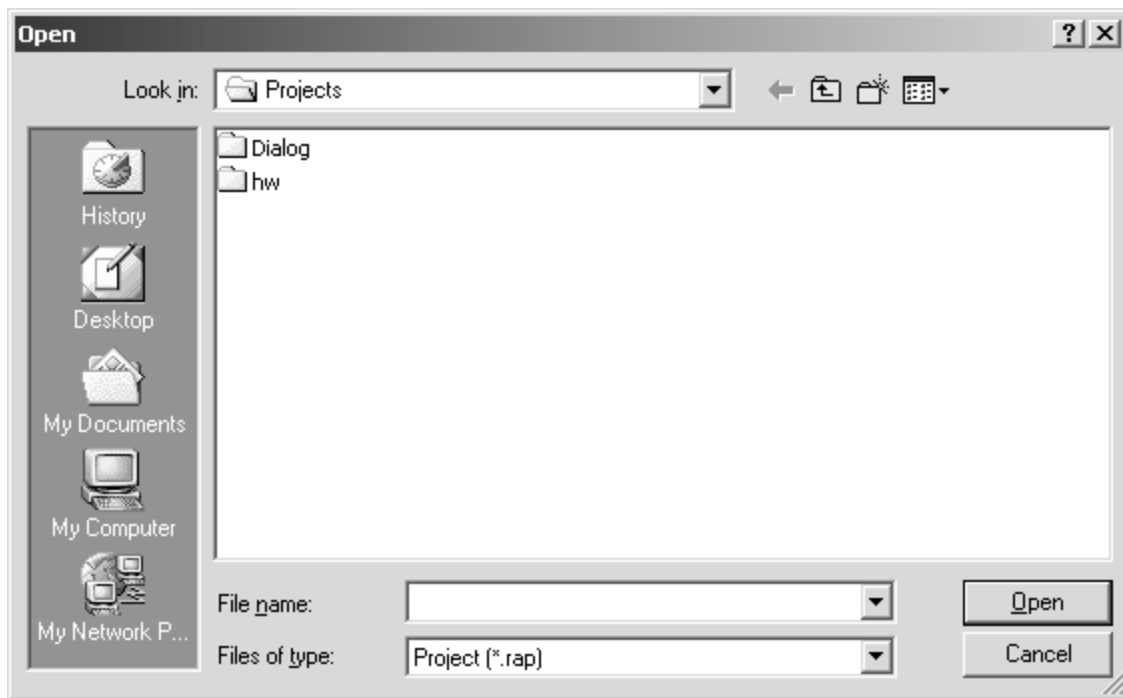
This menu option lets you create a new project using RadASM (this process was explained in detail earlier in this document). When using RadASM/HLA, you're best bet is to always create a new RadASM project using one of the RadASM templates supplied with the RadASM/HLA package. The end result of the "File>New Project" selection is a new project directory with

associated files (including a RadASM “.rap” file that describes the project plus any source files you’ve created for the project).

4.2.15.1.2 File>Open Project

This menu option opens a dialog box that lets you open an existing RadASM project (.rap file). See Figure for details. From this dialog box you can locate the .rap file for your particular project and selecting that file will open the project and its associated files.

RadASM Open Project Dialog Box



4.2.15.1.3 File>Close Project

Selecting this menu item closes any open project (you may only have one project open at a time). If any modifications have been made to any files in the project, you will be asked whether you want to save them before closing the project. Note that this menu item is only active if you have a currently open project.

4.2.15.1.4 File>Delete Project

This menu item deletes the currently open project. Use this option with care. Once you delete a project it is gone. This menu item is only active if you have an open project and it deletes that project.

4.2.15.1.5 File>New File

The options creates a new text file and opens up a window for that text file in the editor. Note that this file does not automatically become a part of any project (including the currently open project, if there is one). See the discussion of the Project menu earlier in this document if you want to insert a file into the currently opened project.

4.2.15.1.6 File>Open

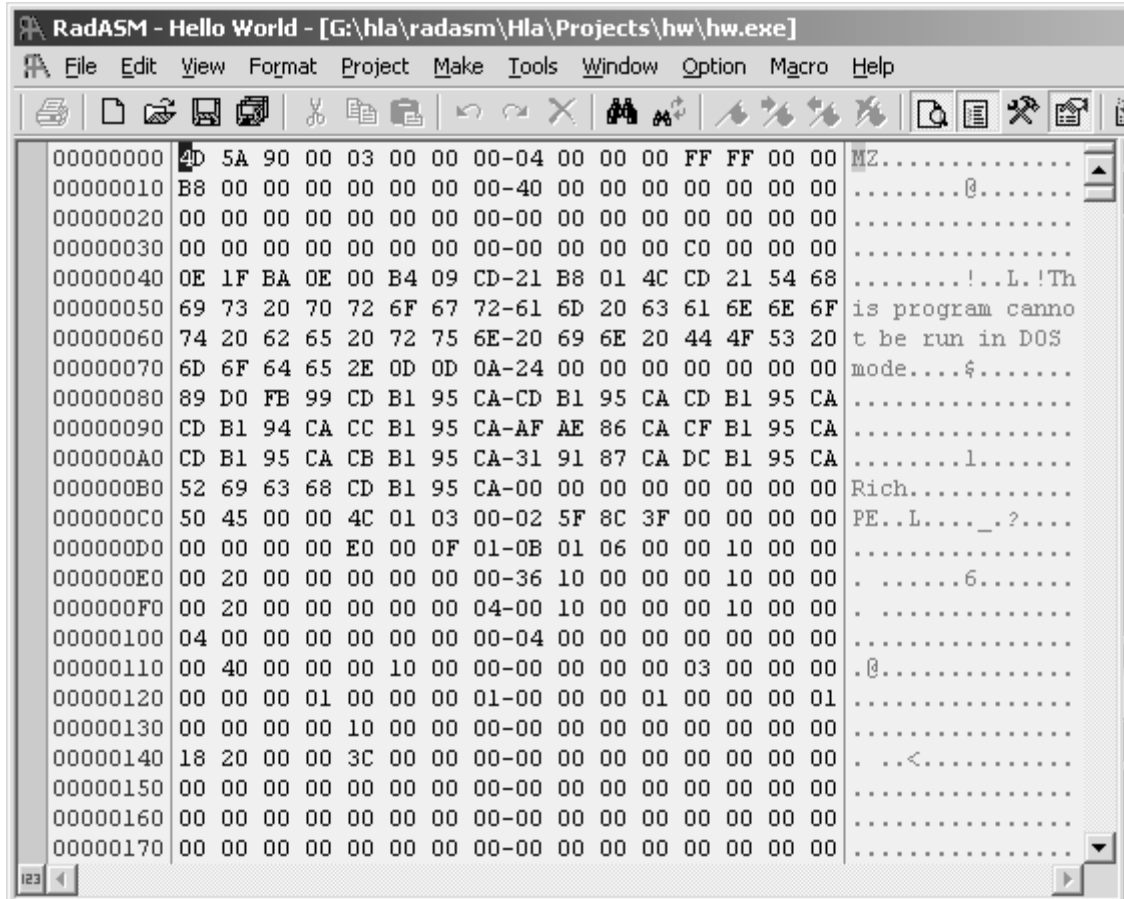
This command opens a text file found on the disk. This file does not have to belong to a currently opened project, and once opened it does not become part of the current project.

4.2.15.1.7 File>Open as Hex

This command opens an arbitrarily typed file (not necessarily a text file) in a hex-editor window (see Figure for an example of the display of the “hw.exe” file in hex format). Note that

you can edit this file (using hex values) and save the result back to disk. This is for advanced programmers only! You can do some serious damage to an executable file if you go poking around in it.

Hex Editor Window



4.2.15.1.8 File>Close File

This command closes the topmost open window. If there are any outstanding modifications, you will be asked if you want to save the file before closing it.

4.2.15.1.9 File>Save File

This saves the top-most open file to disk, without closing the file. Any old data in the file on the disk is replaced.

4.2.15.1.10 File>Save File As

This saves the data in the top-most open file under a different name. The old data in the original file is unchanged. Note that the default name for the top-most file changes to whatever name you supply, so future saves of this file will save their data to the new file rather than the old file.

4.2.15.1.11 File>Save All Files

This quickly saves all modified files that are open in the editor.

4.2.15.1.12 File>Recent Files

This is a hierarchical menu item. Selecting this menu item opens up a secondary menu listing files you've recently edited with RadASM. You may open one of these files by selecting the specified file from the list.

4.2.15.1.13 File>Page Setup

Opens a generic Windows' Printer Set-up dialog.

4.2.15.1.14 File>Print

Opens the generic Window's printer dialog box.

4.2.15.1.15 File>Exit

Quits RadASM.

4.2.15.2 Edit Menu Items

4.2.15.2.1 Edit>Undo, Redo, Cut, Copy, Paste, Delete, Select All

Generic editing options available in most Windows applications

4.2.15.2.2 Edit>Find, Find Next, Find Previous, Replace, Find Word

Opens up a very typical find (or replace) dialog box. (see Figure). Note that by checking the "project" box, you can instruct RadASM to search for a string throughout a project. All the other options are standard Windows' User Interface items that you've seen before.

Find Dialog Box



4.2.15.2.3 Edit>Goto Line

This menu item opens up a small dialog box that lets you enter a line number. RadASM displays that line in the top-most open window.

4.2.15.2.4 Edit>Expand Block

This menu item expands or compresses a begin..end block in an HLA source file (outline mode).

4.2.15.2.5 Edit> Next/Previous/Got/Toggle/Clear Bookmark

RadASM provides a bookmark facility that lets you place markers (bookmarks) on lines of code in your source file. You can quickly navigate between bookmarks by selecting the Next/Previous Bookmark menu items (or by pressing F8 or Ctrl-F8). You can also jump to a specific bookmark (Goto Bookmark) or clear all the set bookmarks in your source file. Bookmarks are especially useful for quickly switching between two sections of code in a source file.

4.2.15.3 The View Menu

The View menu lets you hide or make visible certain components of the RadASM user interface. This menu allows you to enable or disable the following components:

- Toolbar
- Toolbox
- Output Window

- Project Browser
- Properties
- Tab Select
- Info Tool
- Status Bar

4.2.15.4 Format Menu

The format menu contains several useful items that operate on the text within your source file.

4.2.15.4.1 Format>Indent

Indents the selected lines of text by one tab stop (the indentation is to the right).

4.2.15.4.2 Format>Outdent

Outdents the selected lines of text by one tab stop (the outdention is to the left).

4.2.15.4.3 Format>Comment

This command places the HLA line comment delimiter (“//”) at the beginning of each line.

4.2.15.4.4 Format>Uncomment

This command deletes the comment delimiters appearing at the beginning of a block of selected lines.

4.2.15.5 The Project Menu

The project menu contains several items that let you manage your RadASM projects.

4.2.15.5.1 Project>Add New

This menu item lets you add a new file to a project. This is a hierarchical menu item that lets you add source (.hla) files, header (.hhf) files, resource (.rc) files, text (.txt) files, dialog (.dlg) files, menu (.mnu) files, module (.hla) files, or other arbitrary files to your project.

With a bit of project customization, you can have RadASM build a multi-module project by adding module files to your project. However, if you’re interested in creating complex multi-module projects, you’re probably better off using makefiles to control your project builds. For more information about modules in RadASM, please consult the RadASM on-line documentation.

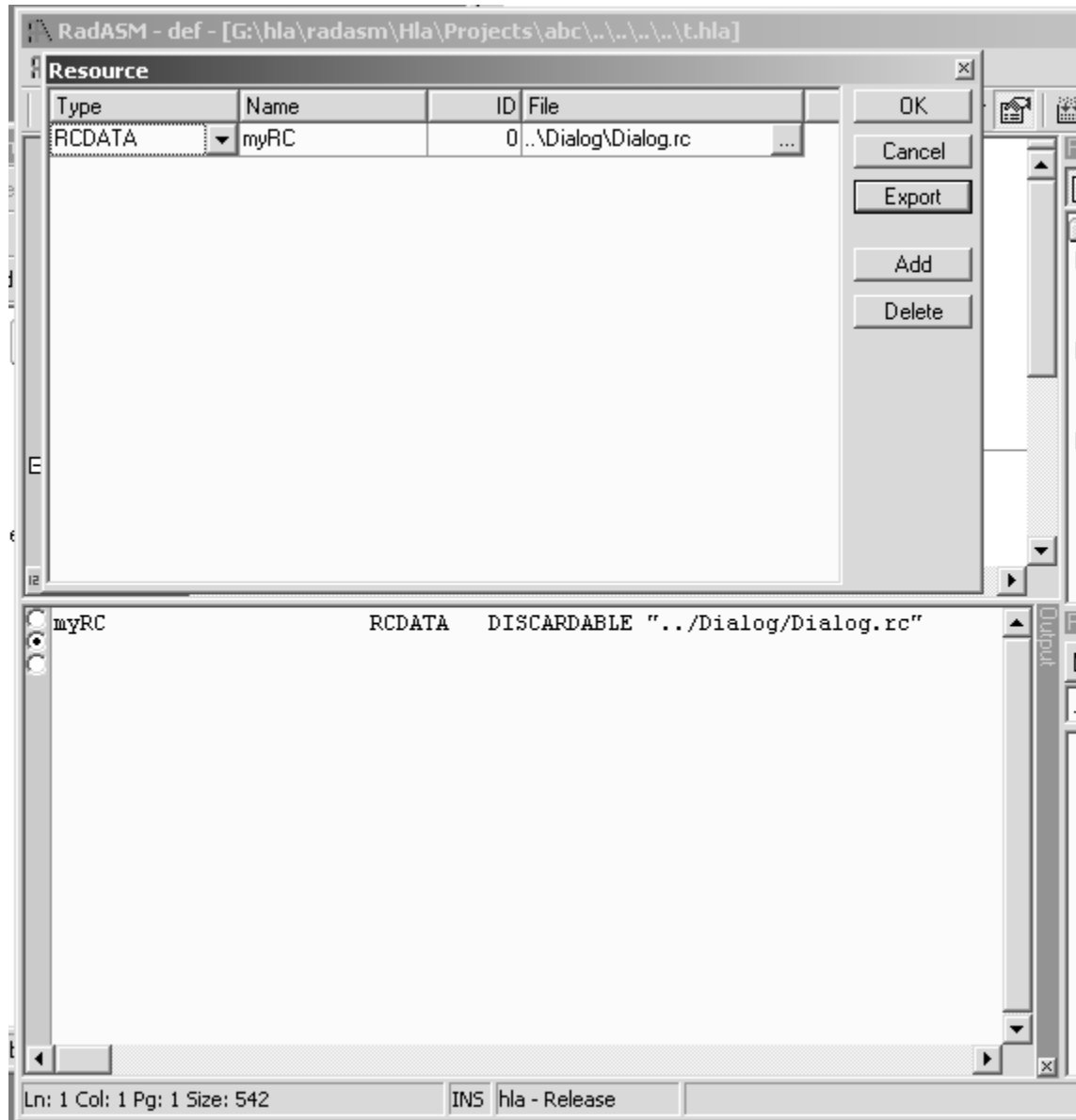
As its name suggests, this menu option creates a new (empty) file to add to a project. You will have to edit the file this option creates in order to place data in the file.

4.2.15.5.2 Project>Add Existing

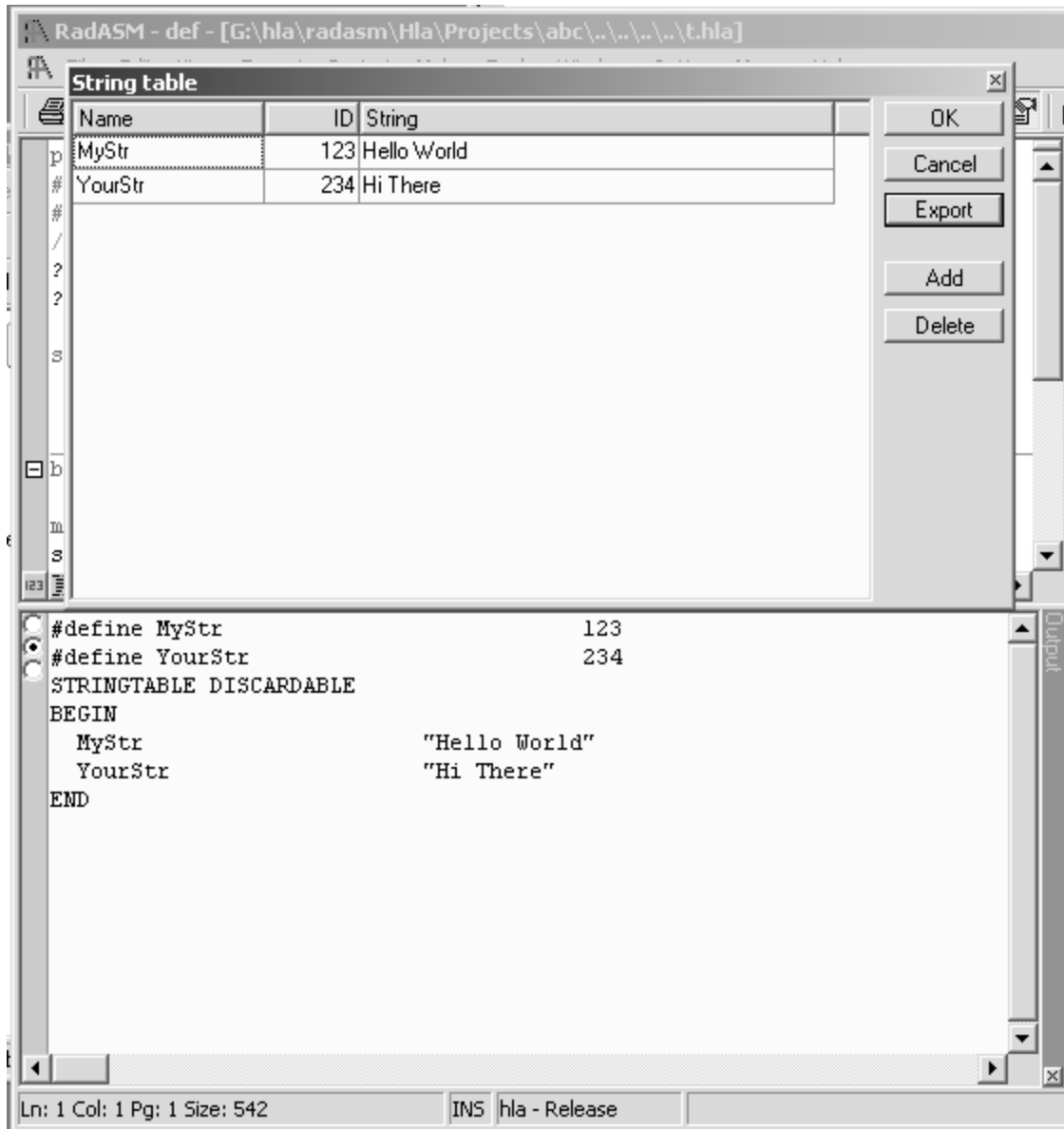
This is another hierarchical menu item that lets you select some file on your hard drive and add it to the current project. If you’ve got some existing files you’d like to convert to a RadASM project, this is the option you use to add those files to a project. Note that this option does not copy the file into your project; it simply creates a link to the file wherever it is on the disk. If you want a copy of the file in your project’s directory you should copy the file to your project directory before adding the file to your project.

4.2.15.5.3 Project>Resource

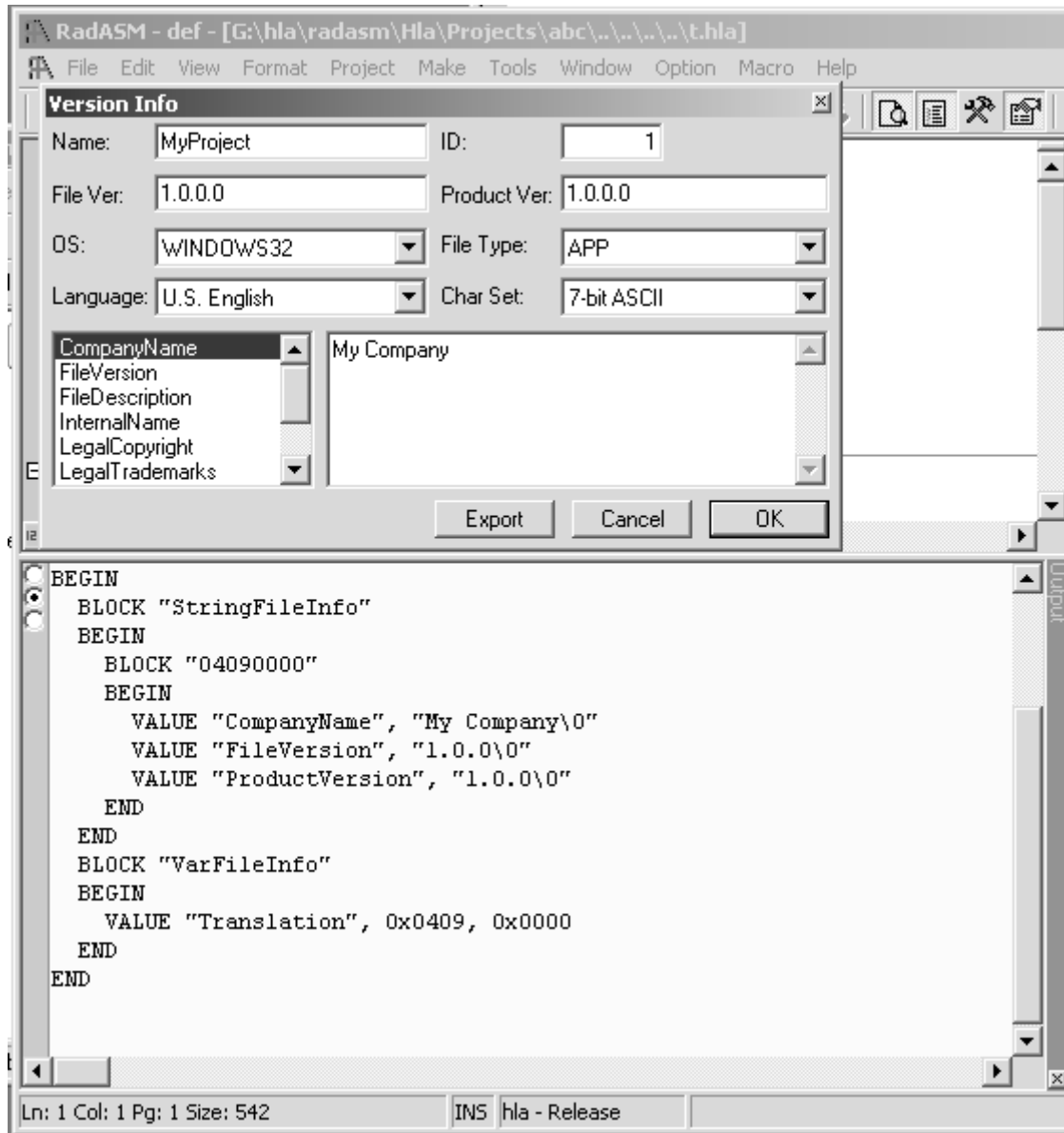
This menu item lets you edit your resource file in a structured fashion. This includes AVI data, Bitmaps, cursors, icons, images, midi data, and so on. This option opens a dialog box that lets you select the resource type, the internal program identifier and value, and the file containing the resource data. By pressing the “export” button in the dialog box that comes up, you can get the text to cut and paste to a resource (.rc) file. See Figure for an example.

Project>Resource Dialog Box (and Export Output)**4.2.15.5.4 Project>Stringtable**

This menu item opens up a dialog box that lets you create string resources. You type in strings, identifiers, and values, and then press the export button (see Figure). The dialog box writes to the output window a data set that you can cut and paste into a resource (.rc) file.

Project>Stringtable Dialog Box**4.2.15.5.5 Project>Versioninfo**

This menu item creates a version information resource. You enter all the appropriate information in the dialog box that comes up (see Figure), press the export button, and RadASM writes a resource (.rc) file compatible block of text to the output window that you can cut and paste to an appropriate resource file.

Project>Versioninfo Menu Item**4.2.15.5.6 Project>Set Assembler**

This option is only available if you've set up RadASM to work with other assemblers in addition to HLA. The RadASM/HLA package leaves this option disabled, by default.

4.2.15.5.7 Project>Remove From Project

This menu item removes the selected file (selected in the project browser) from the project.

4.2.15.5.8 Project>Create Template

This menu item lets you create your own custom templates for RadASM projects. See the on-line help for more details concerning the creation of templates.

4.2.15.5.9 Project>Project Options

This is one of the more important options under the Project menu. This option (which has been discussed earlier) lets you set various options for the currently opened project. This includes the

selection of debug/release mode, which items appear in the make menu, and the specification of commands to execute for each of the items in the make menu. See the discussion earlier in this document or the on-line help for more details.

4.2.15.5.10 Project>Main Project Files

This lets you specify the file types that a project can use. Note that it is *very* dangerous to modify this file list for an existing project. You can easily break RadASM/HLA's build facility by changing these names. Only expert RadASM users should play with this dialog box/menu item. See the on-line help and the RadASM customization guide for more details.

4.2.15.6 Make Menu

The make menu has been fully discussed elsewhere in this document. Note that RadASM/HLA uses a special layout for the Make menu that is not typical of RadASM when used with other assemblers. Therefore, when reading the on-line help, you'll notice that the items don't correspond to the items present in the RadASM/HLA make menu. As it turns out, the items in this menu are customizable on a project by project basis (which is how they got changed for RadASM/HLA). See the section on Customizing RadASM for more details.

4.2.15.7 The Tools Menu

This menu runs various little applications ("applets") including the "snippets" manager, the notepad editor, the Windows calculator, the command line prompt (command window), ASCII table, and a toolbar generator application. Of these, the "snippets manager" is probably of greatest interest to HLA programmers. The snippets manager lets you save short pieces of code ("snippets") that you can cut and paste into your current project. You can expand the available snippets by saving files in the ..RadASM\HLA\Snippets subdirectory.

4.2.15.8 The Window Menu

This menu lets you organize the window display in RadASM. The principle items you use in this menu are the file listings at the bottom of the Window Menu. From here, you can quickly select (and bring to the front) any given editor window that is currently open.

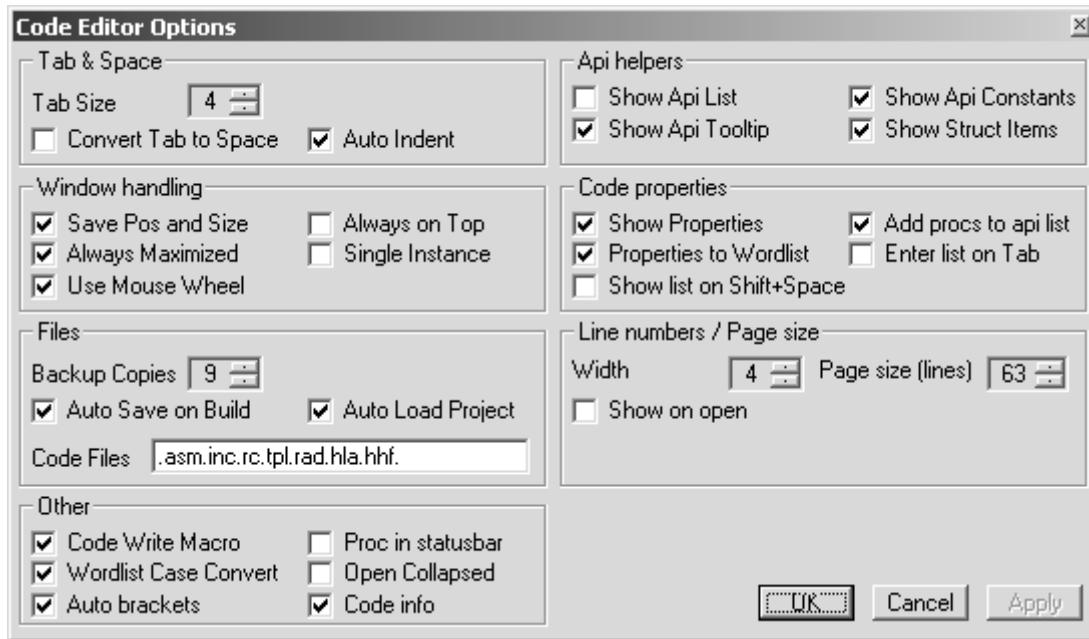
4.2.15.9 The Option Menu

This menu lets you open up dialog boxes that control how RadASM operates. Most of the items in this menu are for advanced RadASM users only, so we'll not spend a whole lot of time discussing them, but a few of the menu items are quite useful and deserve a quick mention.

4.2.15.9.1 Option>Code Editor Options

This menu item opens the Code Editor Options dialog box (see Figure) that lets you set various editor-wide options that are useful while editing projects.

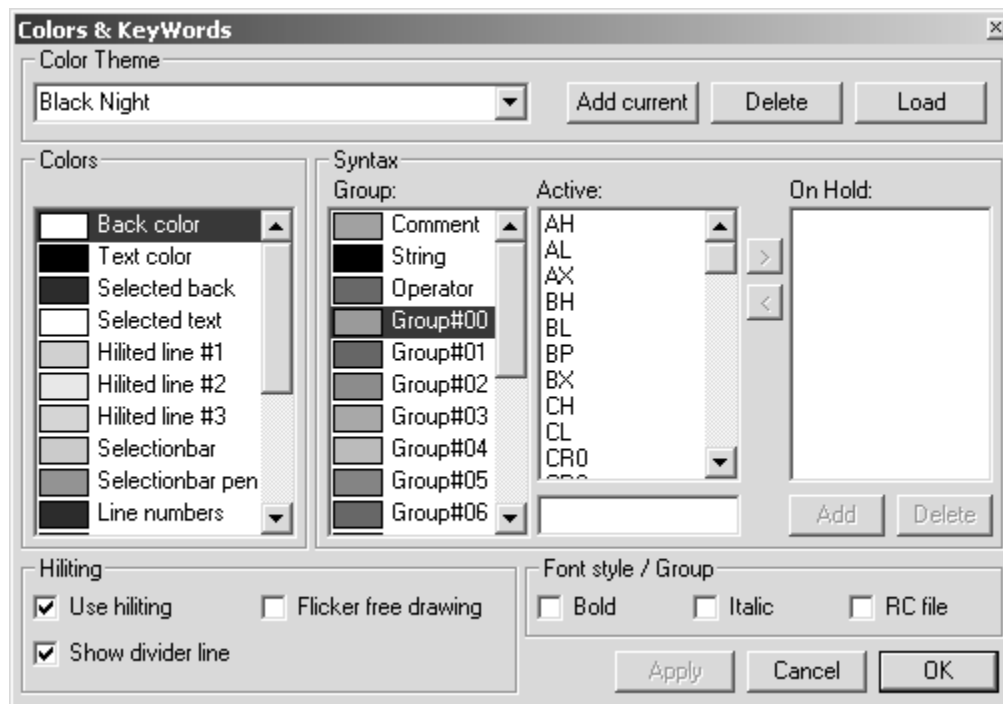
Code Editor Option Dialog Box



4.2.15.9.2 Options>Colors & Keywords

This menu item lets you select the colors that RadASM will use for syntax high-lighting and other purposes within the editor. You can also choose the keywords that RadASM will recognize (for coloring purpose) within this dialog box (see Figure).

Colors & Keyword Dialog Box



4.2.15.9.3 Options>Code Editor Font

This menu item brings up a font selection dialog. RadASM uses this font whenever displaying source code. Note that you should always choose a fixed pitch font (e.g., courier or fixedsys) when editing source code. Mainly, you'll use this option to change the size of the font in your display windows.

4.2.15.9.4 Options>Line Number Font

This brings up a font dialog box that lets you choose the font RadASM uses when displaying line numbers. This is usually a smaller font than used for code or text.

4.2.15.9.5 Options>Text Editor Font

This brings up a font selection dialog box that lets you choose the font used when editing text files. Typically, this would be the same font as the code.

4.2.15.9.6 Options>Printer Options, Printer Font

The Printer Options dialog lets you specify page headings and output color capabilities for print-outs from RadASM.

The Printer Font menu item opens up a font selection dialog that lets you choose the output font when printing text from RadASM.

4.2.15.9.7 Options>File Browser

This menu item opens up a small dialog box that lets you select the directories that RadASM will cycle through when pressing the arrow buttons in the project browser pane. This dialog also lets you select the file filters the project browser window will use when displaying files.

4.2.15.9.8 Options>External File Types

This menu lets you specify various non-RadASM recognized file types and the applications that RadASM will open in order to process such files.

4.2.15.9.9 Options>Snippets

This menu item opens a dialog box that lets you tell RadASM how you want to cut and paste snippets into your code.

4.2.15.9.10 Options>Set Paths

This option brings up a dialog box that lets you specify where RadASM can find certain folders in the system. Generally, it's dangerous to mess with these paths as the RadASM/HLA installation should set these paths up properly for you. Be sure to consult the RadASMini.rtf document (shipped with RadASM) if you want to change these items.

4.2.16 Customizing RadASM

RadASM is a relatively generic integrated development environment for assembly language development. This single program supports the HLA, MASM, TASM, NASM, and FASM assemblers. Each of these different assemblers feature different tool sets (executable programs), command line parameters, and ancillary tools. In order to control the execution of these different programs, the RadASM system uses ".INI" files to let you specifically configure RadASM for the assembler(s) you're using. HLA users will probably want to make modifications to two different ".INI" files that RadASM reads: *radasm.ini* and *hla.ini*. You'll find these two files in the subdirectory containing the *radasm.exe* executable file. Both files are plain ASCII text files that you can edit with any regular text editor (including the editor that is built into RadASM).

The RadASM package includes an ".RTF" (Word/Wordpad) documentation file that explains the basic format of these ".INI" files that RadASM uses. Readers interested in making major changes to these ".INI" files, or those attempting to adopt RadASM to a different assembler, will want to read that document. In this chapter, we'll explore the modifications to a basic set of ".INI" files that a typical HLA user might want to make. The assumption is that you're starting with the stock *radasm.ini* and *hla.ini* files that come with RadASM and you're wanting to customize them to support the development paradigm that this document proposes.

4.2.16.1 The RADASM.INI Initialization File

The *radasm.ini* file specifies all the generic parameters that RadASM uses. In particular, this ".INI" file specifies initial window settings, file histories, OS and language information, and menu entries for certain user modifiable menus. RadASM, itself, actually modifies most of the

information in this “.ini” file. However, there are a few entries an HLA user will need to change and a couple of entries an HLA user may want to change. We’ll discuss those sections here.

Note: there is a preconfigured `radasm.ini` file found in the WPA samples subdirectory. This initialization file is compatible with all the sample programs found in this book and is a good starting point should you decide to make your own customizations to RadASM.

The first item of interest in the `radasm.ini` file is the “[Assembler]” section. This section in the “.INI” file specifies which assemblers RadASM supports and which assembler is the default assembler it will use when creating new projects. By default, the “[Assembler]” section takes the following form:

```
[Assembler]
Assembler=masm, fasm, tasm, nasm, hla
```

The first assembler in this list is the default assembler RadASM will use when creating a new project. The standard `radasm.ini` file is set up to assume that MASM is the default assembler (the first assembler in the list is the default assembler). HLA users will probably want to tell RadASM to use HLA as the default assembler, this is easily achieved by changing the “Assembler=” statement to the following:

```
[Assembler]
Assembler=hla, masm, fasm, tasm, nasm
```

Changing the default assembler is the only “necessary” change that you’ll need to make. However, there are a few additional changes you’ll probably want that will make using RadASM a little nicer. Again, by default, RadASM assumes that you’re developing MASM32 programs. Therefore, the help menu contains several entries that bring up help information for MASM32 users. While some of this information is, arguably, of interest to HLA users, a good part of the default help information doesn’t apply at all to HLA. Fortunately, RadASM’s `radasm.ini` file lets you specify the entries in RadASM’s help menu and where to locate the help files for those menu entries. The “[MenuHelp]” and “[F1-Help]” sections specify where RadASM will look when the user requests help information (by selecting an item from the “Help” menu or by pressing the F1 key, respectively). The default `radasm.ini` file specifies these two sections as follows:

```
[MenuHelp]
1=&Win32 Api,0,H,$H\Win32.hlp
2=&X86 Op Codes,0,H,$H\x86eas.hlp
3=&Masm32,0,H,$H\Masm32.hlp
4=$Resource,0,H,$H\Rc.hlp
5=A&gner,0,H,$H\Agnr.hlp

[F1-Help]
F1=$H\Win32.hlp
CF1=$H\x86eas.hlp
SH1=$H\Masm32.hlp
CSF1=$H\Rc.hlp
```

Each numbered line in the “[MenuHelp]” section corresponds to an entry in RadASM’s “Help” menu. These entries must be sequential numbers starting from one and these numbers specify the order of the item in the “Help” menu (the order in the `radasm.ini` file does not specify the order of the entries in the “Help” menu, you do not have to specify the “[MenuHelp]” entries in numeric order, RadASM will rearrange them according to the numbers you specify). Entry entry in the “[MenuHelp]” section takes the following form:

```
menu# = Menu Text, accelerator, H, helpfile
```

where “menu#” is a numeric value (these values must start from one and there can be no gaps in the set), “Menu Text” is the text that RadASM will display in the menu for that particular item, accelerator is a Windows’ accelerator key value (generally, this is zero, meaning no accelerator

value), “H” is required by RadASM to identify this as a “Help” entry, and “helpfile” is the path to the help file to display (or a program that will bring up a help file).

You may have noticed the ampersand character (“&”) in the menu text. The ampersand precedes the character you can press on the keyboard to select a menu item when the menu is opened. For example, pressing “X” when the menu is open (with the “[HelpMenu]” items in this example) selects the “X86 Op Codes” menu entry.

You will note that the paths in the “[MenuHelp]” section all begin with “\$H”. This is a RadASM shorthand for “the path where RadASM can find all the help files.” There is no requirement that you use this shortcut or even place all your help files in the same directory. You could just also specify the path to a particular help file using a fully qualified pathname like *c:\hla\doc\Win32.hlp*. However, it’s often convenient to specify paths using the various shortcuts that RadASM provides. RadASM supplies the shortcuts found in Table .

Path Shortcuts for Use in RadASM “.INI” Files

Table 3:

Shortcut	Meaning
\$A=	Path to where RadASM is installed
\$B=	Where RadASM finds binaries and executables (e.g., c:\hla)
\$D=	Where RadASM finds “Addin” modules. Usually \$A\AddIns.
\$H=	Where RadASM finds “Help” files. Default is \$A\Help, but you’ll probably want to change this to \$B\Doc.
\$I=	Where RadASM finds include files. Default is \$A\Include, but you’ll probably want to change this to \$B\include.
\$L=	Where RadASM finds library files. Default is \$A\Lib but you’ll probably want to change this to \$B\hllib.
\$R=	Path where RadASM is started (e.g., c:\RadASM).
\$P=	Where RadASM finds projects. This is usually \$R\Projects.
\$S=	Where RadASM find snippets. This is usually \$R\Snippets.
\$T=	Where RadASM finds templates. This is usually \$R\Templates
\$M=	Where RadASM finds keyboard macros. This is usually \$R\Macro

You can define several of these variables in the *hla.ini* file. See the next section for details.

As noted earlier, the default help entries are really intended for MASM32 users and do not particularly apply to HLA users. Therefore, it’s a good idea to change the “[MenuHelp]” entries to reflect the location of some HLA-related help files. Here are the “[MenuHelp]” entries that might be more appropriate for an HLA installation (assuming, of course, you’ve placed all these help files in a common directory on your system):

```
[MenuHelp]
1=&Win32 Api,0,H,$H\Win32.hlp
2=&Resource,0,H,$H\Rc.hlp
3=A&gner,0,H,$H\Agner.hlp
4=&HLA Reference,0,H,$H\PDF\HLARef.pdf
5=HLA Standard &Library,0,H,$H\pdf\HLAStdlib.pdf
6=&Kernel32 API,0,H,$H\pdf\kernelref.pdf
7=&User32 API,0,H,$H\pdf\userRef.pdf
8=&GDI32 API,0,H,$H\pdf\GDIRef.pdf
```

Here's a suggestion for the F1, Ctrl-F1, Shift-F1, and Ctrl-Shift-F1 help items:

```
[F1-Help]
F1=$H\Win32.hlp
CF1=$H\PDF\HLARef.pdf
SF1=$H\pdf\HLAStdlib.pdf
CSF1=$H\Rc.hlp
```

These are probably the extent of the changes you'll want to make to the `radasm.ini` file for HLA use; there are, however, several other options you can change in this file, please see the *radASMini.rtf* file that accompanies the RadASM package for more details on the contents of this file.

4.2.16.2 The HLA.INI Initialization File

The *hla.ini* file is actually where most of the customization for HLA takes place inside RadASM. This file lets you customize RadASM's operation specifically for HLA¹. The `hla.ini` file appearing in the WPA subdirectory (on the accompanying CD-ROM or in the Webster HLA/Examples download file) contains a set of default values that provide a good starting point for your own customizations.

Note: although *hla.ini* provides a good starting point for a system, you will probably need to make changes to this file in order for it to work on your specific system. Without these changes, RadASM may not work on your system.

Without question, the first section to look at in the *hla.ini* file is the section that begins with “[Paths]”. This is where you tell RadASM the paths to various directories where it expects to find various files it needs (see Table for the meaning of these various path values). A typical “[Paths]” section might look like the following:

```
[Paths]
$A=C:\Hla
$B=$A
$D=$R\AddIns
$H=$A\Doc
$I=$A\Include
$L=$A\hlalib
$P=$R\Hla\Projects
$S=$R\Hla\Snippets
$T=$R\Hla\Templates
$M=$R\Hla\Macro
```

Note that the `$A` prefix specifies the path where RadASM can find the executables for HLA. In fact, RadASM does not run HLA directly (remember, we're going to have the make program run HLA for us), but the application path (`$A`) becomes a prefix directory we'll use for defining other directory prefixes. **Be sure to check this path** in your copy of the `hla.ini` file and verify that it points at your main HLA subdirectory (usually “C:\HLA” though this may be different if you've installed HLA elsewhere).

The `$R` prefix specifies the path to the subdirectory containing RadASM. RadASM automatically sets up this prefix, you don't have to explicitly set its value. The remaining subdirectory paths are based off either the `$A` prefix or the `$R` prefix.

The “[Project]” section of the *hla.ini* file is where the fun really begins. This section takes the following form in the default file provided in the WPA subdirectory:

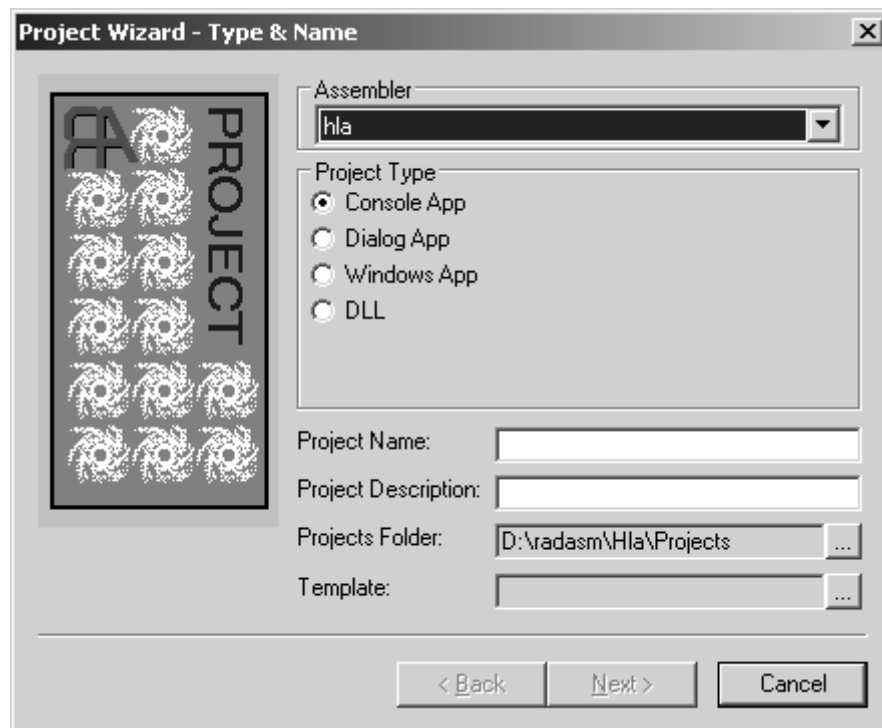
```
[Project]
```

1. There are comparable initialization files for MASM, TASM, NASM, FASM, and other assemblers that RadASM supports.

```
Type=Console App,Dialog App,Windows App,DLL
Files=hla,hhf,rc,def
Folders=Bak,Res,Tmp,Doc
MenuMake=Build,Build All,Compile RC,Check Syntax,Run
Group=1
GroupExpand=1
```

The line beginning with “Type=” specifies the type of projects RadASM supports for HLA. The default configuration supports console applications (“Console App”), dialog applications (“Dialog App”), Windows applications (“Windows App”), and dynamic linked library (“DLL”). The names are arbitrary, though other sections of the *hla.ini* file will use these names. Whenever you create a new project in HLA, it will create a list of “Project Type” names based on the list of names appearing after “Type=” in the “[Project]” section. Adding a string to this comma-separated list will add a new name to the project types that the RadASM user can select from (note, however, that to actually support these project types requires some extra work later on in the *hla.ini* file). Figure shows what the New Project dialog box in RadASM displays in response to the entries on the “Type=...” line.

RadASM Project Types



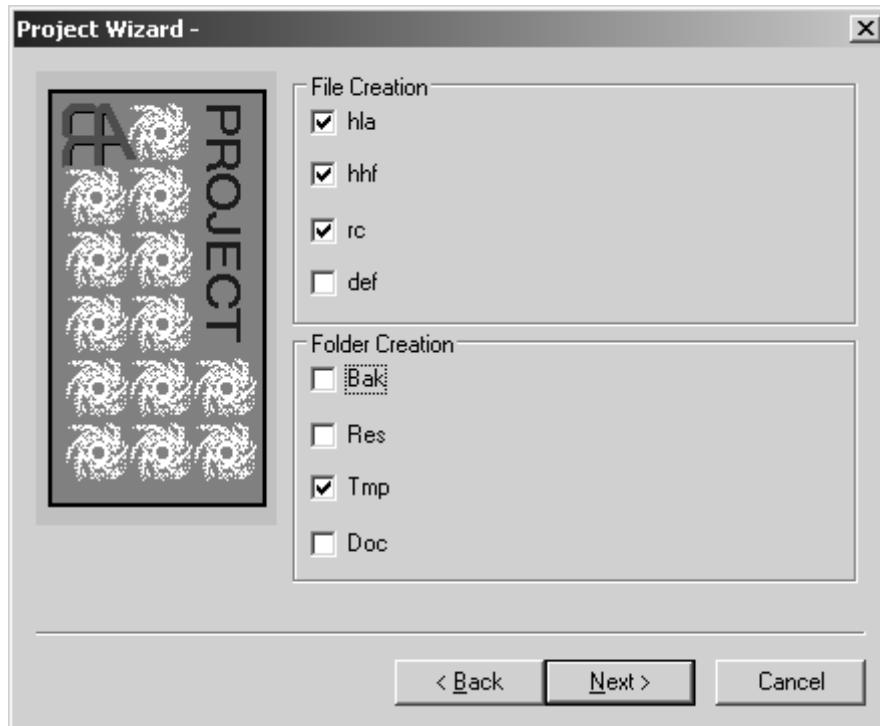
The line beginning with “Files=” in the “[Project]” section specifies the suffixes for the files that RadASM will associate with this project. The “hla” and “hhf” entries, of course, are the standard file types that HLA uses. The “rc” file type is for *resource compiler* files (we’ll talk about the resource compiler in a later chapter). If you want to be able to create additional file types and include them in a RadASM project, you would add their suffix here.

The “Folders=...” statement tells RadASM what subdirectories it should allow the user to create when they start a new project. The make file system we’re going to use will assume the presence of a “Tmp” directory, hence that option needs to be present in the list. the “bak”, “res”, and “doc” directories let the user create those subdirectories.

Figure shows the dialog box that displays the information found on the “Files=” and “Folders=” lines. By checking the appropriate boxes in the File Creation group, the RadASM user

can tell RadASM to create a file with the project's name and the appropriate suffix as part of the project. Similarly, by checking the appropriate boxes in the Folder Creation group, the RadASM user can tell RadASM to create the appropriate directories.

File and Folder Creation Dialog Box in RadASM

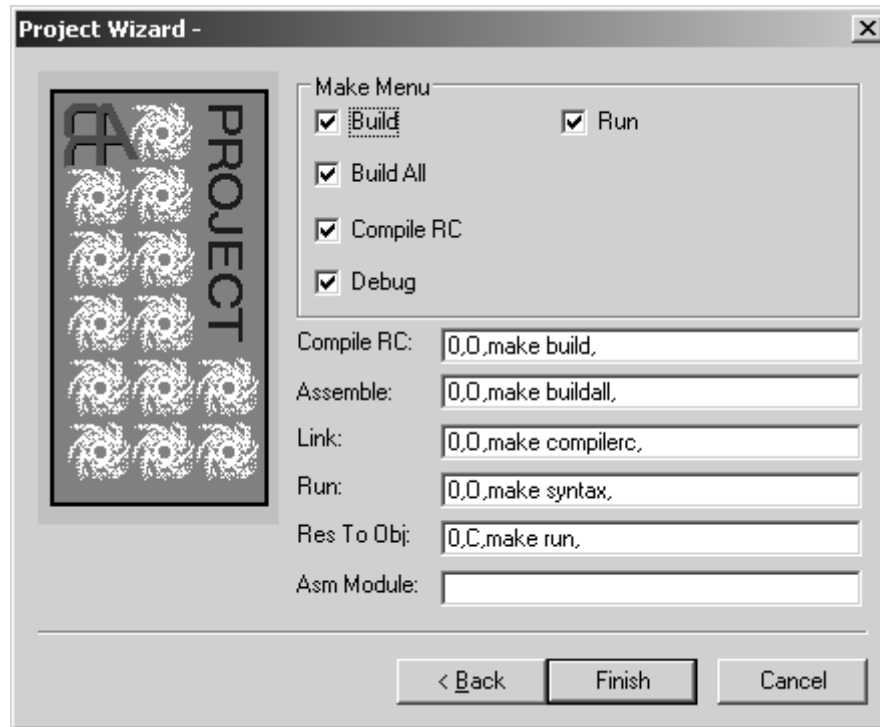


The “MenuMake=...” line specifies the IDE options that will be available for processing the files in this project. Unlike the other options, you cannot specify an arbitrary list of commands here. RadASM provides five items in the Make menu that you can modify, you don't have the option of adding additional items here (you can disable options if you want fewer, though). Originally, these five slots were intended for the following commands:

- Compile RC (compile a resource file)
- Assemble (assemble the currently open file)
- Link (create an EXE from the object files)
- Run (execute the EXE file, building it if necessary)
- Res To Obj (convert a resource file to an object file)

Because these options aren't as applicable to HLA projects as they are to MASM projects (on which the original list was built), the default *hla.ini* file co-opts these make items for operations that make more sense for the way we're going to be building HLA applications in this book. You can actually turn these make items on or off on a project by project basis (for certain types of projects, certain make objects may not make sense). Figure shows the dialog box that RadASM displays and presents this information. Note that in the version used here, RadASM only displays the correct labels for the check boxes in the Make Menu group. The labels on the text entry boxes should also be “Build”, “Build All”, “Compile RC”, “Check Syntax”, and “Run” (in that order), but these labels turn out to be hard-coded to the original MASM specifications. Fortunately, you won't normally use these text entry boxes (the default text appearing in them appears in the *hla.ini* file), so you can ignore the fact that they are mis-labelled here.

RadASM Make Menu Selection Dialog Box



For each of the project types you specify on the “Type= . . .” line in the “[Project]” section, you must add a section to the *hla.ini* file, using that project type’s name, that tells RadASM how to deal with projects of that type. In the *hla.ini* file we’re discussing here, the project types are “Console App”, “Dialog App”, “Windows App”, and “DLL” so we will need to have sections named “[Console App]”, “[Dialog App]”, “[Windows App]”, and “[DLL]”. It also turns out that RadASM requires one additional section named “[MakeDefNoProject]” that RadASM uses to process files in the IDE that are not associated with a specific project.

When running RadASM, you can have exactly one project open (or no project at all open, just some arbitrary files) at a time. This project will be one of the types specified on the “Type= . . .” line in the “[Project]” section. Based on the open project, RadASM may execute a different set of commands for each of the items in the Make menu; the actual commands selected are specified in the project-specific sections of the *hla.ini* file. Here’s what the “[MakeDefNoProject]” section looks like:

```
[MakeDefNoProject]
MenuMake=1,1,1,1,1
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,O,make run,
11=0,O,make dbg_build,
12=0,O,make dbg_buildall,
13=0,O,make dbg_compilerc,
14=0,O,make dbg_syntax,
15=0,C,make dbg_run,
```

The “MenuMake=...” line specifies which items in the RadASM Make menu will be active when a project of this type is active in RadASM (or, in the case of `MakeDefNoProject`, when no project is loaded). This is a list of boolean values (true=1, false=0) that specify whether the menu items in the Make menu will be active or deactivated. Each of these values correspond to the items on the MenuMake line in the “[Project]” section (in our case, this corresponds to “Build”, “Build All”, “Compile RC”, “Syntax Check”, and “Run”, in that order). A “1” activates the corresponding menu item, a zero deactivates it. For most HLA project types, we’ll generally leave all of these options enabled. The exception is DLL; normally you don’t “run” DLLs so we’ll disable the run option when building DLL projects.

The remaining lines specify the actions RadASM will take whenever you select one of the items from the Make menu. To understand how these items work, let’s first take a look at another section in the `hla.ini` file, the “[MenuMake]” section:

```
[MenuMake]
1=&Build,55,M,1
2=Build &All,31,M,2
3=&Compile RC,91,M,3
4=&Syntax,103,M,4
5=-,0,M,
6=&Run,67,M,5
```

Each item in the “[MenuMake]” section corresponds to a menu entry in the Make menu. The numbers specify the index to the menu entry (e.g., “1=” specifies the first menu item, “2=” specifies the second menu item, etc.). The first item after the “n=” prefix specifies the actual text that will appear in the Make menu. If this text is just the character “-” then RadASM displays a menu separator for that particular entry. As you can see, the default menu entries are “Build”, “Build All”, “Compile RC”, “Syntax”, and “Run”.

The next item, following the menu item text, is the accelerator value. These are “magic” values that specify keystrokes that do the same job as selecting items from the menu. For example, 55 (in the “Build” item) corresponds to Shift+F5, 31 (in “Build All”) corresponds to F5. We’ll discuss accelerators in a later chapter. So just ignore (and copy verbatim) these files for right now.

The third item on each line is always the letter “M”. This tells RadASM that this is a make menu item.

The fourth entry on each line is probably the most important. This is the command to execute when someone selects this particular menu item. This is either some text containing the command line to execute or a numeric index into the current project type. As you can see in this example, each of the commands use an index value (one through five in this example). These numbers correspond to the lines in each of the project sections. For example, if you select the “Build” option from the Make menu, RadASM notes that it is to execute command #1. It goes to the current project type section and locates the line that begins with “1=...” and executes that operation, e.g.,

```
1=0,0,make build,
```

In a similar vein, selecting “Build All” from the Make menu instructs RadASM to execute the command that begins with “2=...” in the current project type’s section (i.e., “2=0,0,make buildall,”). And so on.

The lines in the project type section are divided into two groups, those that begin with 1, 2, 3, 4, or 5 and those that begin with 11, 12, 13, 14, or 15. The “[MenuMake]” command index selects one of the commands from these two groups based on whether RadASM is producing a “release build” or a “debug build”. Release builds always execute the command specified by the “[MenuMake]” command index (i.e. 1-5). If you’re building a debug version, then RadASM executes the commands in the range 11-15 in response to command indexes 1-5. We’ll ignore debug builds for the time being (we’ll discuss them in a later chapter on debugging). So for right now, we’ll always assume that we’re building a release image.

The fields of each of the indexed commands in the project type section have the following meanings:

```
index = delete_option, output_option, command, files
```


The *delete_option* item specifies which files to delete before doing the build. If this entry is zero, then RadASM will not delete any files before the build. Because we're having a make file do the actual build for us, and it can take care of cleaning up any files that need to be deleted first, we'll always put a zero here when using RadASM with HLA.

The *output_option* item is either "C", "O" (that's an "oh" not a "zero"), or zero. This specifies whether the output of the command will go to a Windows console window ("C"), the RadASM output window ("O", which is "oh"), or the output will simply be thrown away (zero). We'll usually want the output sent to RadASM's output window, so most of the time you'll see the letter "O" ("oh") here.

The *command* entry is the command line text that RadASM will pass on to windows whenever you execute this command. This can be any valid command prompt operation. For our purposes, we'll always use a make command with a single parameter to specify the type of make operation to perform. Here are the commands we're going to support in RadASM:

- Build - "make build"
- Build All - "make buildall"
- Compile RC - "make compilerc"
- Syntax - "make syntax"
- Run - "make run"

Now it's up to the makefile to handle each of these various commands properly (using the standard makefile scheme we defined in the first chapter).

This may seem like a considerable amount of indirection -- why not just place the commands directly in the "[MenuMake]" section? However, this scheme is quite flexible and makes it easy to adjust the options on a project type by project type basis (in fact, it's even possible to set these options on a project by project basis).

With this discussion out of the way, it's time to look at the various project type sections. Without further ado, here they are:

```
[Console App]
Files=1,1,1,1,0,0
Folders=1,0,1,0
MenuMake=1,1,1,1,1,0,0,0
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,C,make run,
11=0,O,make build,
12=0,O,make buildall,
13=0,O,make compilerc,
14=0,O,make syntax,
15=0,C,make run,
```

Console applications, by default, want to create an .HLA file and a .HHF file, a BAK folder and a TMP folder. All menu items are active for building and running console apps (that is, there are five ones after "MenuMake"). Finally, the commands ("1=..." "2=...", etc.) are all the standard build commands.

```
[Dialog App]
Files=1,1,1,0,0
Folders=1,1,1
MenuMake=1,1,1,1,1,0,0,0
1=0,O,make build,
2=0,O,make buildall,
3=0,O,make compilerc,
4=0,O,make syntax,
5=0,C,make run,
```

```
11=0,O,make build,  
12=0,O,make buildall,  
13=0,O,make compilerc,  
14=0,O,make syntax,  
15=0,C,make run,
```

By default, dialog applications will create HLA, HHF, RC, and DEF files and they will create a BAK and a TMP subdirectory. All five menu items will be active and dialog apps use the standard command set.

```
[Windows App]  
Files=1,1,1,1,0  
Folders=1,1,1,1  
MenuMake=1,1,1,1,1,0,0,0  
1=0,O,make build,  
2=0,O,make buildall,  
3=0,O,make compilerc,  
4=0,O,make syntax,  
5=0,C,make run,  
11=0,O,make build,  
12=0,O,make buildall,  
13=0,O,make compilerc,  
14=0,O,make syntax,  
15=0,C,make run,
```

By default, window applications will create HLA, HHF, RC, and DEF files and they will create a BAK, RES, DOC, and a TMP subdirectory. All five menu items will be active and dialog apps use the standard command set.

The hla.ini file allows you to control several other features in RadASM. The options we've discussed in this chapter are the crucial ones you must set up, most of the remaining options are of an aesthetic or non-crucial nature, so we won't bother discussing them here. Please see the RadASM documentation (the RTF file mentioned earlier) for details on these other options.

Once you've made the appropriate changes to the hla.ini file (and, of course, you've made a backup of your original file, right?), then you can copy the file to the RadASM subdirectory and replace the existing hla.ini file with your new one. After doing this, RadASM should operate with the new options when you run RadASM..

5 HLA Internal Operation

To effectively use HLA, it helps to understand how HLA translates HLA source files into executable machine code. This information is particularly useful if you install HLA incorrectly and you cannot successfully compile a simple demo program. Beyond that, this information can also help you take advantage of more advanced HLA and OS features.

As noted earlier in the HLA documentation, HLA is not a single application; the HLA system is a collection of programs that work together to translate your HLA source files into executable files. This is not unusual, most compilers and assemblers provide only part of the conversion from source to executable (e.g., you still have to run a linker with most compilers and assemblers to produce an executable).

The HLA system offers a rich set of different configurations that allow you to mix and match components to efficiently process your assembly language applications. First of all, HLA is relatively *portable*. The compiler itself is written with Flex, Bison, C/C++, along with some platform-independent assembly language code (written in HLA, of course). This makes it easy to move the compiler from one operating system to another. Currently, HLA is supported under Windows, Linux, FreeBSD, and Mac OSX. Plans include porting HLA to NetBSD, OpenBSD, QNX and, perhaps, Solaris at some point in the future. Even within a single operating system, HLA offers multiple configurations that you can employ, based on your needs and desires. This section will describe some of the possible configurations you might create.

The compilation of a typical HLA source file using a command line such as "hla hw" goes through three or four major phases:

- The hla.exe (Windows) or hla (other OSes) program processes command-line parameters and acts as a "traffic cop" directing the execution of the remaining components of the HLA system.
- The hlaparse.exe (Windows) or hlaparse (other OSes) program is responsible for translating the HLA source file either into an object file or into the syntax of some other assembler. Usually, the hlaparse program is run automatically by some other program such as hla.exe/hla or HIDE (the HLA Integrated Development Environment). You would not normally run hlaparse directly from a command-line (though it is certainly possible to do this if you are so inclined).
- If you elect to have HLA produce an assembly language output file rather than an object module, then the next step towards producing an executable file is to run the associated back-end assembler on the source output that HLA produced. This step isn't strictly necessary because HLA can produce an object file directly without using some external assembler, but there are some (rather esoteric) reasons why you might want to go through some other assembler rather than having HLA directly produce the object file. Generally, the hla.exe (hla) program will automatically run the assembler for you.
- The last step is to run a linker to combine the object module the previous steps created with the HLA Standard Library and any other necessary object modules for the project. The output of the linkage step is an executable file (assuming, of course, there were no errors in the compilation of your program). Generally, the hla.exe (hla) program will automatically run the linker for you.

There is a fifth, optional, step that can also take place under Windows. If you are creating an application that makes use of compiled resources, as the fourth step (before the linking stage) the hla.exe (Windows only) program can run a resource compiler to translate those resources into an object module (.res) that the linker can link into your final executable.

As it turns out, the HLA system can employ a wide variety of linkers, librarians, assemblers, and other tools based on the underlying operating system. Here is the list of tools that HLA has been qualified with:

Under Windows:

- Microsoft's MASM v9 assembler
- Microsoft's linker (v9)
- Microsoft's resource compiler
- Microsofts LIB library manager
- The Flat Assembler (FASM)

- The Netwide Assembler (NASM)
- Pelles C POLINK linker
- Pelles C POLIB library manager
- Pelles C PORC resource compiler
- Borland's Turbo Assembler

Note: you can use Borland's TLINK and TLIB utilities with HLA, but you will have to manually run these applications; the HLA system will not automatically execute them. Note that HLA v2.0 and later have deprecated support for the Borland tools and future versions may not support them at all.

Under Linux, Mac OSX, and FreeBSD:

- The Free Software Foundation's (FSF) Gas assembler (as)
- FSF's linker (ld)

A couple of obvious questions that might come up: "Why provide all these options? Why not simply pick a single configuration and go with that?" Well, as it turns out, there are advantages and disadvantages to each configuration and allowing multiple configurations affords you the most flexibility when writing code.

The first point to consider is object code file versus assembly language output. The obvious choice is to have HLA produce an object code file. After all, the job of an assembler is to produce an object code file; why make the detour of producing an intermediate assembly language file that some other assembler must convert to an object file?

Historically, HLA provides this option because this was the only option available under HLA v1.x (direct object code output became available in HLA v2.0). Persons with old makefiles and other build systems may be expecting to run HLA output through some other assembler. The fact that HLA v2.x maintains the ability to produce object code in this manner preserves those existing make files and build systems.

Another reason for producing an intermediate assembly language file is because you want to see the output from the HLA compiler (for example, to see how it translates HLL-like statements into pure assembly language). The HLA compiler has an HLA output mode specifically for this purpose. It may seem silly, at first, to assemble an HLA source file into a (lower-level) HLA source file, but being able to look at the code that HLA generates is sometimes a very nice feature.

One last reason for producing an assembly language output file from an HLA source file is to allow you to translate HLA source code into the syntax for some other assembler (MASM, FASM, NASM, or Gas). This is useful when you want to combine HLA code with a project written with a different assembler.

Most of the time, of course, you'll use the default setting and directly produce an object file from an HLA assembly. HLA produces standard PE/COFF, ELF, and Mach-o object-code files, so HLA's output will properly link with other object files (perhaps written in different languages).

6 Using the HLA Command-Line Compiler

Once you've installed HLA and verified that it is operational, you can run the HLA compiler. The HLA compiler consists of two executables: `hla(.exe)`¹, which is a shell that processes command line arguments, compiles ".hla" files to ".asm" files, assembles the ".asm" files by calling an assembler, and links the resulting files together using a linker program; the second executable is `hlaparse(.exe)` which compiles a single ".hla" file to an assembly language file. Generally, you would only run `hla(.exe)`. The `hla(.exe)` program automatically runs the `hlaparse(.exe)` and assembler/linker programs. The `hla(.exe)` command uses the following syntax:

```
hla optional_command_line_parameters Filename_list
```

The filenames list consists of one or more unambiguous filenames having the extension: ".hla", ".asm" or ".obj"/".o"². HLA will first run the `hlaparse(.exe)` program on all files with the HLA extension (producing files with the same basename and an ASM extension). Then HLA runs the assembler on all files with the ".asm" extension (including the files produced by `hlaparse`). Finally, HLA runs the linker to combine all the object files together (including the ".obj"/".o" files the assembler produces). The ultimate result, assuming there were no errors along the way, is an executable file (with an EXE extension under Windows, with no extension under Linux/FreeBSD/Mac OSX).

HLA supports the following command line parameters:

```
options:
-@           Do not generate linker response file.
-@@          Always generate a linker response file.
-thread      Enable thread-safe code generation and linkage.
-axxxxx     Pass xxxxx as command line parameter to assembler.
-dxx        Define VAL symbol xx to have type BOOLEAN and value TRUE.
-dxx=yy     Define VAL symbol xx to have type STRING and value "yy".
-e:name     Executable output filename (appends ".exe" under Windows).
-x:name     Executable output filename (does not append ".exe").
-b:name     Binary object file output name (only when using HLABEL).
-i:path     Specifies path to HLA include file directory.
-lib:path   Specifies path to the HLALIB.LIB file.
-license    Displays copyright and license info for the HLA system.
-lxxxxx    Pass xxxxx as command line parameter to linker.
-m          Create a map file during link
-p:path     Specifies path to hold temporary working files.
-r:name     <name> is a text file containing cmd line options.
-obj:path   Specifies path to place object files.
-main:name  Use 'name' as the name of the HLA main program.
-source    Compile to human readable source file format.
-s          Compile to .ASM files only.
-c          Compile and assemble to object files only.
-fasm      Use FASM as back-end (applies to -s and -c)
-gas       Use Gas as back-end (Linux/BSD, applies to -s and -c)
-gasx      Use Gas as back-end (Mac OSX, applies to -s and -c)
```

1. The ".exe" suffix appears only in the Windows' version.
2. Windows object files use the ".obj" suffix while Linux object files have the ".o" suffix. Although Linux users who write assembly code with Gas typically use a ".s" or ".S" suffix, HLA still uses ".asm" since Gas happily accepts this.

```

-hla          Produce a pseudo-HLA source file as output (implies -s).
-hlabe       (Default) Produce obj file using the HLA Back Engine.
-masm       Use MASM as back-end assembler (applies to -s and -c)
-nasm       Use NASM as back-end assembler (applies to -s and -c)
-tasm       Use TASM as back-end assembler (applies to -s and -c)
-sym        Dump symbol table after compile.
-win32      Generate code for Win32 OS.
-linux      Generate code for Linux OS.
-freebsd    Generate code for FreeBSD OS.
-macos      Generate code for Mac OSX.
-test       Send diagnostic info to stdout rather than stderr (This
           option is intended for HLA test/debug purposes).
-v          Verbose compile.
-level=h    High-level assembly language
-level=m    Medium-level assembly language
-level=l    Low-level assembly language
-level=v    Machine-level assembly language (very low level).
-w          Compile as windows app (default is console app).
-?          Display this help message.

```

Note that HLA ignores case when processing command line parameters (unlike typical Linux/FreeBSD/Mac OSX programs). For example, "-s" is equivalent to "-S" when specifying a command line parameter.

```

-@
-@@

```

HLA will produce a "linker response file" that it supplies to the linker program during the link phase. This linker response file contains necessary segment declarations and other vital linker information. By default, HLA uses any existing "*basename*.link" file (where *basename* is the base name of the file you are compiling) whenever you run the compiler; it will create a new "*basename*.link" file only if one does not already exist. The "-@" option tells HLA not to create a new ".link" file, even if one does not already exist. The "-@@" option tells HLA to always create a ".link" file, even if one already exists.

If you specify multiple "*basename*.hla" filenames on the command line, HLA only generates a single "*basename*.link" file using the name of the first "*basename*.HLA" file it encounters.

```
-r:filename
```

The "-r:filename" option lets you specify a response file containing a sequence of HLA command-line parameters. The file specified after this option must contain a sequence of HLA command-line parameters, one per line, which HLA executes exactly as though they were specified on the command line. E.g.,

```
sampleFile.resp:
```

```
-fasm
sampleFile.hla
```

The following command treats each of the above lines as separate HLA command-line parameters:

```
hla -r:sampleFile.resp
```

```
-aXXXXXX
```

The -aXXXXXX option lets you pass assembler-specific command line options to the assembler during the assembler phase. This option is ignored if you use the -s option or you're compiling directly to object code using the HLA back engine. One common form of this command often used

with the MASM assembler is "-aZi -aZf" that tells MASM to generate debugging information in the object file (for use with the Visual Studio debugger program).

-c

The -c option tells HLA to run the hlaparse compiler and the (default) assembler, producing "*basename.obj*"/"*basename.o*" files. HLA will process all filenames on the command line that have ".hla" or ".asm" extension, but it will ignore any filenames with ".obj" or ".o" extensions. If you compile an HLA unit without compiling an HLA program at the same time, you will need to use this option or the linker will complain about not finding the main program.

One common use of this option is to compile HLA units to object files. Since HLA units do not contain a main program, you cannot compile an HLA unit directly to an executable. To compile an HLA unit separately (i.e., without compiling an HLA main program during the same HLA.EXE invocation) you must specify the "-c" option or the compilation will generate an error when it attempts to link the program.

A second reason for using the "-c" option is that you want to explicitly run the linker yourself and supply linker command line options that are different from those that HLA automatically provides.

-fasm

Tells HLA to use FASM as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file output with this command are available only under Windows. Source output is available under any operating system.

-gas

Tells HLA to use GAS as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file output with this command are available only under Linux and FreeBSD. Source output is available under any operating system.

-gasx

Tells HLA to use GAS as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file output with this command are available only under Mac OS X. Source output is available under any operating system.

-hla

Tells HLA to produce a human-readable pseudo-HLA-syntax assembly language output file. This command-line option implies "-source" and "-s". The main use for this option is to see how HLA expands macros, high-level-language-like statements, and other code.

-hlabe

Tells HLA to use the HLA Back Engine as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. This command causes HLA to directly produce an object code file without using an external back-end assembler. If you specify both of the "-source" and "-s" command-line options along with "-hlabe", then HLA will produce a human-readable ".asm" file rather than an object file; this option is useful for debugging HLA or for those curious about how HLABE operates.

-masm

Tells HLA to use MASM as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file

output with this command are available only under Windows. Source output is available under any operating system.

-nasm

Tells HLA to use NASM as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file output with this command are available only under Windows. Source output is available under any operating system.

-tasm

Tells HLA to use TASM as the back-end assembler. This command-line overrides any back-end assembler specification previously on the command-line. Object code and executable file output with this command are available only under Windows. Source output is available under any operating system. Note that TASM support in HLA is deprecated, so it's not a good idea to depend on this option.

-d:XXXXX{=YYYYY}

The **-dXXXXX** option tells HLA to define the symbol *XXXXX* as a boolean **val** constant and initialize it with the value **true**. Generally you use such symbols to control the emission of code during assembly using statements like `"#if(@defined(XXXXX) ..."`

The **-dXXXX=YYYY** option tells HLA to define the symbol *XXXX* as a string **val** constant and give it the initial value *"YYYY"*.

-b:name

When compiling to an object file using the HLA back-engine (rather than a back-end assembler), this option specifies the name of the binary object file. By default, HLA uses the base name (before the ".hla" suffix) with a ".obj" (windows) or ".o" (*NIX) suffix. With the "-b:name" option you may specify a different name. Note that if you do not supply an ".obj" or ".o" suffix at the end of "-b:name" because HLA will automatically attach the suffix.

-e:name

By default, HLA creates an executable filename using the extension ".exe" (Windows) or without an extension (*NIX) and the *basename* of the first filename on the command line. You can use the **-e name** option to specify a different executable file name.

-x:name

Similar to the **-e:name** option, except there is no automatic ".exe" suffix applied under Windows. This lets you explicitly supply the suffix (e.g., ".dll" under Windows, or to force a ".exe" under *NIX).

-lXXXXX

The **-lXXXXX** option passes the text *XXXXX* on to the linker as a command line option. One common command to pass to the Microsoft linker is `"-lDEBUG"` that tells the linker to generate debugging information in the object file.

-m

The **-m** option tells the Microsoft linker or POLINK to produce a map file during the link phase. This is equivalent to the `"-lmap"` option. The *NIX version of HLA ignores this option.

-s

The `-s` option tells the HLA program to run only the `hlaparse` compiler to produce an assembly language source file; HLA will not run a back-end assembler or linker. As a result, HLA ignores any `".asm"` or `".obj"` filenames you supply on the command line. This option is useful if you wish to view the output of an HLA compilation in some other assembler's source format without producing any actual object code. This option must be used with the `"-source"` command-line option if you are using the HLA back engine and you wish to produce a human-readable source file of the HLABE output.

`-sym`

The `-sym` option dumps the symbol table after compiling each file with an HLA extension. This option is primarily intended for testing and debugging the HLA compiler; however, this information can be useful to the HLA programmer on occasion.

`-thread`

The `-thread` option tells HLA to generate thread-safe code (for the code it emits) and link in the thread-safe version of the HLA standard library. Specifying this command-line option sets the HLA `@thread` object to true, which you can test during the compilation of an HLA source file (that must behave differently for thread-safe versus non-thread-safe code).

`-test`

The `-test` option is intended for `hlaparse` testing and debugging purposes only. It causes the compiler to send all error messages to the standard output device rather than the standard error device. This allows the test code to redirect all errors to a text file for comparison against other files. This command also causes `HLAPARSE` to emit some extra comment information to the assembly language output file when producing output files for one of the back-end assemblers (other than the HLA back engine).

`-v`

The `-v` option (verbose) causes HLA to print additional information during compile to show the progress of the compilation. This option also prints certain statistics, such as the number of lines per second that HLA compiles.

`-w`

The `-w` option informs HLA that you are compiling a standard Windows (GUI) application rather than a console application. By default, HLA assumes that you are compiling a executable that will run from the command window. If you want to write a full Windows application, you will need to supply this option to tell HLA not to link the code for console operation. Obviously, this option doesn't apply to *NIX systems. The `"-w"` option tells HLA to invoke the linker using the command line option

```
-subsystem:windows  
rather than the default  
-subsystem:console
```

This provides a convenient mechanism for those who wish to create win32 GUI applications. Most likely, however, if you wish to create GUI applications, you will run the linker explicitly yourself (as this document will explain), so you'll probably not use the `"-w"` option very frequently. It's great for some short GUI demos, but larger GUI programs will probably not use this option. This option is only active if HLA compiles the program to an executable. If you compile the program to an OBJ or ASM file, HLA ignores this option.

If you want to develop Win32 GUI apps, look at Randy Hyde's book "Windows Programming in Assembly". This book provides the linker commands and makefiles for generation such applications (as well as describing how you actually write such code).

-p:path

During compilation, HLA produces several temporary files (that it doesn't delete, because they may be of interest to the HLA user). These files have a habit of cluttering up the current working directory. If you prefer, you can tell HLA to place these files in a temporary directory so they don't clutter up your working directory. One way to accomplish this is by using the "*-p:dirpath*" command line option. For example, the option "*-p:c:\hla\tmp*" tells HLA to put all temporary files (for the current assembly) into the "*c:\hla\tmp*" subdirectory (which must exist). Note that you can set also set the temporary directory using the hla "*hlatemp*" environment variable. The "*-p:dirpath*" option will override the environment variable (if it exists). See the description of the *hlatemp* environment variable for more details.

-obj:path

During compilation, HLA normally writes all object files to the current working directory. Some programmers have requested a way to specify a different directory for the *.OBJ* (*.o* under **NIX*) files that HLA produces. You can accomplish this using the "*-obj:dirpath*" command line option. The *dirpath* item has to be the path to a valid directory. HLA places all object files produced by the compiler and/or resource editor in this directory. Note that, unlike the *-p* option, there is no environment variable that lets you permanently set this path. You must specify the path on a compilation-by-compilation basis (use a makefile if you get tired of typing the path in on each compilation).

-level=h**-level=m****-level=l****-level=v**

The *-level* options enable or disable certain HLA language features. These command-line options are intended for use in programming courses where the instructor needs to batch compile dozens or even hundreds of student projects at one time. This allows the instructor to ensure that the students aren't using high-level control constructs that are inappropriate for that point in the course. For example, towards the end of a course, most instructors don't allow the use of various high-level control constructs; some instructors may never allow them. The "*-level*" command-line options will "turn off" various statements in the HLA language so that the HLA compiler will report an error if the student attempts to use them in a source file.

The default, "*-level=h*" (high) enables the entire HLA language.

The "*-level=m*" (medium level) disables high-level language control constructs, such as "if", "while", and "for" but still allows the use of high-level-like procedure calls in the HLA language. Medium-level assembly language also allows the use of exceptions using HLA's *try..except..endtry* and *raise* statements.

The "*-level=l*" (low-level assembly) disables all high-level control constructs other than the exception-handling statements and disables high-level-like procedure calls in HLA. This option also disables automatic stack frame generation and clean up in HLA procedures (that is, the programmer will be responsible for writing that code themselves).

The "*-level=v*" (very low-level assembly) option disables all high-level control constructs including exception handling. Only machine instructions (and user written macros) are legal in the source file. No high-level control constructs or high-level procedure calls are allowed.

-?

The *-?* option cause HLA to dump the list of command line options and immediately quit without further work.

Note that the command line options this document describes are for HLA v2.2 and later only. Earlier versions of HLA used a different command line set. See the documentation for the specific version you're using if you have questions.

-license

This command displays license information for the entire HLA system. Although the HLA source code written by Randall Hyde is all public domain, certain components of the HLA system, including the back-end assemblers, the linker, and the resource editor, may come from other sources. The "-license" command-line parameter lists license information about these other products.

7 HLA v2.x Language Reference Manual

7.1 HLA Language Elements

Starting with this chapter we begin discussing the HLA source language. HLA source files must contain only seven-bit ASCII characters. These are text files with each source line record containing a carriage return/line feed (Windows) or a just a line feed (*NIX) termination sequence (HLA is actually happy with either sequence, so text files are portable between Oses without change). White space consists of spaces, tabs, and newline sequences. Generally, HLA does not appreciate other control characters in the file and may generate an error if they appear in the source file.

7.2 Comments

HLA uses `"/"` to lead off single line comments. It uses `"/**"` to begin an indefinite length comment and it uses `**/"` to end an indefinite length comment. C/C++, Java, and Delphi users will be quite comfortable with this notation.

7.3 Special Symbols

The following characters are HLA lexical elements and have special meaning to HLA:

`* / + - () [] { } < > : ; , . = ? & | ^ ! @ !`

The following character pairs are HLA lexical elements and also have special meaning to HLA:

`&& || <= >= <> != == := .. << >> ## #()# #{ }#`

7.4 Reserved Words

Here are the HLA reserved words. You may not use any of these reserved words as HLA identifiers except as noted below (with respect to the `#id` and `#rw` operators). HLA reserved words are case insensitive. That is, "MOV" and "mov" (as well as any permutation with respect to case) both represent the HLA "mov" reserved word.

<code>#append</code>	<code>#asm</code>	<code>#closeread</code>	<code>#closewrite</code>
<code>#else</code>	<code>#elseif</code>	<code>#emit</code>	<code>#endasm</code>
<code>#endfor</code>	<code>#endif</code>	<code>#endmacro</code>	<code>#endmatch</code>
<code>#endregex</code>	<code>#endstring</code>	<code>#endtext</code>	<code>#endwhile</code>
<code>#error</code>	<code>#for</code>	<code>#id</code>	<code>#if</code>
<code>#include</code>	<code>#includeonce</code>	<code>#keyword</code>	<code>#linker</code>
<code>#macro</code>	<code>#match</code>	<code>#openread</code>	<code>#openwrite</code>
<code>#print</code>	<code>#regex</code>	<code>#return</code>	<code>#rw</code>
<code>#string</code>	<code>#system</code>	<code>#terminator</code>	<code>#text</code>
<code>#while</code>	<code>#write</code>	<code>@a</code>	<code>@abs</code>
<code>@abstract</code>	<code>@ae</code>	<code>@align</code>	<code>@alignstack</code>
<code>@arb</code>	<code>@arity</code>	<code>@at</code>	<code>@b</code>
<code>@baseptype</code>	<code>@basereg</code>	<code>@basetype</code>	<code>@be</code>
<code>@boolean</code>	<code>@bound</code>	<code>@byte</code>	<code>@c</code>

@cdecl	@ceil	@char	@class
@cos	@cset	@curdir	@curlex
@curobject	@curoffset	@date	@debughla
@defined	@delete	@dim	@display
@dword	@e	@elements	@elementsiz
@enter	@enumsiz	@env	@eos
@eval	@exactlynchar	@exactlyncset	@exactlynchar
@exactlyntomchar	@exactlyntomcset	@exactlyntomichar	@exceptions
@exp	@external	@extract	@fast
@filename	@firstnchar	@firstncset	@firstnchar
@floor	@forward	@fpureg	@frame
@g	@ge	@global	@here
@index	@insert	@int128	@int16
@int32	@int64	@int8	@into
@isalpha	@isalphanum	@isclass	@isconst
@isdigit	@IsExternal	@isfreg	@islower
@ismem	@isreg	@isreg16	@isreg32
@isreg8	@isspace	@istype	@isupper
		@label	
@isxdigit	@l	@lastobject	@le
@leave	@length	@lex	@linenumber
@localoffset	@localsyms	@log	@log10
@lowercase	@lword	@match	@match2
@matchchar	@matchcset	@matchichar	@matchid
@matchintconst	@matchistr	@matchiword	@matchnumericconst
@matchrealconst	@matchstr	@matchstrconst	@matchtoistr
@matchtostr	@matchword	@max	@min
@mmxreg	@na	@nae	@name
@nb	@nbe	@nc	@ne
@ng	@nge	@nl	@nle
@no	@noalignstack	@nodisplay	@noenter
@noframe	@noleave	@norlesschar	@norlesscset
@norlessichar	@normorechar	@normorecset	@normoreichar
@nostackalign	@nostorage	@np	@ns
@ntomchar	@ntomcset	@ntomichar	@nz
@o	@odd	@offset	@onechar
@onecset	@oneichar	@oneormorechar	@oneormorecset
@oneormoreichar	@oneormorews	@optstrings	@p
@parmoffset	@parms	@pascal	@pclass
@pe	@peekchar	@peekcset	@peekichar
@peekistr	@peekstr	@peekws	@po
@pointer	@pos	@ptype	@qword
@random	@randomize	@read	@real128
@real32	@real64	@real80	@reg
@reg16	@reg32	@reg8	@regex
@returns	@rindex	@s	@section
@sin	@size	@sort	@sqrt
@stackalign	@staticname	@stdcall	@strbrk
@string	@strset	@strspan	@substr

@system	@tab	@tan	@tbyte
@text	@thread	@time	@tokenize
@tostring	@trace	@trim	@type
@typename	@uns128	@uns16	@uns32
@uns64	@uns8	@uppercase	@uptochar
@uptocset	@uptochar	@uptoistr	@uptostr
@use	@volatile	@wchar	@word
@ws	@wsoreos	@wstheneos	@wstring
@xmmreg	@z	@zeroormorechar	@zeroormorecset
@zeroormoreichar	@zeroormorews	@zerooronechar	@zerooronecset
@zerooroneichar	@zstring	aaa	aad
aam	aas	abstract	adc
add	addpd	addps	addsd
addss	addsubpd	addsubps	ah
al	align	and	andnpd
andnps	andpd	andps	anyexception
arpl	ax	begin	bh
bl	boolean	bound	bp
break	breakif	bsf	bsr
bswap	bt	btc	btr
bts	bx	byte	call
case	cbw	cdq	ch
char	cl	class	clc
cld	cflush	cli	clts
cmc	cmova	cmovae	cmovb
cmovbe	cmovc	cmove	cmovg
cmovge	cmovl	cmovle	cmovna
cmovnae	cmovnb	cmovnbe	cmovnc
cmovne	cmovng	cmovnge	cmovnl
cmovnl	cmovno	cmovnp	cmovns
cmovnz	cmovo	cmovp	cmovpe
cmovpo	cmovs	cmovz	cmp
cmpeqpd	cmpeqps	cmpeqsd	cmpeqss
cmplepd	cmpleps	cmplepd	cmplless
cmpltpd	cmpltps	cmpltsd	cmpltss
cmpneqpd	cmpneqps	cmpneqsd	cmpneqss
cmpnlepd	cmpnleps	cmpnlesd	cmpnless
cmpnltpd	cmpnltps	cmpnltsd	cmpnlts
cmpordpd	cmpordps	cmpordsd	cmpordss
cmppd	cmpps	cmplib	cmplib
cmpss	cmpsw	cmpunordpd	cmpunordps
cmpunordsd	cmpunordss	cmpxchg	cmpxchg8b
comisd	comiss	const	continue
continueif	cpuid	cr0	cr1
cr2	cr3	cr4	cr5
cr6	cr7	cseg	cset
cvtdq2pd	cvtdq2ps	cvtpd2dq	cvtpd2pi
cvtpd2ps	cvtpi2pd	cvtpi2ps	cvtps2dq
cvtps2pd	cvtps2pi	cvtsd2si	cvtsd2ss

cvtsi2sd	cvtsi2ss	cvtss2sd	cvtss2si
cvttpd2dq	cvttpd2pi	cvttps2dq	cvttps2pi
cvttss2si	cvttss2si	cwd	cwde
cx	daa	das	dec
default	dh	di	div
divpd	divps	divsd	divss
dl	do	downto	dr0
dr1	dr2	dr3	dr4
dr5	dr6	dr7	dseg
dup	dword	dx	dx:ax
eax	ebp	ebx	ecx
edi	edx	edx:eax	else
elseif	emms	end	endclass
			endlabel
endconst	endfor	endif	endproc
endreadonly	endrecord	endstatic	endstorage
endswitch	endtry	endtype	endunion
endval	endvar	endwhile	enter
enum	eseg	esi	esp
exception	exit	exitif	external
f2xm1	fabs	fadd	faddp
fbld	fbstp	fchs	fclex
fcmova	fcmovae	fcmovb	fcmovbe
fcmove	fcmovna	fcmovnae	fcmovnb
fcmovnbe	fcmovne	fcmovnu	fcmovu
fcom	fcomi	fcomip	fcomp
fcompp	fcos	fdecstp	fdiv
fdivp	fdivr	fdivrp	felse
ffree	fiadd	ficom	ficomp
fidiv	fidivr	file	fimul
fincstp	finit	fist	fistp
fisttp	fisub	fisubr	fld
fld1	fldcw	fldenv	fldl2e
fldl2t	fldlg2	fldln2	fldpi
fldz	fmul	fmulp	fnclx
fninit	fnop	fnsave	fnstcw
fnstenv	fnstsw	for	foreach
forever	forward	fpatan	fprem
fprem1	fptan	frndint	frstor
fsave	fscale	fseg	fsin
fsincos	fsqrt	fst	fstcw
fstenv	fstp	fstsw	fsub
fsubp	fsubr	fsubrp	ftst
fucom	fucomi	fucomip	fucomp
fucompp	fwait	fxam	fxch
fxrstor	fxsave	fxtract	fyl2x
fyl2xp1	gseg	haddpd	haddps
hlt	hsubpd	hsubps	idiv
if	imod	imul	in

inc	inherits	insb	insd
insw	int	int128	int16
int32	int64	int8	intmul
into	invd	invlpg	iret
iretd	iterator	ja	jae
jb	jbe	jc	jcxz
je	jecxz	jf	jg
jge	jl	jle	jmp
jna	jnae	jnb	jnb
jnc	jne	jng	jnge
jnl	jnle	jno	jnp
jns	jnz	jo	jp
jpe	jpo	js	jt
jz	label	lahf	lar
lazy	lddqu	ldmxcsr	lds
lea	leave	les	lfence
lfs	lgdt	lgs	lidt
lldt	lmsw	lock.adc	lock.add
lock.and	lock.btc	lock.btr	lock.bts
lock.cmpxchg	lock.dec	lock.inc	lock.neg
lock.not	lock.or	lock.sbb	lock.sub
lock.xadd	lock.xchg	lock.xor	lodsb
lodsd	lodsw	loop	loope
loopne	loopnz	loopz	lsl
lss	ltreg	lword	maskmovdqu
maskmovq	maxpd	maxps	maxsd
maxss	method	mfence	minpd
minps	minsd	minss	mm0
mm1	mm2	mm3	mm4
mm5	mm6	mm7	mod
monitor	mov	movapd	movaps
movd	movddup	movdq2q	movdqa
movdqu	movhlps	movhpd	movhps
movlhps	movlpd	movlps	movmskpd
movmskps	movntdq	movnti	movntpd
movntps	movntq	movq	movq2dq
movsb	movsd	movshdup	movsldup
movss	movsw	movsx	movupd
movups	movzx	mul	mulpd
mulps	mulsd	mulss	mwait
name	namespace	neg	nop
not	null	or	orpd
orps	out	outsb	outsd
	overloads		
outsw	override	overrides	packssdw
packsswb	packuswb	paddb	paddd
paddq	paddsb	paddsw	paddusb
paddusw	paddw	pand	pandn
pause	pavgb	pavgw	pcmpeqb

pcmpeqd	pcmpeqw	pcmpgtb	pcmpgtd
pcmpgtw	pextrw	pinsrw	pmaddwd
pmaxsw	pmaxub	pminsw	pminub
pmovmskb	pmulhw	pmulhw	pmullw
pmuludq	pointer	pop	popa
popad	popf	popfd	por
prefetchnta	prefetcht0	prefetcht1	prefetcht2
proc	procedure	program	psadbw
pshufd	pshufhw	pshufw	pshufw
pslld	pslldq	psllq	psllw
psrad	psraw	psrld	psrldq
psrlq	psrlw	psubb	psubd
psubq	psubsb	psubsw	psubusb
psubusw	psubw	punpckhbw	punpckhdq
punpckhqdq	punpckhwd	punpcklbw	punpckldq
punpcklqdq	punpcklwd	push	pusha
pushad	pushd	pushf	pushfd
pushw	pxor	qword	raise
rcl	rcpps	rcpss	rcr
rdmsr	rdpmc	rdtsc	readonly
real128	real32	real64	real80
record	regex	rep.insb	rep.insd
rep.insw	rep.movsb	rep.movsd	rep.movsw
rep.outsb	rep.outsd	rep.outsw	rep.stosb
rep.stosd	rep.stosw	repe.cmpsb	repe.cmpsd
repe.cmpsw	repe.scasb	repe.scasd	repe.scasw
repeat	repne.cmpsb	repne.cmpsd	repne.cmpsw
repne.scasb	repne.scasd	repne.scasw	repnz.cmpsb
repnz.cmpsd	repnz.cmpsw	repnz.scasb	repnz.scasd
repnz.scasw	repz.cmpsb	repz.cmpsd	repz.cmpsw
repz.scasb	repz.scasd	repz.scasw	result
ret	returns	rol	ror
rsm	rsqrtps	rsqrtss	sahf
sal	sar	sbb	scasb
scasd	scasw	segment	seta
setae	setb	setbe	setc
sete	setg	setge	setl
setle	setna	setnae	setnb
setnbe	setnc	setne	setng
setnge	setnl	setnle	setno
setnp	setns	setnz	seto
setp	setpe	setpo	sets
setz	sfence	sgdt	shl
shld	shr	shrd	shufpd
shufps	si	sidt	sldt
smsw	sp	sqrtpd	sqrtps
sqrtsd	sqrtss	sseg	st0
st1	st2	st3	st4
st5	st6	st7	static

stc	std	sti	stmxcscr
storage	stosb	stosd	stosw
streg	string	sub	subpd
subps	subsd	subss	switch
sysenter	sysexit	tbyte	test
text	then	this	thunk
to	try	type	ucomisd
ucomiss	ud2	union	unit
unpckhpd	unpckhps	unpcklpd	unpcklps
unprotected	uns128	uns16	uns32
uns64	uns8	until	val
valres	var	verr	verw
vmt	wait	wbinvd	wchar
welse	while	word	wrmsr
wstring	xadd	xchg	xlat
xmm0	xmm1	xmm2	xmm3
xmm4	xmm5	xmm6	xmm7
xor	xorpd	xorps	zstring

Note that **@debughla** is also a reserved compiler symbol. However, this is intended for internal (HLA) debugging purposes only. When the compiler encounters this symbol, it immediately stops the compiler with an assertion failure. Obviously, you should never put this statement in your source code unless you're debugging HLA and you want to stop the compiler immediately after the compilation of some statement.

Because the set of HLA reserved words is changing frequently, a special feature was added to HLA to allow a programmer to "disable" HLA reserved words. This may allow an older program that uses new HLA reserved words as identifiers to continue working with only minor modifications to the HLA source code. The ability to disable certain HLA reserved words also allows you to create macros that override certain machine instructions.

All HLA reserved words take two forms: the standard, mutable, form (appearing in the table above) and a special immutable form that consists of a tilde character (~) followed by the reserved word. For example, 'mov' is the mutable form of the move instruction while '~mov' is the immutable form. By default, the immutable and mutable forms are equivalent when you begin an assembly. However, you can use the **#id** compile-time statement to convert the mutable form to an identifier and you can use the **#rw** compile-time statement to turn it back into a reserved word. Regardless of the state of the mutable form, the immutable form always behaves like the reserved word as far as HLA is concerned. Here's an example of the **#id** and **#rw** statements:

```
#id( mov ) //From this point forward, mov is an identifier, not a
reserved word
mov:
    ~mov( i, eax ); // Must use ~mov while mov is a reserved word!
    cmp( eax, 0 );
    jne mov;
#rw( mov ) // Okay, now mov is a reserved word again.
    mov( 0, eax );
```

Note that you can use the **#id** facility to disable certain instructions. For example, by default HLA handles almost all (32-bit flat model) instructions up through the latest Intel processors. If you want to write code for an earlier processor, you may want to disable instructions available only on later processors to help avoid their use. You can do this by placing the offending instructions in **#id** statements.

The `#rw` statement will not turn an arbitrary identifier into a reserved word. It will only revert a reserved word that was previously converted to an identifier back into a reserved word.

One use of the `#id` statement is to change the syntax of existing HLA instructions. For example, some x86 programmers are completely incapable of handling HLA's (and Gas') "source, dest" syntax and insist on using the original Intel "dest, source" syntax. This isn't a good reason for giving up on HLA because you can easily override HLA's syntax by using the `#id` statement and a set of macros. Consider the following example for the `mov` instruction:

```
#id( mov )
#macro mov( dest, source );
    ~mov( source, dest )
#endmacro
```

By creating an include file (let's calling "intel.hhf") with all the appropriate macros and `#id` statements, you can easily change HLA's syntax to take on a more "Intel" feel.

7.5 External Symbols and Assembler Reserved Words

HLA v2.x, in addition to directly producing object code, offers the option of producing an assembly language file during compilation and invoking an assembler such as MASM, FASM, NASM, or Gas to complete the compilation process. HLA automatically translates normal identifiers you declare in your program to benign identifiers in the assembly language program (in HLA v2.2 these identifiers typically took the form *original_name*_hla_xxxx where *original_name* is the original symbol and xxxx is a unique four-digit hexadecimal value). However, HLA does not translate **external** symbols, but preserves these names in the assembly language file it produces. Therefore, you must take care not to use external names that conflict with the underlying assembler's set of reserved words or that assembler will generate an error when it attempts to process HLA's output. Obviously, this is not an issue when directly producing object code with HLA (rather than producing an assembly language source file to be assembled by some other assembler).

For a list of assembler reserved words, please see the documentation for the back-end assembler you are using.

7.6 HLA Identifiers

HLA identifiers must begin with an alphabetic character or an underscore. After the first character, the identifier may contain alphanumeric and underscore symbols. There is no technical limit on identifier length in HLA, but you should avoid external symbols greater than about 32 characters in length since the assembler and linkers that process HLA identifiers may not be able to handle such symbols. Also note that if you are generating assembly language source output files, HLA may add some additional characters to the identifiers you use (typically something like `__HLA_xxxx` where "xxxx" is a 4-digit hexadecimal number) in order to prevent conflicts with the assembler's own reserved word set. As such, you may want to limit yourself to about 20-22 characters if you're using a back-end assembler that has limited identifier lengths.

HLA identifiers are always *case neutral*. This means that identifiers are case sensitive insofar as you must always spell an identifier exactly the same way (with respect to alphabetic case). However, you are not allowed to declare two identifiers whose only difference is alphabetic case.

Although technically legal in your program, do not use identifiers that begin and end with a single underscore. HLA reserves such identifiers for use by the compiler and the HLA standard library. If you declare such identifiers in your program, the possibility exists that you may interfere with HLA's or the HLA Standard Library's use of such a symbol.

By convention, HLA programmers use symbols beginning with two underscores to represent private fields in a class. Therefore, you should avoid such identifiers except when defining such private fields in your own classes.

7.7 External Identifiers

HLA lets you explicitly provide a string for external identifiers. External identifiers are not limited to the format for HLA identifiers. HLA allows any string constant to be used for an external

identifier. If you're using a back-end assembler, it is your responsibility to use only those characters that are legal in that assembler. Note that this feature lets you use symbols that are not legal in HLA but are legal in external code (e.g., Win32 APIs use the '@' character in identifiers and some non-HLA code may use HLA reserved words as identifiers). See the discussion of the **external** option in the chapters on *HLA Program Structure* and *HLA Procedures* for more details.

7.8 HLA Literal Constants

HLA supports literal numeric, string, character, character set, Boolean, array, record, and union constants. For more details on these HLA language elements, please see the chapters on *HLA Constants and Constant Expressions* and *HLA Data Types*.

8 HLA Data Types

8.1 Data Types in HLA

Unlike traditional x86 assemblers that tend to work only with bytes, words, double-words, quad-words, and long- (oct-) words, HLA provides a rich set of basic primitive types. This chapter discusses all the built-in and user-definable types that HLA supports.

8.2 Native (Primitive) Data Types in HLA

HLA provides the following basic primitive types:

boolean	One byte; zero represents false, one represents true (any non-zero value also represents true).
Enum	One, two, or four bytes (program selectable, default is one byte); user defined IDs with unique values.
Uns8	Unsigned values in the range 0..255.
Uns16	Unsigned integer values in the range 0..65535.
Uns32	Unsigned integer values in the range 0..4,204,967,295.
Uns64	Unsigned 64-bit integer.
Uns128	Unsigned 128-bit integer.
Byte	Generic eight-bit value.
Word	Generic 16-bit value.
DWord	Generic 32-bit value.
QWord	Generic 64-bit value.
TByte	Generic 80-bit value.
LWord	Generic 128-bit value.
Int8	Signed integer values in the range -128..+127.
Int16	Signed integer values in the range -32768..+32767.
Int32	Signed integer values in the range -2,147,483,648..+2,147,483,647.
Int64	Signed 64-bit integer values.
Int128	Signed 128-bit integer values.
Char	Character values.
WChar	Unicode character values.
Real32	32-bit floating-point values.
Real64	64-bit floating-point values.
Real80	80-bit floating-point values.
Real128	128-bit floating-point values (for SSE/2 instructions).
String	Dynamic length string constants. (Run-time implementation: four-byte pointer.)
ZString	Zero-terminated dynamic length strings (run-time implementation: four-byte pointer).
Unicode	Unicode strings.
CSet	A set of up to 128 different ASCII characters (16-byte bitmap).

Text	Similar to string, but text constants expand in-place (like #define in C/C++).
Thunk	A set of machine instructions to execute.

Often, it is convenient to discuss the types above in various groups. The HLA language reference manual will often use the following terms:

Ordinal:	boolean, enum, uns8, uns16, uns32, byte, word, dword, int8, int16, int32, char.
Unsigned:	uns8, uns16, uns32, byte, word, dword.
Signed:	int8, int16, int32, byte, word, dword.
Number:	uns8, uns16, uns32, int8, int16, int32, byte, word, dword
Numeric:	uns8, uns16, uns32, int8, int16, int32, word, dword, real32, real64, real80

8.2.1 Enumerated Data Types

HLA provides the ability to associate a list of identifiers with a user-defined type. Such types are known as enumerated data types (because HLA enumerates, or numbers, each of the identifiers in the list to give them a unique value). The syntax for an enumerated type declaration (in an HLA `type` section, see the description a little later) takes the following form:

```
typename : enum{ list_of_identifiers };
```

Here is a typical example:

```
type
  color_t :enum{ red, green, blue, magenta, yellow, cyan, black, white };
```

Internally, HLA treats enumerated types as though they were unsigned integer values (though **enum** types are not directly compatible with the unsigned types). HLA associates the value zero with the first identifier in the **enum** list and then attaches sequentially increasing values to the following identifiers in the list. For example, HLA will associate the following values with the **color_t** symbolic constants:

```
red      0
green    1
blue     2
magenta3
yellow   4
cyan     5
black    6
white    7
```

Because each enumerated constant in a given **enum** list is unique, you may compare these values, use them in computations, etc. Also note that, because of the way HLA assigns internal values to these constant names, you may compare objects in an enumerated list for less than and greater than in addition to equal or not equal.

Note that HLA uses zero as the internal representation for the first symbol of every **enum** list. HLA only guarantees that the values it associates with **enum** types are unique for a single type; it does not make this guarantee across different enumerated types (in fact, you're guaranteed that different **enum** types *do not* use unique values for their symbol sets). In the following example, HLA uses the value zero for both the internal representation of *const0* and *c0*. Likewise, HLA uses the value one for both *const1* and *c1*. And so on...

```
type
  enumType1 :enum{ const0, const1, const2 };
  enumType2 :enum( c0, c1, c2 );
```

Note that the enumerated constants you specify are not "private" to that particular type. That is, the constant names you supply in an enumerated data type list must be unique within the current scope (see the definition of identifier scope elsewhere in the HLA documentation). Therefore, the following is *not* legal:

```

type
  enumType1 :enum{ et1, et2, et3, et4 };
  enumType2 :enum{ et2, et2a, et2b, et2c }; //et2 is a duplicate symbol!

```

The problem here is that both type lists attempt to define the same symbol: *et2*. HLA reports an error when you attempt this.

One way to view the enumerated constant list is to think of it as a list of constants in an HLA **const** section (see the description of declaration sections a little later in this document), e.g.,

```

const
  red      : color_t := 0;
  green    : color_t := 1;
  blue     : color_t := 2;
  magenta: color_t := 3;
  yellow   : color_t := 4;
  cyan     : color_t := 5;
  black    : color_t := 6;
  white    : color_t := 7;

```

By default, HLA uses 8-bit values to represent enumerated data types. This means that you can represent up to 256 different symbols using an enumerated data type. This should prove sufficient for most applications. HLA provides a special "compile-time variable" that lets you change the size of an enumerated type from one to two or four bytes. All you have to do is assign the value two or four to this variable and HLA will automatically resize the storage for enumerated types to handle longer lists of objects. Example:

```

?@enumSize := 4; // Use dword size for enum types

type
  enumDword:enum{ d0, d1, d2, d3};

var
  ed :enumDword; // Reserves four bytes of storage

```

8.2.2 HLA Type Compatibility

HLA is unusual among assembly language insofar as it does some serious type checking on its operands. While the type checking isn't quite as "strong" as some high-level languages, HLA clearly does a lot more type checking than other assemblers, even those that purport to do type checking on operands (e.g., MASM). The use of strong type checking can help you locate logical errors in your code that would otherwise go unnoticed (except via a laborious and time consuming testing/debug session).

The downside to strong type checking is that experienced assembly programmers may become somewhat annoyed with HLA's reports that they are doing something wrong when, in fact, the programmer knows exactly what they are doing. There are two solutions to this problem: use type coercion (described a little bit later) or use the "untyped" types that reduce type checking to simply ensuring that the sizes of the operands match. However, before discussing how to override HLA's type checking system, it's probably a good idea to first describe how HLA uses data types.

Fundamentally, HLA divides the data types into classes based on the size of their underlying representation. Unless you explicitly override a type with a type coercion operation, attempting to mix object sizes in a memory or register operand will produce an error (in constant expressions, HLA is a bit more forgiving; it will automatically promote between certain types and adjust the type of the result accordingly). With most of HLA's data types, it's obvious what the size of the underlying representation is, because most HLA type names incorporate the size (in bits) in the type's name. For example, the **uns16** data type is a 16-bit (two-byte) type. Nevertheless, this rule isn't true for all data types, so it's a good idea to begin this discussion by looking at the underlying sizes of each of the HLA types.

```

8 bits:      boolean, byte, char, enum, int8, uns8
16 bits:     int16, uns16, wchar, word

```

32 bits: dword, int32, pointer types, real32, string, zstring, unicode, uns32
 64 bits: int64, qword, real64, uns64
 80 bits: real80, tbyte
 128 bits: cset, int128, lword, uns128, real128

The **byte**, **word**, **dword**, **qword**, **tbyte**, and **lword** types are somewhat special. These are known as *untyped data types*. They are directly compatible with any scalar, ordinal, data type that is the same size as the type in question. For example, a **byte** object is directly compatible with any object of type **boolean**, **byte**, **char**, **enum** (assuming **@enumSize** is 1), **int8**, or **uns8**. No special coercion is necessary when assigning a **byte** value to an object that has one of these other types; likewise, no special coercion operation is necessary when assigning a value of one of these other types to a **byte** object.

Note that **cset**, **real32**, **real64**, **real80**, and **real128** objects are not ordinal types. Therefore, you cannot directly mix these types with **lword**, **dword**, **qword**, **tbyte**, or **lword** objects without an explicit type coercion operation. Also, keep in mind that composite data types (see the next section) are not directly compatible with **bytes**, **words**, **dwords**, **qwords**, **tbytes**, and **lwords**, even if the composite data type has the same number of bytes (the only exception is the pointer data type, which is compatible with the **dword** type).

8.3 Composite Data Types

In addition to the primitive types above, HLA supports pointers, arrays, records (structures), unions, and classes of the primitive types (except for text objects).

8.4 Array Data Types

HLA allows you to create an array data type by specifying the number of array elements after a type name. Consider the following HLA type declaration that defines *intArray* to be an array of int32 objects:

```
type intArray : int32[ 16 ];
```

The "[16]" component tells HLA that this type has 16 four-byte integers. HLA arrays use a zero-based index, so the first element is always element zero. The index of the last element, in this example, is 15 (total of 16 elements with indices 0..15).

HLA also supports multidimensional arrays. You can specify multidimensional arrays by providing a list of indices inside the square brackets, e.g.,

```
type intArray4x4 : int32[ 4, 4 ];  
type intArray2x2x4 : int32[ 2,2,4 ];
```

The mechanism for accessing array elements differs depending upon whether you are accessing compile-time array constants or run-time array variables. A complete discussion of this will appear in later sections.

8.5 Union Data Types

HLA implements the discriminate union type using the **union..endunion** reserved words. The following HLA type declaration demonstrates a union declaration:

```
type  
  allInts:  
    union  
      i8:    int8;  
      i16:   int16;  
      i32:   int32;  
    endunion;
```


All fields in a union have the same starting address in memory. The size of a union object is the size of the largest field in the union. The fields of a union may have any type that is legal in a variable declaration section (see the discussion of the **var** section in the chapter on *HLA Program Structure* for more details).

Given a union object, say *i* of type *allInts*, you access the fields of the union using the familiar dot-notation. The following 80x86 **mov** instructions demonstrate how to access each of the fields of the *i* variable:

```
mov( i.i8, al );
mov( i.i16, ax );
mov( i.i32, eax );
```

Unions also support a special field type known as an anonymous record (see the next section for a description of records). The syntax for an anonymous record in a union is the following:

type

```
unionWrecord:
  union
    u1Field: byte;
    u2Field: word;
    u3Field: dword;
  record
    u4Field: byte[2];
    u5Field: word[3];
  endrecord;
  u6Field: byte;
endunion;
```

Fields appearing within the anonymous record do not necessarily start at offset zero in the data structure. In the example above, *u4Field* starts at offset zero while *u5Field* immediately follows it two bytes later. The fields in the union outside the anonymous record all start at offset zero. If the size of the anonymous record is larger than any other field in the union, then the record's size determines the size of the union. This is true for the example above, so the union's size is 16 bytes since the anonymous record consumes 16 bytes.

8.6 Record Data Type¹s

HLA's records allow programmers to create data types whose fields can be different types. The following HLA type declaration defines a simple record with four fields:

type

```
Planet:
  record

    x:      int32;
    y:      int32;
    z:      int32;
    density: real64;

  endrecord;
```

Objects of type *Planet* will consume 20 bytes of storage at run-time.

1. For C/C++ programmers: an HLA record is similar to a C struct. In language design terminology, a record is often referred to as a "cartesian product."

The fields of a record may be of any legal HLA data type including other composite data types. Like unions, anything that is legal in a **var** section is a legal field of a **record**. As for unions, you use the dot-notation to access fields of a **record** object.

In addition to the **var**-like declarations, you may also declare anonymous unions within a record. An anonymous union is a **union** declaration without a fieldname associated with the **union**, e.g.,

```
type
  DemoAU:
    record
      x:real32;
      union
        u1:int32;
        r1:real32;
      endunion;
      y:real32;
    endrecord;
```

In this example, *x*, *u1*, *r1*, and *y* are all fields of *DemoAU*. To access the fields of a variable *D* of type *DemoAU*, you would use the following names: *D.x*, *D.u1*, *D.r1*, and *D.y*. Note that *D.u1* and *D.r1* share the same memory locations at run-time, while *D.x* and *D.y* have unique addresses associated with them.

Record types may *inherit fields* from other record types. Consider the following two HLA type declarations:

```
type
  Pt2D:
    record

      x: int32;
      y: int32;

    endrecord;

  Pt3D:
    record inherits( Pt2D )

      z: int32;

    endrecord;
```

In this example, *Pt3D* inherits all the fields from the *Pt2D* type. The **inherits** keyword tells HLA to copy all the fields from the specified record (*Pt2D* in this example) to the beginning of the current record declaration (*Pt3D* in this example). Therefore, the declaration of *Pt3D* above is equivalent to:

```
Pt3D:
  record

    x: int32;
    y: int32;
    z: int32;

  endrecord;
```

In some special situations, you may want to override a field from a previous field declaration. For example, consider the following record declarations:

```
BaseRecord:
  record
    a: uns32;
    b: uns32;
  endrecord;

DerivedRecord:
  record inherits( BaseRecord )
    b: boolean; // New definition for b!
    c: char;
  endrecord;
```

Normally, HLA will report a "duplicate" symbol error when attempting to compile the declaration for *DerivedRecord* since the *b* field is already defined via the "inherits(BaseRecord)" option. However, in certain cases it's quite possible that the programmer wishes to make the original field inaccessible in the derived class by using the same name. That is, perhaps the programmer intends to actually create the following record:

```
DerivedRecord:
  record
    a: uns32; // Derived from BaseRecord
    b: uns32; // Derived from BaseRecord, but inaccessible here.
    b: boolean; // New definition for b!
    c: char;
  endrecord;
```

HLA allows a programmer explicitly override the definition of a particular field by using the **overrides** keyword before the field they wish to override. While the previous declarations for *DerivedRecord* produce errors, the following is acceptable to HLA:

```
BaseRecord:
  record
    a: uns32;
    b: uns32;
  endrecord;

DerivedRecord:
  record inherits( BaseRecord )
    overrides b: boolean; // New definition for b!
    c: char;
  endrecord;
```

Normally, HLA aligns each field on the next available byte offset in a record. If you wish to align fields within a record on some other boundary, you may use the **align** directive to achieve this. Consider the following record declaration as an example:

```
type
  AlignedRecord:
  record
    b :boolean; // Offset 0
    c :char; // Offset 1
    align(4);
    d :dword; // Offset 4
    e :byte; // Offset 8
```

```

        w :word;           // Offset 9
        f :byte;          // Offset 11
    endrecord;

```

Note that field *d* is aligned at a four-byte offset while *w* is not aligned. We can correct this problem by sticking another **align** directive in this record:

```

type
    AlignedRecord2 :
        record
            b :boolean;    // Offset 0
            c :char;       // Offset 1
            align(4);
            d :dword;      // Offset 4
            e :byte;       // Offset 8
            align(2);
            w :word;       // Offset 10
            f :byte;       // Offset 12
        endrecord;

```

Be aware of the fact that the **align** directive in a **record** only aligns fields in memory if the record object itself is aligned on an appropriate boundary. For example, if an object of type *AlignedRecord2* appears in memory at an odd address, then the *d* and *w* fields will also be misaligned (that is, they will appear at odd addresses in memory). Therefore, you must ensure appropriate alignment of any record variable whose fields you're assuming are aligned.

Note that the *AlignedRecord2* type consumes 13 bytes. This means that if you create an array of *AlignedRecord2* objects, every other element will be aligned on an odd address and three out of four elements will not be double-word aligned (so the *d* field will not be aligned on a four-byte boundary in memory). If you are expecting fields in a record to be aligned on a certain byte boundary, then the size of the record must be an even multiple of that alignment factor if you have arrays of the record. This means that you must pad the record with extra bytes at the end to ensure proper alignment. For the *AlignedRecord2* example, we need to pad the record with three bytes so that the size is an even multiple of four bytes. This is easily achieved by using an **align** directive as the last declaration in the record:

```

type
    AlignedRecord2 :
        record
            b :boolean;    // Offset 0
            c :char;       // Offset 1
            align(4);
            d :dword;      // Offset 4
            e :byte;       // Offset 8
            align(2);
            w :word;       // Offset 10
            f :byte;       // Offset 12
            align(4)       // Ensures we're padded to a multiple of four
        bytes.
        endrecord;

```

Note that you can only use values that are integral powers of two in the **align** directive and the value must be 16 or less.

If you want to ensure that all fields are appropriately aligned on some boundary within the record, but you don't want to have to manually insert **align** directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```

type
    alignedRecord3 : record[4]
        << Set of fields >>
    endrecord;

```

The "[4]" immediately following the **record** reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object's size (and the size of the objects preceding the field). HLA allows any integer expression that produces a value that is a power of two in the range 1..16 inside these parentheses. If you specify the value 1 (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than 1, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element's size.

Note that if you set the record alignment using this syntactical form, any **align** directive you supply in the record may not produce the desired results. When HLA sees an **align** directive in a record that is using field alignment, HLA will first align the current offset to the value specified by **align** and then align the next field's offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
  alignedRecord4 : record[4]
    a  :byte;
    b  :byte;
    c  :record[8]
      d :byte;
      e :byte;
    endrecord;
    f  :byte;
    g  :byte;
  endrecord;
```

In this example, HLA aligns fields *a*, *b*, *f*, and *g* on double-word boundaries, it aligns *d* and *e* (within *c*) on 8-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if *c* turns out to be aligned on some boundary other than an 8-byte boundary, then *d* and *e* will not actually be on 8-byte boundaries; they will, however be on 8-byte boundaries relative to the start of *c*.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
  recordname : record[maximum : minimum]
    << fields >>
  endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object's size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object's size. If the object's size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
  r: record[ 4:1 ];
    a  :byte;          // offset 0
    b  :word;         // offset 2
    c  :byte;         // offset 4
    d  :dword; [2] // offset 8
    e  :byte;         // offset 16
    f  :byte;         // offset 17
    g  :qword;       // offset 20
  endrecord;
```

Note that HLA aligns *g* on a double-word boundary (not quad-word, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the *f* field to be aligned on an odd boundary (because it's a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field within the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA sophisticated record alignment facilities let you specify record field alignments that match that used by most major high-level language compilers. This lets you easily access data types used in those HLLs without resorting to inserting lots of `ALIGN` directives inside the record.

When declaring record variables in a **var**, **static**, **readonly**, or **storage** declaration section, HLA associates the offset zero with the first field of a record. Each additional field in the record is assigned an offset corresponding to the sum of the sizes of all the prior fields. So in the following example, *x* would have the offset zero, *y* would have the offset four, and *z* would have the offset eight.

```
Pt3D:
    record

        x: int32;
        y: int32;
        z: int32;

    endrecord;
```

If you would like to specify a different starting offset, you can use the following syntax for a record declaration:

```
Pt3D:
    record := 4;

        x: int32;
        y: int32;
        z: int32;

    endrecord;
```

The constant expression specified after the assignment operator ("`:=`") specifies the starting offset of the first field in the record. In this example *x*, *y*, and *z* will have the offsets 4, 8, and 12, respectively.

Warning: setting the starting offset in this manner does not add padding bytes to the record. This record is still a 12-byte object. If you declare variables using a record declared in this fashion, you may run into problems because the field offsets do not match the actual offsets in memory. This option is intended primarily for mapping records to pre-existing data structures in memory. Only advanced assembly language programmers should use this option.

8.7 Pointer Types

HLA allows you to declare a pointer to some other type using syntax like the following:

```
pointer to base_type
```

The following example demonstrates how to create a pointer to a 32-bit integer within the type declaration section:

```
type pi32: pointer to int32;
```

HLA pointers are always 32-bit (near32) pointers.

HLA v1.x allowed you to define pointers to existing procedures using syntax like the following:

```
procedure someProc( parameter_list );
<< procedure options, followed by @external, @forward, or procedure body>>
.
.
.
type
  p : pointer to procedure someProc;
```

However, this pointer syntax has been deprecated as of HLA v2.0 and this syntax will disappear sometime in HLA v2.x. The modern way to declare pointers that are compatible with a particular procedure is to use the "new style" procedure declarations in the HLA proc section. This is done as follows:

```
type
  p : procedure( parameter_list );
.
.
.
proc
  someProc:p {optional procedure options};
.
.
.
```

See the HLA reference manual chapter on *HLA Procedures* for more details about the **proc** section.

Note that HLA provides the reserved word **null** (or **NULL**, reserved words are case insensitive) to represent the nil pointer. HLA replaces **NULL** with the value zero. The **NULL** pointer is compatible with any pointer type (including strings, which are pointers).

8.8 Thunks

A "thunk" is an 8-byte variable that contains a pointer to a piece of code to execute and an execution environment pointer (i.e., a pointer to an activation record). The code associated with a thunk is, essentially, a small procedure that uses the activation record of the surrounding code rather than creating its own activation record. HLA uses thunks to implement the iterator "yield" statement as well as pass by name and pass by lazy evaluation parameters. In addition to these two uses of thunks, HLA allows you to declare your own thunk objects and use them for any purpose you desire. To declare a **thunk** variable is easy, just use a declaration like the following in a **var**, **static**, **readonly**, or **storage** section:

```
thunkVar: thunk;
```

This declaration reserves eight bytes of storage. The first double-word holds the address of the code to execute, the second double-word holds a pointer to the activation record to load into EBP when the thunk executes.

Of course, like almost any pointer variable, declaring a **thunk** variable is the easy part; the hard part is making sure the **thunk** variable is initialized before attempting to call the thunk. While you could manually load the address of some code and the frame pointer value into a **thunk** variable, HLA provides a better syntax for initializing thunks with small code fragments: the **thunk** statement. The **thunk** statement uses the following syntax:

```
thunk thunkVar := #{ sequence_of_statements }#;
```

Consider the following example:

```

program ThunkDemo;
#include( "stdio.hhf" );

procedure proc1;
var
    i:      int32;
    p1Thunk: thunk;

    procedure proc2( t:thunk );
    var
        i:int32;
    begin proc2;

        mov( 25, i );
        t();
        stdout.put( "Inside proc2, i=", i, nl );

    end proc2;

begin proc1;

    thunk p1Thunk := #{ mov( 0, i ); }#;

    mov( 1, i );
    proc2( p1Thunk );
    stdout.put( "i=", i, nl );

end proc1;

begin ThunkDemo;

    proc1();

end ThunkDemo;

```

In this example, *proc1* has two local variables, *i* and *p1Thunk*. The **thunk** statement initializes the *p1Thunk* variable with the address of some code that moves a zero into the *i* variable. The **thunk** statement also initializes *p1Thunk* with a pointer to the current activation record (that is, a pointer to *proc1*'s activation record). Then *proc1* calls *proc2* passing *p1Thunk* as a parameter.

The *proc2* routine has its own local variable named *i*. Of course, this is a different variable from the *i* in *proc1*. *Proc2* begins by setting its variable *i* to the value 25. Then *proc2* invokes the thunk (passed to it as a parameter). This thunk sets the variable *i* to zero. However, because the thunk uses the current activation record when the **thunk** statement was executed, this statement sets *proc1*'s *i* variable to zero rather than *proc2*'s *i* variable. This program produces the following output:

```

Inside proc2, i=25
i=0

```

Although you probably won't use thunks that often, they are quite nice for deferred execution. This is especially useful in AI (Artificial Intelligence) programs.

8.9 Class Types

Classes and object-oriented programming are the subject of a different HLA Reference Manual Document. See the chapter on *HLA Classes* for more details.

8.10 Regular Expression Types

The HLA compile-time language supports a special data type known as a "compiled regular expression". Please see the section on regular expression macros in the chapter on the *HLA Compile-Time Language* for more details on this data type.

9 HLA Literal Constants and Constant Expressions

9.1 HLA Literal Constants

Literal constants are those language elements that we normally think of as non-symbolic constant objects. HLA supports a wide variety of literal constants. The following sections describe those constants.

9.1.1 Numeric Constants

HLA lets you specify several different types of numeric constants.

9.1.1.1 Decimal Constants

The first and last characters of a decimal integer constant must be decimal digits (0..9). Interior positions may contain decimal digits and underscores. The purpose of the underscore is to provide a better presentation for large decimal values (i.e., use the underscore in place of a comma in large values). Example: `1_234_265`.

Note: Technically, HLA does not allow negative literal integer constants. However, you can use the unary "-" operator to negate a value, so you'd never notice this omission (e.g., `-123` is legal, it consists of the unary negation operator followed by a positive decimal literal constant). Therefore, HLA always returns type `unsXX` for all literal decimal constants. Also, note that HLA always uses a minimum size of `uns32` for literal decimal constants. If you absolutely, positively, want a literal constant to be treated as some other type, use one of the compile-time type coercion functions to change the type (e.g., `uns8(1)`, `word(2)`, or `int16(3)`). Generally, the type that HLA uses for the object is irrelevant since HLA will automatically promote a value to a larger or smaller type as appropriate.

Here are the following ranges for the various HLA unsigned data types:

<code>uns8:</code>	<code>0..255</code>
<code>uns16:</code>	<code>0..65,535</code>
<code>uns32:</code>	<code>0..4,294,967,295</code>
<code>uns64:</code>	<code>0..18,446,744,073,709,551,615</code>
<code>uns128:</code>	<code>0..340,282,366,920,938,463,463,374,607,431,768,211,455</code>

9.1.1.2 Hexadecimal Constants

Hexadecimal literal constants must begin with a dollar sign ("\$\$") followed by a hexadecimal digit and must end with a hexadecimal digit (0..9, A..F, or a..f). Interior positions may contain hexadecimal digits or underscores. Hexadecimal constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., `$$1A_2F34_5438`.

If the constant fits into 32 bits or less, HLA always returns the `dword` type for a hexadecimal constant. For larger values, HLA will automatically use the `qword` or `lword` type, as appropriate. If you would like the hexadecimal value to have a different type, use one of the HLA compile-time type coercion functions to change the type (e.g., `byte($12)` or `qword($54)`).

Here are the following ranges for the various HLA hexadecimal data types:

<code>uns8:</code>	<code>0..\$FF</code>
<code>uns16:</code>	<code>0..\$FFFF</code>
<code>uns32:</code>	<code>0..\$FFFF_FFFF</code>
<code>uns64:</code>	<code>0..\$FFFF_FFFF_FFFF_FFFF</code>
<code>uns128:</code>	<code>0..\$FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF_FFFF</code>

9.1.1.3 Binary Constants

Binary literal constants begin with a percent sign ("%") followed by at least one binary digit (0/1) and they must end with a binary digit. Interior positions may contain binary digits or underscore

characters. Binary constants are easiest to read if each group of four digits (starting from the least significant digit) is separated from the others by an underscore. E.g., `%10_1111_1010`.

Like hexadecimal constants, HLA always associates the type `dword` with a "raw" binary constant; it will use the `qword` or `lword` type if the value is greater than 32 bits or 64 bits (respectively). If you want HLA to use a different type, use one of the compile-time type coercion functions to achieve this.

Obviously, binary constants may have as many binary digits as there are bits in the underlying type. This document will not attempt to write out the maximum binary literal constant!

9.1.1.4 Numeric Set Constants

HLA provides a special numeric constant form that lets you specify a numeric value by the bit positions containing ones. This corresponds to a *powerset* of integer values in the range 0..31. These constants take the following form:

```
@{ comma_separated_list_of_digits }
```

The *comma_separate_list_of_digits* can be empty (signifying no set bits, i.e., the value zero), a single digit, or a set of digits separated by commas. Here are some examples:

```
@{
@{8}
@{1,2,8,24}
```

The corresponding numeric constant is given the type `dword` and is assigned the value that has ones in all the specified bit positions. For example, "`@{8}`" is equal to 256 since this value has a single set bit in bit position eight. Note that "`@{0}`" equals one, not zero (because the value one has a single set bit in position zero).

9.1.1.5 Real (Floating-Point) Constants

Floating-point (real) literal constants always begin with a decimal digit (never just a decimal point). A string of one or more decimal digits may be optionally followed by a decimal point and zero or more decimal digits (the fractional part). After the optional fractional part, a floating-point number may be followed by "e" or "E", a sign ("+" or "-"), and a string of one or more decimal digits (the exponent part). Underscores may appear between two adjacent digits in the floating-point number; their presence is intended to substitute for commas found in real-world decimal numbers.

Examples:

```
1.2
2.345e-2
0.5
1.2e4
2.3e+5
1_234_567.0
```

Literal real constants are always 80 bits and have the default type `real80`. If you wish to specify `real32` or `real64` literal constants, then use the `real32` or `real64` compile-time coercion functions to convert the values, e.g., `real32(3.14159)`. During compile time, it's rare that you'd want to use one of the smaller types since they are less accurate at representing floating-point values (although this might be precisely why you decide to use the smaller real type, so the accuracy matches the computations you're doing at run-time).

The range of `real32` constants is approximately $10^{\pm 38}$ with $6\frac{1}{2}$ digits of precision; the range of `real64` values is approximately $10^{\pm 308}$ with approximately $14\frac{1}{2}$ digits of precision, and the range of `real80` constants is approximately $10^{\pm 4096}$ with about 18 digits of precision.

9.1.2 Boolean Constants

Boolean constants consist of the two predefined identifiers `true` and `false`. Note that your program may redefine these identifiers, but doing so is incredibly bad programming style. Since these are actual identifiers in the symbol table (and not reserved words), you must spell these identifiers in all lower case or HLA will complain (unlike reserved words that are case insensitive).

Internally, HLA represents **true** with one and **false** with zero. In fact, HLA's compile-time boolean operations only look at bit #0 of the boolean value (and always clear the other bits). HLA compile-time statements that expect a boolean expression do not use zero/not zero like C/C++ and a few other languages. Such expressions must have a boolean type with the values **true/false**; you cannot supply an integer expression and rely on zero/not zero evaluation as in C/C++ or BASIC.

9.1.3 Character Constants

Character literals generally consist of a single (graphic) character surrounded by apostrophes. To represent the apostrophe character use four apostrophes, e.g., `''''`.

Another way to specify a character constant is by typing the `"#"` symbol followed by a numeric literal constant (decimal, hexadecimal, or binary). Examples: `#13`, `#$D`, `##%1101`.

9.1.4 Unicode Character Constants

Unicode character constants are 16-bit values. HLA provides limited support for Unicode literal constants. HLA supports the UTF/7 code point (character set), which is just the standard seven-bit ASCII character set and nine high-order zero bits. To specify a 16-bit literal Unicode constant simply prefix a standard ASCII literal constant with a `'u'` or `'U'`, e.g.,

```
u'A' - UTF/7 character constant for 'A'
```

Note that UTF/7 constants are simply the ASCII character codes zero extended to 16 bits.

HLA provides a second syntax for Unicode character constants that lets you enter values whose character codes are outside the range \$20..\$7E. You can specify a Unicode character constant by its numeric value using the `'u#nnnn'` constant form. This form lets you specify a 16-bit value following the `'#'` in either binary, decimal, or hexadecimal form, e.g.,

```
u#1233
u#$60F
u%100100101001
```

9.1.5 String Constants

String literal constants consist of a sequence of (graphic) characters surrounded by quotes. To embed a quote within a string, insert a pair of quotes into the string, e.g., `"He said ""This"" to me."`

If two string literal constants are adjacent in a source file (with nothing but whitespace between them), then HLA will concatenate the two strings and present them to the parser as a single string. Furthermore, if a character constant is adjacent to a string, HLA will concatenate the character and string to form a single string object. This is useful, for example, when you need to embed control characters into a string, e.g.,

```
"This is the first line" #d #a "This is the second line" #d #a
```

HLA treats the above as a single string with a Windows newline sequence (CR/LF) at the end of each of the two lines of text.

9.1.6 Unicode String Constants

HLA lets you specify Unicode string literals by prefixing a standard string constant with a `'u'` or a `'U'`. Such string constants use the UTF/7 character set (that is, the standard ASCII character set) but reserve 16 bits for each character in the string. Note that the high order nine bits of each character in the string will contain zero.

As this was being written, there is no support for Unicode strings in the HLA Standard Library, though support for Unicode string functions should appear shortly (note that Windows' programmers can call the Unicode string functions that are part of the Windows' API).

9.1.7 Character Set Constants

A character set literal constant consists of several comma delimited character set expressions within a pair of braces. The character set expressions can either be individual character values or a pair of character values separated by an ellipse (`".."`). If an individual character expression appears

within the character set, then that character is a member of the set; if a pair of character expressions, separated by an ellipse, appears within a character set literal, then all characters between the first such expression and the second expression are members of the set. As a convenience, if a string constant appears between the braces, HLA will take the union of all the characters in that string and add those characters to the character set.

Examples:

```
{'a','b','c'} // a, b, and c.
{'a'..'c'}    // a, b, and c.
{'A'..'Z','a'..'z'} // Alphabetic characters.
{"cset"}      // The character set 'c', 'e', 's', and 't'.
{' ','#d,#a,#9'} // Whitespace (space, return, linefeed, tab).
```

HLA character sets are currently limited to holding characters from the 128-character ASCII character set. In the future, HLA may support an **xcset** type (supporting 256 elements) or even **wcset** (wide character sets), but that support does not currently exist.

9.2 Structured Constants

Structured constants are those whose data type is not a scalar. The structured constant types include array constants, record constants, union constants, and pointer constants.

9.2.1 Array Constants

HLA lets you specify an array literal constant by enclosing a set of values within a pair of square brackets. Since array elements must be homogenous, all elements in an array literal constant must be the same type or conformable to the same type. Examples:

```
[ 1, 2, 3, 4, 9, 17 ]
[ 'a', 'A', 'b', 'B' ]
[ "hello", "world" ]
```

Note that each item in the list of values can actually be a constant expression, not a simple literal value.

HLA array constants are always one-dimensional. This, however, is not a limitation because if you attempt to use array constants in a constant expression, the only thing that HLA checks is the total number of elements. Therefore, an array constant with eight integers can be assigned to any of the following arrays:

```
const
a8:      int32[8]      := [1,2,3,4,5,6,7,8];
a2x4:    int32[2,4]    := [1,2,3,4,5,6,7,8];
a2x2x2: int32[2,2,2] := [1,2,3,4,5,6,7,8];
```

Although HLA doesn't support the notation of a multi-dimensional array constant, HLA does allow you to include an array constant as one of the elements in an array constant. If an array constant appears as a list item within some other array constant, then HLA expands the interior constant in place, lengthening the list of items in the enclosing list. E.g., the following three array constants are equivalent:

```
[ [1,2,3,4], [5,6,7,8] ]
[ [ [1,2], [3,4] ], [ [5,6], [7,8] ] ]
[1,2,3,4,5,6,7,8]
```

Although the three array constants are identical, as far as HLA is concerned, you might want to use these three different forms to suggest the shape of the array in an actual declaration, e.g.,

```

const
  a8:      int32[8]      := [1,2,3,4,5,6,7,8];
  a2x4:    int32[2,4]    := [ [1,2,3,4], [5,6,7,8] ];
  a2x2x2:int32[2,2,2]:= [[ [1,2], [3,4] ], [ [5,6], [7,8] ]];

```

Also note that symbol array constants, not just literal array constants, may appear in a literal array constant. For example, the following literal array constant creates a nine-element array holding the values one through nine at indexes zero through eight:

```
const Nine:int32[ 9 ]:= [ a8, 9 ];
```

This assumes, of course, that *a8* was previously declared as above. Since HLA "flattens" all array constants, you could have substituted *a2x4* or *ax2x2x* for *a8* in the example above and obtained identical results.

You may also create an array constant using the HLA **dup** operator. This operator uses the following syntax:

```
expression DUP [expression_to_replicate]
```

Where *expression* is an integer expression and *expression_to_replicate* is some expression, possibly an array constant. HLA generates an array constant by repeating the values in the *expression_to_replicate* the number of times specified by the expression. (Note: this does not create an array with *expression* elements unless the *expression_to_replicate* contains only a single value; it creates an array whose element count is *expression* times the number of items in the *expression_to_replicate*). Examples:

```

10 dup [1]  -- equivalent to [1,1,1,1,1,1,1,1,1,1]
5 dup [1,2] -- equivalent to [1,2,1,2,1,2,1,2,1,2]

```

Please note that HLA does not allow class constants, so class objects may not appear in array constants. In addition, HLA does not allow generic pointer constants; only certain types of pointer constants are legal. See the discussion of pointer constants for more details.

9.2.2 Record Constants

HLA supports record constants using a syntax very similar to array constants. You enclose a comma-separated list of values for each field in a pair of square brackets. To differentiate array and record constants, the name of the record type and a colon must precede the opening square bracket, e.g.,

```

type
  Planet:
    record
      x :int32;
      y :int32;
      z :int32;
      density:real64;
    endrecord;

const
  p :Planet := Planet:[ 1, 12, 34, 1.96 ]

```

HLA associates the items in the list with the fields as they appear in the original record declaration. In this example, the values 1, 12, 34, and 1.96 are associated with fields *x*, *y*, *z*, and *density*, respectively. Of course, the types of the individual constants must match (or be conformable to) the types of the individual fields.

Note that you may not create a record constant for a particular record type if that record includes data types that cannot have compile-time constants associated with them. For example, if a field of a record is a class object, you cannot create a record constant for that type since you cannot create class constants.

9.2.3 Union Constants

Union constants allow you to initialize static union data structures in memory as well as initialize union fields of other data structures (including anonymous union fields in records). There are some important differences between HLA compile-time union constants and HLA run-time unions (as well as between the HLA run-time union constants and unions in other languages). Therefore, it's a good idea to begin the discussion of HLA's union constants with a description of these differences.

There are a couple of different reasons people use unions in a program. The original reason was to share a sequence of memory locations between various fields whose access is mutually exclusive. When using a union in this manner, one never reads the data from a field unless they've previously written data to that field and there are no intervening writes to other fields. The HLA compile-time language fully (and only) supports this use of union objects.

A second reason people use unions (especially in high-level languages) is to provide aliases to a given memory location; in particular, aliases whose data types are different. In this mode, a programmer might write a value to one field and then read that data using a different field (in order to access that data's bit representation as a different type). *HLA does not support this type of access to union constants.* The reason is quite simple: internally, HLA uses a special "variant" data type to represent all possible constant types. Whenever you create a union constant, HLA lets you provide a value for a single data field. From that point forward, HLA effectively treats the union constant as a scalar object whose type is the same as the field you've initialized; access to the other fields through the union constant is no longer possible. Therefore, you cannot use HLA compile-time constants to do type coercion; nor is there any need to since HLA provides a set of type coercion operators like `@byte`, `@word`, `@dword`, `@int8`, etc. As noted above, the main purpose for providing HLA union constants is to allow you to initialize static **union** variables; since you can only store one value into a memory location at a time, union constants only need to be able to represent a single field of the union at one time. Of course, at run time, you may access any field of the static union object you've created; but at compile-time you may only access the single field associated with a union constant.

An HLA literal union constant takes the following form:

```
typename.fieldname: [ constant_expression ]
```

The *typename* component above must be the name of a previously declared HLA union data type (i.e., a union type you've created in the type section). The *fieldname* component must be the name of a field within that union type. The *constant_expression* component must be a constant value (expression) whose type is the same as, or is automatically coercible to, the type of the *fieldname* field. Here is a complete example:

```
type
  u:union
    b:byte;
    w:word;
    d:dword;
    q:qword;
  endunion;

static
  uVar   :u      := u.w:[$1234];
```

The declaration for *uVar* initializes the first two bytes of this object in memory with the value \$1234. Note that *uVar* is actually eight bytes long; HLA automatically zeros any unused bytes when initializing a static memory object with a union constant.

Note that you may place a literal union constant in records, arrays, and other composite data structures. The following is a simple example of a record constant that has a union as one of its fields:

```
type
  r   :record
```

```

        b:byte;
        uf:u;
        d:dword;
    endrecord;

static
    sr :r := r:[0, u.d:[$1234_5678], 12345];

```

In this example, HLA initializes the *sr* variable with the byte value zero, followed by a double-word containing \$1234_5678 and a double-word containing zero (to pad out the remainder of the union field), followed by a double-word containing 12345.

You can also create records that have anonymous unions in them and then initialize a record object with a literal constant. Consider the following type declaration with an anonymous union:

```

type
    rau :record
        b:byte;
        union
            c:char;
            d:dword;
        endunion;
        w:word;
    endrecord;

```

Because anonymous unions within a record do not have a type name associated with them, you cannot use the standard literal union constant syntax to initialize the anonymous union field (that syntax requires a type name). Instead, HLA offers you two choices when creating a literal record constant with an anonymous union field. The first alternative is to use the reserved word **union** in place of a *typename* when creating a literal union constant, e.g.,

```

static
    srau :rau := rau:[ 1, union.d:[$12345], $5678 ];

```

The second alternative is a shortcut notation. HLA allows you to simply specify a value that is compatible with the first field of the anonymous union and HLA will assign that value to the first field and ignore any other fields in the union, e.g.,

```

static
    srau2 :rau := rau:[ 1, 'c', $5678 ];

```

This is slightly dangerous since HLA relaxes type checking a bit here, but when creating tables of record constants, this is very convenient if you generally provide values for only a single field of the anonymous union; just make sure that the commonly used field appears first and you're in business.

Although HLA allows anonymous records within a union, there was no syntactically acceptable way to differentiate anonymous record fields from other fields in the union; therefore, HLA does not allow you to create union constants if the union type contains an anonymous record. The easy workaround is to create a named record field and specify the name of the record field when creating a union constant, e.g.,

```

type
    r :record
        c:char;
        d:dword;
    endrecord;

    u :union

```



```

        b:byte;
        x:r;
        w:word;
    endunion;

static
    y :u := u.x:[ r:[ 'a', 5]];

```

You may declare a union constant and then assign data to the specific fields as you would a record constant. The following example provides some samples of this:

```

type
    u_t :union
        b:byte;
        x:r;
        w:word;
    endunion;

val
    u :u_t;
    .
    .
    .
    ?u.b := 0;
    .
    .
    .
    ?u.w := $1234;

```

The two assignments above are roughly equivalent to the following:

```

    ?u := u_t.b:[0];

and

    ?u := u_t.w:[$1234];

```

However, to use the straight assignment (the former example) you must first declare the value *u* as a *u_t* union.

To access a value of a union constant, you use the familiar "dot notation" from records and other languages, e.g.,

```

    ?x := u.b;
    .
    .
    .
    ?y := u.w & $FF00;

```

Note, however, that you may only access the last field of the union into which you've stored some value. If you store data into one field and attempt to read the data from some other field of the union, HLA will report an error. Remember, you don't use union constants as a sneaky way to coerce one value's type to another (use the coercion functions for that purpose).

9.2.4 Pointer Constants

HLA allows a very limited form of a pointer constant. If you place an ampersand ("&") in front of a static object's name (i.e., the name of a **static** variable, **readonly** variable, **storage** variable, procedure, or iterator), HLA will compute the run-time offset of that variable. Pointer constants may not be used in arbitrary constant expressions. You may only use pointer constants in expressions used to initialize **static** or **readonly** variables or as constant expressions in 80x86 instructions. The following example demonstrates how pointer constants can be used:

```
program pointerConstDemo;

static
    t:int32;
    pt: pointer to int32 := &t;

begin pointerConstDemo;

    mov( &t, eax );

end pointerConstDemo;
```

Pointer constants also allow a fixed constant offset by appending "[constant_expression]" to the pointer constant, for example:

```
program pointerConstDemo;

static
    t:int32;
    pt: pointer to int32 := &t[2];

begin pointerConstDemo;

    mov( &t[-4], eax );

end pointerConstDemo;
```

These pointer constants are the address of the specified object plus an offset that is the number of bytes specified by the constant integer expression.

Also note that HLA allows the use of the reserved word **NULL** anywhere a pointer constant is legal. HLA substitutes the value zero for **NULL**. You can also use the HLA compile-time function **@pointer(n)** with an integer constant to tell HLA to treat that number as a pointer constant.

Note that you may obtain the address of the current location counter as a pointer constant by applying the "&" operator to the **@here** keyword, e.g.,

```
mov( &@here, eax );
```

This places the address of the start of the instruction into EAX.

9.2.5 Regular Expression Constants

HLA uses compile-time "regex"-typed variables to hold compiled versions of regular expression. There is no literal form of a regular expression constant. The only way to generate a regular expression constant is in a **val**, **const**, or "?" declaration by assigning the "value" of a #regex macro declaration to a symbol, e.g.,

```
#regex someRegexMacro;
    <<regex macro body>>

#endregex
```

```
const
  compiledRegex :regex := someRegexMacro;
```

See the section on regular expressions in the chapter on *The HLA Compile-Time Language* for more details.

9.3 Constant Expressions in HLA

HLA provides a rich expression evaluator to process assembly-time expressions. HLA supports the following operators (sorting by decreasing precedence):

```
! (unary not), - (unary negation)
*, div, mod, /, <<, >>
+, -
=, ==, <>, !=, <=, >=, <, >
&, |, &, in
```

The following subsections describe each of these operators in detail.

9.3.1 Type Checking and Type Promotion

Many dyadic (two-operand) operators expect the types of their operands to be the same. Prior to actually performing such an operation, HLA evaluates the types of the operands and attempts to make them compatible. HLA uses a type algebra to determine if two (different) types are compatible; if they are not, HLA will report a type mismatch error during assembly. If the types are compatible, HLA will attempt to make them identical via *type promotion*. The type algebra describes how HLA promotes one type to another in order to make the two types compatible.

Usually, you can state a type algebra easily enough by providing "algebraic" type equations. For example, in high-level languages one could use a statement like "r = r + i" to suggest that the type of the resulting sum is real when the left operand is real and the right operand is integer (around the "+" operator). Unfortunately, HLA supports so many different data types and operators that any attempt to describe the type algebra in this fashion will produce so many equations that it would be difficult for the reader to absorb it all. Therefore, this document will rely on an informal English description of the type algebra to explain how HLA operates.

First, if two operands have the same basic type, but are different sizes, HLA promotes the smaller object to the same size as the larger object. Basic types include the following sets: {uns8, uns16, uns32, uns64, uns128}, {int8, int16, int32, int64, int128}, {byte, word, dword, qword, lword}, and {real32, real64, real80}¹. So, if any two operands appear from one of these sets, then both operands are promoted to the larger of the two types.

If an unsigned and a signed operand appear around an operator, HLA produces a signed result. If the unsigned operand is smaller than the signed operand, HLA assigns both operands the signed type prior to the operation. If the unsigned and signed operands are the same size (or the unsigned operand is larger), HLA will first check the H.O. bit of the unsigned operand. If it is set, then HLA promotes the unsigned operand to the next larger signed type (e.g., **uns16** becomes **int32**). If the resulting signed type is larger than the other operand's type, it will be promoted as well. This scheme fails if you have an **uns128** value whose H.O. bit is set. In that case, HLA promotes both operands to **int128** and will produce incorrect results (because the **uns128** value just went negative when it's really positive). Therefore, you should attempt to limit unsigned values to 127 bits if you're going to be mixing signed and unsigned operations in the same expression.

Any mixture of hexadecimal types (**byte**, **word**, **dword**, **qword**, or **lword**) and an unsigned type produces an unsigned type; the size of the resulting unsigned type will be the larger of the two types. Likewise, any mixture of hexadecimal types and signed integer types will produce a signed integer whose size is the larger of the two types. This "strengthening" of the type (hexadecimal types are "weaker" than signed or unsigned types) may seem counter-intuitive to a die-hard

1. As this is being written, HLA doesn't fully support wchar or wstring types; ultimately the support will appear and you can add the sets {char, wchar} and {string, wstring} to the list.

assembly programmer; however, making the result type hexadecimal rather than signed/unsigned can create problems if the result has the H.O. bit set since information about whether the result is signed or unsigned would be lost at that point.

Mixing unsigned values and a **real32** value will produce a **real32** result or an error. HLA produces an error if the unsigned value requires more than 24 bits to represent exactly (which is the largest unsigned value you may represent within the **real32** format). Note that in addition to promoting the unsigned type to **real32**, HLA will also convert the unsigned value to a **real32** value. Promoting the type is not the same thing as converting the value; e.g., promoting **uns8** to **uns16** simply involves changing the type designation of the **uns8** object. HLA doesn't have to deal with the actual value because it keeps all values in an internal 128-bit format. However, the binary representation for unsigned and **real32** values is completely different, so HLA must do the value conversion as well. Note that if you really want to convert a value that requires more than 24 bits of precision to a **real32** object (with truncation), just convert the unsigned operand to **real64** or **real80** and then convert the larger operand to **real32** using the **real32(expr)** compile-time function. Since unsigned values are, well, unsigned and **real32** objects are signed, the conversion process always produces a non-negative value.

Mixing signed and **real32** values in an expression produces a **real32** result. Like unsigned operands, signed operands are limited to 24 bits of precision or HLA will report an error. Technically, you should get one more bit of precision from signed operands (since the **real32** format maintains its sign apart from the mantissa), but HLA still limits you to 24 bits during this conversion. If the signed integer value is negative, so will be the **real32** result.

If you mix hexadecimal and **real32** types, HLA treats the hexadecimal type as an unsigned value of the same size. See the discussion of unsigned and **real32** values earlier for the details.

If you mix an unsigned, signed, or hexadecimal type with a **real64** type, the result is an error (if HLA cannot exactly represent the value in **real64** format) or a **real64** result. The conversion is very similar to the **real32** conversion discussed above except you get 52 bits of integer precision rather than only 24 bits.

If you mix an unsigned, signed, or hexadecimal type with a **real80** type, the result is an error (if HLA cannot exactly represent the value in **real80** format) or a **real80** result. The conversion is very similar to the **real32** conversion discussed above except you get 64 bits of integer precision rather than only 24 bits. Note that conversion of integer objects 64-bits or less will always proceed without error; 128-bit values are the only ones that will get you into trouble.

If you mix a pair of different sized real values in the same expression, HLA will promote (and convert) the smaller real value to the same size as the larger real value.

The only non-numeric promotions that take place in an expression are between characters and strings. If a character and a string both appear in an expression, HLA will promote the character to a string of length one.

9.3.2 Type Coercion in HLA

HLA will report a type mismatch error if objects of incompatible types appear within an expression. Note that you may use the type-coercion compile-time functions to convert between types that HLA does not automatically support in an expression (see the discussion later in this document). You can also use the HLA type coercion operator to attach a specific type to a constant expression. The type coercion operator takes the following form:

```
(type typename constexpr)
```

The *typename* component must be a valid, declared type identifier (including any of the built-in types or appropriate user-defined types). The *constexpr* component can be any constant expression that is reasonably compatible with the specified type. "Reasonably compatible" means that the types are the same size or one of the primitive types. Examples:

```
(type int8 'a')
(type real32 constExpression+2)
(type boolean int8Val)
```

One important thing to remember is that type coercion is a bitwise operation. No conversion is done when coercing one type to another using this type coercion operation.

HLA also achieves type coercion using several compile-time functions. See the chapter on *The HLA Compile-Time Language* for more details on those type coercion functions.

9.3.3 !expr

The expression must be either boolean or a number. For boolean values, `!` computes the standard logical not operation. Numerically, HLA inverts only the L.O. bit of boolean values and clears the remaining bits of the boolean value. Therefore, the result is always zero or one when NOTting a boolean value (even if the boolean object errantly contained other set bits prior to the `!` operation). Remember, the `!` operator only looks at the L.O. bit; if the value was originally non-zero but the L.O. bit was clear¹, then `!` produces true. This is not a zero/not-zero operation.

For numbers, `!` computes the bitwise not operation on the bits of the number, that is, it inverts all the bits. The exact semantics of this operation depend upon the original data type of the value you're inverting. Therefore, the result of applying the `!` operator to an integer number may not always be intuitive because HLA always maintains 128-bits of precision, regardless of the underlying data type. Therefore, a full explanation of this operator's semantics must be given on a type-by-type basis.

uns8: Bits 8..127 of an `Uns8` object are always zero. Therefore, when you apply the `!` operator to an `Uns8` value, the result can no longer be an `Uns8` object since bits 8..127 will now contain ones. Zeroing out the H.O. bits is not wise, because you could be assigning the result of this expression to a larger data type and you may very well expect those bits to be set. Therefore, HLA converts the type of `!u8expr` to type `byte` (which does allow the H.O. bits to contain non-zero values). If you assign an object of type `byte` to a larger object (e.g., type `word`), HLA will copy over the H.O. bits from the byte object to the larger object. Example:

```
val
  u8 :uns8 := 1;
  b8 := !u8; // produces $FFF..FFFE but registers as byte $FE.
  w16 :word := b8; // produces $FF..FFFE but registers as word $FFFE.
```

Note: If you really want to chop the value off at eight bits, you can use the compile-time `byte` function to achieve this, e.g.,

```
val
  u8 :uns8 := 1;
  b8 := byte(!u8); // produces $FE.
  w16 :word := b8; // produces $00FE.
```

uns16: The semantics are similar to `uns8` except, of course, applying `!` to an `uns16` value produces a `word` value rather than a `byte` value. Again, the `!` operator will set bits 16..127 to one in the result. If you want to ensure that the result contains no set bits beyond bit #15, use the compile-time `word` function to strip the value down to 16 bits (just like the `byte` function in the example above).

uns32: The semantics are similar to `uns8` except, of course, applying `!` to an `uns32` value produces a `dword` value rather than a `byte` value. Again, the `!` operator will set bits 32..127 to one in the result. If you want to ensure the result contains no set bits beyond bit #31 use the compile-time `dword` function to strip the value down to 32 bits (just like the `byte` function in the example above).

uns64: The semantics are similar to `uns8` except, of course, applying `!` to an `uns64` value produces a `qword` value rather than a `byte` value. Again, the `!` operator will set bits 64..127 to one in the result. If you want to ensure the result contains no set bits beyond bit #63 use the compile-time `qword` function to strip the value down to 64 bits (just like the `byte` function in the example above).

1. In theory, this should never happen since HLA maintains boolean values as zero or one.

uns128:	Applying the "!" operator to an uns128 object simply inverts all the bits. There are no funny semantics here. Resulting expression type is set to lword .
int8:	Same semantics as byte (see explanation below).
int16:	Same semantics as word (see explanation below).
int32:	Same semantics as dword (see explanation below).
int64:	Same semantics as qword (see explanation below).
int128:	Applying the "!" operator to an int128 object simply inverts all the bits. There are no funny semantics here. Resulting expression type is set to lword .
byte:	Bits 8..127 of a byte (int8) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..7 in the original value and returns this inverted result. Note that the type of the new value is always byte (even if the original sub-expression was int8).
word:	Bits 16..127 of a word (int16) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..15 in the original value and returns this inverted result. Note that the type of the new value is always word (even if the original sub-expression was int16).
dword:	Bits 32..127 of a dword (int32) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..31 in the original value and returns this inverted result. Note that the type of the new value is always dword (even if the original sub-expression was int32).
qword:	Bits 64..127 of a qword (int64) value must all be zeros or all ones. The "!" operator enforces this. If any of the H.O. bits are non-zero, the "!" operator sets them all to zero in the result; if all of the H.O. bits are zero, the "!" operator sets the H.O. bits to ones in the result. Of course, this operator inverts bits 0..63 in the original value and returns this inverted result. Note that the type of the new value is always qword (even if the original sub-expression was int64).
lword:	Applying the "!" operator to an lword object simply inverts all the bits. There are no funny semantics here..

No other types are legal with the "!" operator. HLA will report a type conflict error if you attempt to apply this operator to some other type.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.4 - expr (unary negation operator)

The expression must either be a numeric value or a character set. For numeric values, "-" negates the value. For character sets, the "-" operator computes the complement of the character set (that is, it returns all the characters not found in the set).

Again, the exact semantics depend upon the type of the expression you're negating. The following paragraphs explain exactly what this operator does to its expression. For all integer values (**unsXX**, **intXX**, **byte**, **word**, **dword**, **qword**, and **lword**), the negation operator always does a full 128-bit negation of the supplied operand. The difference between these different data types is how HLA sets the resulting type of the expressions (as explained in the paragraphs below).

uns8:	If the original value was in the range 128..255, then the resulting type is int16 , otherwise the resulting type is int8 . Because uns8 values are always positive, the negated result is always negative, hence the result type is always a signed integer type.
-------	--

uns16:	If the original value was in the range 32678..65535, then the resulting type is int32 , otherwise the resulting type is int16 . Because uns16 values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.
uns32:	If the original value was in the range \$8000_0000..\$FFFF_FFFF, then the resulting type is int64 , otherwise the resulting type is int32 . Because uns32 values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.
uns64:	If the original value was in the range \$8000_0000_0000_0000..\$FFFF_FFFF_FFFF_FFFF, then the resulting type is int128 , otherwise the resulting type is int64 . Because uns64 values are always positive, the negated result is always negative; hence, the result type is always a signed integer type.
uns128:	The result type is always set to int128 . Note that there is no check for overflow. Effectively, HLA treats uns128 operands as though they were int128 operands with respect to negation. So large positive (uns128) values become smaller unsigned values after the negation. If you need to mix and match 128-bit values in an expression, you should attempt to limit your unsigned values to 127 bits.
byte, int8,	
word, int16,	
dword, int32,	
qword, int64,	
lword,	
int128:	Negates the expression (full 128 bits) and assigns the original expression type to the result.
real32:	Negates the real32 value and returns a real32 result.
real64:	Negates the real64 value and returns a real64 result.
real80:	Negates the real80 value and returns a real80 result.
cset:	Computes the set complement (returns cset type). The set complement is all the items that were <i>not</i> previously in the set. Since HLA uses a bitmap representation for character sets, the complement of a character set is the same thing as inverting all the bits in the powerset.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.5 **expr1 * expr2**

For numeric operands, the "*" operator produces their product. For character set operands, the "*" operator produces the intersection of the two sets. The exact result depends upon the types of the two operands to the "*" operator. To begin with, HLA attempts to make the types of the two operands identical if they are not already identical. HLA achieves this via type promotion (see the discussion earlier).

If the operands are unsigned or hexadecimal operands, HLA will compute their unsigned product. If the operands are signed, HLA computes their signed product. If the operands are real, HLA computes their real product. If the operands are integer (signed or unsigned) and less than (or equal to) 64 bits, HLA computes their exact result. If the operands are greater than 64 bits and their product would require more than 128 bits, HLA quietly overflows without error. Note that HLA always performs a 128-bit multiplication, regardless of the operands' sizes; however, objects that require 64 bits or less of precision will always produce a product that is 128 bits or less. HLA automatically extends the size of the result to the next greater size if the product will not fit into an integer that is the same size as the operands. HLA will actually choose the smallest possible size for the product (e.g., if the result only requires 16 bits of precision, the resulting type will be **uns16**, **int16**, or **word**). The resulting type is always unsigned if the operands were unsigned, signed if the operands were signed, and hexadecimal if the operands were hexadecimal.

If the operands are real operands, HLA computes their product and always produces a **real80** result. If you want to produce a smaller result via the '*' operator, use the **real32** or **real64** compile-time function to produce the smaller result, e.g., "real32(r32const * r32const)". Note that all real arithmetic inside HLA is always performed using the FPU, hence the results are always **real80**. Other than trying to simulate the actual products a running program would produce, there is no real reason to coerce the product to a smaller value.

If the operands are character set operands, the '*' operator computes the intersection of the two sets. Since HLA uses a bitmap representation for character sets, this operator does a bitwise logical AND of the two 16-byte operands (this operation is roughly equivalent to applying the "&" operator to two **lword** operands).

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.6 **expr1 div expr2**

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. Supplying any other data type as an operand will produce an error. The **div** operator divides the first expression by the second and produces the truncated quotient result.

If the operands are unsigned, HLA will do a full 128/128-bit division and the resulting type will be unsigned (HLA sets the type to the smallest unsigned type that will completely hold the result). If the operands are signed, HLA will do a full 128/128 bit signed division and the resulting type will be the smallest **intXX** type that can hold the result. If the operands are hexadecimal values, HLA will do a full 128/128 bit unsigned division and set the resulting type to the smallest hex type that can hold the result.

Note that the **div** operator does not allow real operands. Use the "/" operator for real division.

HLA will set the type of the result to the smallest type within its class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.7 **expr1 mod expr2**

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The **mod** operator divides the first expression by the second and produces their remainder (this value is always positive).

If the operands are unsigned, HLA will do a full 128/128-bit division and return the remainder. The resulting type will be unsigned (HLA sets the type to the smallest unsigned type that will completely hold the result).

If the operands are signed, HLA will do a full 128/128 bit signed division and return the remainder. The resulting type will be the smallest **intXX** type that can hold the result.

If the operands are hexadecimal values, HLA will do a full 128/128 bit unsigned division and set the resulting type to the smallest hex type that can hold the result.

Note that the **mod** operator does not allow real operands. You'll have to define real modulus and write the expression yourself if you need the remainder from a real division.

HLA will set the type of the result to the smallest type within its class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values).

9.3.8 **expr1 / expr2**

The two expressions must be numeric. The '/' operator divides the first expression by the second and produces their (**real80**) quotient result.

If the operands are integers (unsigned, signed, or hexadecimal) or the operands are **real32** or **real64**, HLA first converts them to **real80** before doing the division operation. The expression result is always **real80**.

9.3.9 **expr1 << expr2**

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The second operand must be a small (32-bit or less) non-negative value in the range 0..128. The << operator shifts the first expression to the left the number of bits specified by the second expression. Note that

the result may require more bits to hold than the original type of the left operand. Any bits shifted out of bit position 127 are lost.

HLA will set the type of the result to the smallest type within the left operand's class (signed, unsigned, or hexadecimal) that can hold the result (not including sign extension bits for negative numbers or zero extension bits for non-negative values). Note that the '<<<' operator can yield a smaller type (specifically, an eight bit type) if it shifts all the bits off the H.O. end of the number; generally, though, this operation produces larger result types than the left operand.

9.3.10 `expr1 >> expr2`

The two expressions must be integer (signed, unsigned, or hexadecimal) numbers. The second operand must be a small (32-bit or less) non-negative value in the range 0..128. The >> operator shifts the first expression to the right the number of bits specified by the second expression. Any bits shifted out of the L.O. bit are lost. Note that this shift is a *logical* shift right, not an *arithmetic* shift right (this is true even if the left operand is an **intXX** value). Therefore, this operation always shifts a zero into bit position 127.

Shift rights may produce a smaller type than the value of the left operand. HLA will always set the type of the result value to the minimum type size that has the same base class as the left operand.

9.3.11 `expr1 + expr2`

If the two expressions are numeric, the "+" operator produces their sum.

If the two expressions are strings or characters, the "+" operator produces a new string by concatenating the right expression to the end of the left expression.

If the two operands are character sets, the "+" operator produces their union.

If the operands are integer values (signed, unsigned, or hexadecimal), then HLA adds them together. Any overflow out of bit #127 (unsigned or hexadecimal) or bit #126 (signed) is quietly lost. HLA sets the type of the result to the smallest type size that will hold the sum; the type class (signed, unsigned, hexadecimal) will be the same as the operands. Note that it is possible for the type size to grow or shrink depending on the values of the operands (e.g., adding a positive and negative number could reduce the type size, adding two positive or two negative numbers may expand the result type's size).

When adding two real values (or a real and an integer value), HLA always produces a **real80** result.

Since HLA uses a bitmap to represent character sets, taking the union of two character sets is the same as doing a bitwise logical OR of all 16 bytes in the character set.

9.3.12 `expr1 - expr2`

If the two expressions are numeric, the "-" operator produces their difference.

If the two expressions are character sets, the "-" operator produces their set difference (that is, all the characters in *expr1* that are not also in *expr2*).

If the operands are integer values (signed, unsigned, or hexadecimal), then HLA subtracts the right operand from the left operand. Any overflow out of bit #127 (unsigned or hexadecimal) or bit #126 (signed) is quietly lost. HLA sets the type of the result to the smallest type size that will hold their difference; the type class (signed, unsigned, hexadecimal) will be the same as the operands. Note that it is possible for the type size to grow or shrink depending on the values of the operands (e.g., subtracting two negative or non-negative numbers could reduce the type size, subtracting a negative value from a non-negative value may expand the result type's size).

When subtracting two real values (or a real and an integer value), HLA always produces a **real80** result.

Since HLA uses a bitmap to represent character sets, taking the set of two character sets is the same as doing a bitwise logical AND of the left operand with the inverse of the right operand.

9.3.13 Comparisons (`=`, `==`, `<>`, `!=`, `<`, `<=`, `>`, and `>=`)

```
expr1 = expr2
```

```
expr1 == expr2
```

```
expr1 <> expr2
```

```
expr1 != expr2
```

```
expr1 < expr2
expr1 <= expr2
expr1 > expr2
expr1 >= expr2
```

Note: "!=" and "<>" operators are identical. "=" and "==" operators are also identical.

The two expressions must be compatible (described earlier). These operators compare the two operands and return true or false depending upon the result of the comparison.

You may use the "=" and "<>" operators to compare two pointer constants (e.g., "&abc" or "&ptrVar[2]"). The other operators do not allow pointer constant operands.

All the above operators allow you to compare boolean values, enumerated values (types must match), integer (signed, unsigned, hexadecimal) values, character values, string values, real values, and character set values.

When comparing boolean values, note that **false** < **true**.

One character set is less than another is if it is a proper subset of the other. A character set is less than or equal to another set if it is a subset of that second set. Likewise, one character set is greater than, or greater than or equal to, another set if it is a proper superset, or a superset, respectively.

As with any programming language, you should take care when comparing two real values (especially for equality or inequality) as minor precision drifts can cause the comparison to fail.

9.3.14 expr1 & expr2

Note: "&&" and "&" mean different things to HLA. See the section on high-level language control structures for details on the "&&" operator.

The operands must both be boolean or they must both be numbers. With boolean operands the AND operator produces the logical and of the two operands (boolean result). With numeric operands, the AND operator produces the bitwise logical AND of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.15 expr1 in expr2

The first expression must be a character value. The second expression must be a character set. The **in** operator returns **true** if the character is a member of the specified character set; it returns **false** otherwise.

9.3.16 expr1 | expr2

Note: "||" and "|" mean different things to HLA. See the section on high-level language control structures for details on the "||" operator.

The operands must both be boolean or they must both be numbers. With boolean operands the OR operator produces the logical OR of the two operands (boolean result). With numeric operands, the OR operator produces the bitwise or of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.17 expr1 ^ expr2

The operands must both be boolean or they must both be numbers. With boolean operands the ^ operator produces the logical exclusive-or of the two operands (boolean result). With number operands, the ^ operator produces the bitwise exclusive-or of the operands.

If the operand is one of the integer types (signed, unsigned, hexadecimal), then HLA will set the type of the result to the smallest type within that class (signed, unsigned, or hexadecimal) that can hold the result.

9.3.18 (expr)

You may override the precedence of any operator(s) using parentheses in HLA constant expressions.

9.3.19 [comma_separated_list_of_expressions]

This produces an array expression. The type of the expression is an array type whose base element is the type of one of the expressions in the list. If there are two or more constant types in the array expression, HLA promotes the type of the array expression following the rules for mixed-mode arithmetic (see the rules earlier in this document).

9.3.20 record_type_name : [comma separated list of field expressions]

This produces a record expression. The expressions appearing within the brackets must match the respective fields of the specified record type. See the discussion earlier in this document.

9.3.21 identifier

An identifier is a legal component of a constant expression if the identifier's classification is **const** or **val** (that is, the identifier was declared in a constant or value section of the program). The expression evaluator substitutes the current declared value and type of the symbol within the expression. Constant expressions allow the following types:

boolean, enumerated types, uns8, uns16, uns32, uns64, uns128 byte, word, dword, qword, lword, int8, int16, int32, int64, int128, char, real32, real64, real80, string, and cset.

You may also specify arrays whose element base type is one of the above types (or a record or union subject to the following restriction). Likewise, you can specify record or union constants if all of their respective fields are one of the above primitive types or a value array, record, or union constant.

HLA allows array, record, and union constants. If you specify the name of an array, for example, HLA works with all the values of that array. Likewise, HLA can copy all the values of a record or union with a single statement.

HLA allows literal Unicode character and string constants (e.g., u'a' and u"unicode") or identifiers that are of **wchar** or **wstring** type in an expression, but no other terms are allowed in such an expression (as this is being written).

9.3.22 identifier1.identifier2 {...}

Selects a field from a **record** or **union** constant. *Identifier1* must be a record or union object defined in a **const** or **val** section. *Identifier2* (and any following dot-identifiers) must be a field of the record or union. HLA replaces this object with the value of the specified field.

Examples:

```
recval.fieldval
recval.subrecval.fieldval
```

Don't forget that with union constant, you may only access the last field into which you've actually stored data (see the section on union constants for more details).

9.3.23 identifier [index_list]

Identifier must be an array constant defined in either a **const** or **val** section. *Index_list* is a list of constant expressions separated by commas. The index list selects a specified element of the

"identifier" array. HLA reports an error if you supply more indices than the array has dimensions. HLA returns an array slice if you specify fewer indices than the array has dimensions (for example, if an array is declared as "a:uns8[4,4]" and you specify "a[2]" in a constant expression, HLA returns the third row of the array (a[2,0]..a[2,3]) as the value of this term).

Examples:

```
arrayval [0]
aval [1, 4, 0]
```

10 HLA Program Structure and Organization

10.1 HLA Program Structure

HLA supports two types of compilations: *programs* and *units*. A *program* is an HLA source file that includes the code (the "main program") that executes immediately after the operating system loads the program into memory. A *unit* is a module that contains procedures, methods, iterators, and data that is to be linked with other modules. Note that units may be linked with other HLA modules (including an HLA main program) or with code written in other languages (including high-level languages or other x86 assembly languages). This chapter will discuss the generic form of an HLA program; see the chapter on *HLA Units and External Compilation* for a detailed description of HLA units.

An executable file must have exactly one main program (written either in HLA or some other language). Therefore, most applications written entirely in HLA will have exactly one program module and zero or more units (it is possible to fake a program module using units; for more information see the on-line documents "Taking Control Over Code Emission" and "Calling HLA Code from Non-HLA Programs with Exception Handling" on Webster (<http://webster.cs.ucr.edu>). Therefore, the best place to begin discussion HLA program structure is by defining the HLA **program**. Here's the minimalist HLA program:

```
program pgmID;  
begin pgmID;  
end pgmID;
```

In this example, *pgmID* is a user-defined identifier that names the program. Note that this name is local to the program (that is, it is not visible outside the source file and neither the source file's name nor the executable's file name need be the same as this name (though it's not a bad idea to make them the same). Note that the exact same identifier following the **program** reserved word must follow the **begin** and **end** reserved words.

The minimalist HLA program, above, doesn't do much; if you compile and execute this program it will immediately return control to the operating system. However, this short program actually does quite a bit for an empty assembly language program. When you create an HLA program, you're asking HLA to automatically generate some template code to do certain operations such as initializing the exception-handling system, possibly setting up command-line parameters for use by the program, and emitting code to automatically return control to the operating system when the program completes execution (by running into the **end *pgmID*** clause). As this code is generally needed for every HLA assembly language program, it's nice that the HLA compiler will automatically emit this template code for you. If you happen to be a die-hard assembly programmer and you don't want the compiler emitting any instructions you haven't explicitly written, fear not, HLA doesn't force you to accept the code it's written; for more details, see the "Taking Control Over Code Emission" article on Webster that was mentioned earlier.

A non-minimalist HLA program takes the following generic form:

```
program pgmID;  
  << declarations >>  
begin pgmID;  
  << main program instructions>>  
end pgmID;
```

The <<*declarations*>> section is where you will put the declarations/definitions for constants, data types, variables, procedures, methods, iterators, tables, and other data. The <<*main program instructions*>> section is where you will put machine instructions and HLA HLL-like statements.

An HLA unit is even simpler than an HLA program. It takes the following form:

```
unit unitID;  
  << declarations >>  
end unitID;
```

In this example, *unitID* is a user-defined identifier that names the unit. Note that this name is local to the unit (that is, it is not visible outside the source file and the source file's name need be the same as this name. Note that the exact same identifier following the **unit** reserved word must follow the **end** reserved word. Unlike programs, units do not have a **begin** clause following by a sequence of instructions; this is because units don't provide the main program code for the application. Again, for more details about units, see the chapter on HLA Units.

10.2 The HLA Declaration Section

The declaration section in an HLA program or unit is relatively complex, supporting the definition and declaration of most of the components in the HLA program or unit. An HLA declaration section generally contains one or more of the following items:

- A labels section (**label**)
- A constant declaration section (**const**)
- A values declaration section (**value**)
- An automatic variables declaration section (**var**)
- An initialized static data storage declaration section (**static**)
- An initialized read-only data storage declaration section (**readonly**)
- An uninitialized static data storage declaration section (**storage**)
- A procedures declaration section (**proc**)
- Old-style procedure, method, and iterator declarations
- A **namespace** declaration section

These sections may appear in any order in a **program** or **unit** declarations section and multiple instances of each of these sections may appear in the declarations. The following subsections describe each of these declaration sections in detail.

10.2.1 The HLA LABEL Declaration Section

The HLA label section is a very special-purpose (and rarely used) declaration section in which you declare forward-referenced and external statement labels. The syntax for the **label** section is either of the following:

```
label  
  << label declarations >>
```

or

```
label  
  << label declarations >>  
endlabel;
```

The **endlabel** clause is optional. If it is present it explicitly marks the end of the forward label declaration section; if it is absent, then the next declaration section or the **begin** keyword will implicitly end the forward label declaration section.

Each label declaration takes one of the three following forms:

```
userLabel_1;
userLabel_2; external;
userLabel_3; external( "externalLabelName" );
```

In these examples, *userLabel_x* (x=1, 2, or 3) is a user-defined identifier.

The first example above is a *forward label declaration*. This tells HLA that you're promising to declare the statement label within the scope of the **label** section (HLA will generate an error if you fail to declare the statement label within the scope of the **label** statement).

The *scope* of a **label** statement is the body of instructions associated with the main program, procedure, method, or iterator that immediately contains the **label** declaration section. For example, if the label statement appears in the declaration section of an HLA program, the corresponding statement label must be defined in the body of that program:

```
program labelDemo;
label
    someLabel;

    << other declarations >>
begin labelDemo;

    << main program instructions, part 1 >>
    someLabel: // someLabel must be defined in this code.
    << main program instructions, part 2 >>

end labelDemo;
```

Note that HLA automatically handles forward-referenced labels within the (machine instructions) body of a program, procedure, method, or iterator, without an explicit **label** declaration. The following is legal even though you do not have a forward declaration of *someLabel*:

```
program labelDemo;
.
.
.
begin labelDemo;

.
.
.
    lea( eax, &someLabel );
.
.
.
    jmp someLabel;
.
.
.
    someLabel: // someLabel's declaration appears after its use.
```

```

        .
        .
        .
end labelDemo;

```

The above is legal because the procedure references *someLabel* in the same scope where it is declared. Now consider the following example:

```

program labelDemo;
    .
    .
    .
    procedure ReferencesSomeLabel;
        .
        .
        .
        begin ReferencesSomeLabel;
            .
            .
            .
            lea( eax, &someLabel );// Illegal! someLabel is not defined in this
            procedure.
            .
            .
            .
        end ReferencesSomeLabel;
begin labelDemo;

    .
    .
    .
    someLabel: // someLabel's declaration appears outside the scope of its
    use.

    .
    .
    .

end labelDemo;

```

HLA will generate an error in this example because forward references to statement labels must be resolved within the scope of the procedure (or program) containing the forward reference. When HLA encounters the "**end** *ReferencesSomeLabel*;" clause in the procedure above, it will report that you haven't defined *someLabel* in that procedure. The solution to this problem is to use the **label** statement to create a forward symbol definition so that *someLabel* is defined (albeit at a different lex level) when HLA encounters the **lea** statement in the previous example. The following code demonstrates how to do this:

```

program labelDemo;
label
    someLabel;
    .
    .

```



```
.
procedure ReferencesSomeLabel;
.
.
begin ReferencesSomeLabel;
.
.
    lea( eax, &someLabel ); // This is legal because of the label
statement.
.
.
end ReferencesSomeLabel;

begin labelDemo;

.
.
    someLabel: // someLabel had a forward declaration.

.
.

end labelDemo;
```

You can also create external label definitions by attaching the **external** option to a label definition. External label definitions take one of two forms:

```
label
    someLabel; external;
    someExtLabel; external( "externalName" );
```

The first form assumes that *someLabel* is defined (and the name is made public) in some other source/object module using the name *someLabel*. The second form assumes that "*externalName*" is defined in some other source/object module and uses the name *someExtLabel* to refer to that symbol.

To create a public label that you can reference in another source module, you put an external label definition in the same source file as the actual symbol declaration, e.g.,

```
program labelDemo;
label
    someLabel; external;
.
.
.

begin labelDemo;

.
.
```

```

        .
        someLabel: // someLabel is a public symbol.
        .
        .
        .
end labelDemo;

```

The **label** statement rarely appears in most HLA programs. It is very unusual to reference a symbol that is declared outside the scope of that usage. External symbols are usually procedures, methods, or iterators, and a program will typically use an external **procedure**, **iterator**, or **method** declaration rather than a **label** statement to declare such symbols. Nevertheless, **label** declarations are necessary on occasion, so you should keep the forward label declaration statement in mind.

Note that **label** declarations will not make a local symbol in some scope (that is, within some procedure) visible to code outside that scope. The following will generate an error:

```

program labelDemo;
label
    someLabel;
    .
    .
    .
    procedure declaresSomeLabel;
        .
        .
        .
        begin declaresSomeLabel;
            .
            .
            someLabel:// This is local to this procedure.
            .
            .
            .
        end declaresSomeLabel;
begin labelDemo;
    .
    .
    .
    // This does not reference someLabel in declaresSomeLabel!
    lea( eax, &someLabel );
    .
    .
    .
end labelDemo;

```

The scope of the symbol *someLabel* defined in *declaresSomeLabel* is limited to the *declaresSomeLabel* procedure. In order to make *someLabel* visible outside of *declaresSomeLabel*, you must make that symbol global. This is done by following the label declaration with two colons instead of one colon:

```

program labelDemo;
.
.
.
procedure declaresSomeLabel;
.
.
.
begin declaresSomeLabel;
.
.
.
    someLabel::// This is a global symbol.
.
.
.
end declaresSomeLabel;

begin labelDemo;

.
.
.
// This is legal

lea( eax, &someLabel );

.
.
.

end labelDemo;

```

Note that global symbols are not automatically public. If you need a symbol to be both global to a procedure and public (visible outside the source file), you must also define that global symbol as external in a **label** statement:

```

program labelDemo;
label
    someLabel; external;
.
.
.
procedure declaresSomeLabel;
.
.
.
begin declaresSomeLabel;
.
.
.
    someLabel::// This is a global and public symbol.
.
.
.

```

```

        .
        end declaresSomeLabel;

begin labelDemo;

        .
        .
        .
        // This is legal

        lea( eax, &someLabel );

        .
        .
        .

end labelDemo;

```

Note that global label declarations only make the symbol global at the previous lex level, not across the whole program. The following will not work properly because *label1* is only visible in the *q* and *p* procedures, not in the main program.

```

program t;
label
    label1;

procedure p;

    procedure q;
    begin q;

        label1::

    end q;

begin p;
end p;

begin t;

    lea( eax, label1 );

end t;

```

The solution to this problem is to make the symbol public by declaring it **external** in both the *q* procedure and in the main program:

```

program t;
label
    label1; external;

procedure p;

```

```
    procedure q;  
    label  
        label1; external;  
  
    begin q;  
  
        label1:  
  
    end q;  
  
begin p;  
end p;  
  
begin t;  
  
    lea( eax, label1 );  
  
end t;
```

Of course, referencing a label in a nested procedure like this is highly unusual and is probably an indication of a poorly designed program. If you find yourself writing this kind of code, you might want to reconsider your program's architecture.

10.2.2 The HLA **CONST** Declaration Section

The HLA **const** section is where you declare symbolic (manifest) constants in an HLA program or unit. The syntax for the **const** section is either of the following:

```
const  
    << constant declarations >>  
  
or  
  
const  
    << constant declarations >>  
endconst;
```

The **endconst** clause is optional at the end of the constant declarations in a declaration section; some programmers prefer to explicitly end a constant declaration section with **endconst**, others prefer to implicitly end the constant declarations (by starting another declaration section or with the **begin** keyword). The choice is yours, the language doesn't prefer either method nor does good programming style particularly specify one syntax over the other.

Each constant declaration takes one of the following forms:

```
userDefinedID := <<constant expression>>;  
or  
userDefinedID : typeId := <<constant expression>>;
```

Here are some examples:

```
const  
    hasEdge_c := false;  
    elementCnt_c := 25;
```

```

weight_c:= 32.5;

debugMode_c:boolean:= true;
maxCnt_c:uns32:= 15;
oneHalf_c:real32:= 0.5;

```

The "c" suffix is an HLA programming convention that tells the reader the identifier is a constant identifier. Although it's probably good programming style for you to follow this convention in your own HLA programs, HLA does not require this suffix on constant identifiers; any valid HLA identifier is fine when creating symbolic constants.

If you do not specify a data type for the symbolic constant declaration (as the first three examples above demonstrate), then HLA will infer the data type from the type of the constant expression. While this is convenient in many cases, do be aware that HLA might not choose the same data type you would explicitly provide. This is because a constant expression's type can be ambiguous, in which case HLA will use whatever type it finds convenient that will work. In the examples above, *hasEdge_c* must be a boolean constant because there is no ambiguity about the type of the constant **false**. The remaining two examples without an explicit type (*elementCnt_c* and *weight_c*) do not have constant expressions with an unambiguous type. The constant 25 is valid for types **uns8**, **uns16**, **uns32**, **uns64**, **uns128**, **byte**, **word**, **dword**, **qword**, **tbyte**, **lword**, **int8**, **int16**, **int32**, **int64**, and **int128** (and even **real32**, **real64**, and **real80** if you really want to push things). The HLA language definition does not require this constant (25) to assume any one of these particular values; HLA is free to choose whatever compatible type it wants for this constant. In most cases, it won't make a difference whether HLA chooses the type **uns8** or **uns32** for this constant (or any other of the legal types). However, there are many times that HLA might choose a type that will create problems with the code you're writing; therefore, it's a good idea to always explicitly provide a data type as do the last three examples above.

Note that constant expressions in a constant declaration support all the valid constant expression types discussed in the chapter on HLA Language Elements, including string, character set, array, record, union, and pointer constants. Indeed, it is often more convenient to create a constant for some structured data type and use that constant when initializing a static object than to assign the structured constant directly to the static object, e.g.,

```

const
    myArray_c :dword[ 8 ]:= [0,1,2,3,4,5,6,7];

static
    myArray:dword[@elements(myArray_c)] := myArray_c;

```

Note the use of *@elements(myArray_c)* rather than 8 to specify the number of dword array elements in the *myArray* declaration. By declaring the static array this way, you can change the *myArray_c* constant declaration by adding or removing array elements and the declaration for *myArray* will adjust its size automatically when you recompile.

One benefit to use structured constant declarations to initialize static objects is that you have full access to the (individual element or field) values of that structured constant during assembly. For example, you could reference *myArray_c[0]* in an HLA compile-time language sequence and know that you're getting the same value that goes into element zero of *myArray* at run time.

Objects you declare in a **const** section, as the name suggests, have a fixed value throughout the scope containing that symbol declaration. The value remains fixed both at compile time and at run time. Note, however, that HLA supports block-structured scoping rules so the symbol and its value might not be visible or available for use outside the scope in which you've declared the symbol. Consider the following program example:

```

program constDemo;
const
    sym := 10;
    .
    .
    .
procedure usesSym;

```

```

begin usesSym;
    .
    .
    .
    mov( sym, eax ); // Loads 10 into eax
    .
    .
    .
end usesSym;

begin constDemo;

    .
    .
    .
mov( sym, eax ); // Loads 10 into eax
    .
    .
    .

end constDemo;

```

In this example, both instructions that use the symbol `sym` reference the same object and load the same value into `eax`. This is because HLA's block-structured scoping rules make global symbols visible inside procedures that don't redefine the symbol. Consider, however, the following example:

```

program constDemo;
    .
    .
    .
    procedure usesSym;
    const
        sym := 10;
    begin usesSym;
        .
        .
        .
        mov( sym, eax ); // Loads 10 into eax
        .
        .
        .
    end usesSym;

begin constDemo;

    .
    .
    .
mov( sym, eax ); // This is illegal!
    .
    .
    .

```

```
end constDemo;
```

HLA will reject this example because the second usage of *sym*, in the main program, is outside the scope of the symbol's declaration within the *usesSym* procedure (see the chapter on procedures for more information about HLA's scoping rules). Now consider this last example:

```
program constDemo;
const
  sym = 10;
  .
  .
  .
  procedure usesSym;
  begin usesSym;
    .
    .
    mov( sym, eax ); // Loads 10 into eax
    .
    .
  end usesSym;

  procedure declaresLocalSym;
  const
    sym := 25;
  begin declaresLocalSym;
    .
    .
    mov( sym, eax ); // Loads 25 into eax
    .
    .
  end declaresLocalSym;

begin constDemo;
  .
  .
  .
  mov( sym, eax ); // Loads 10 into eax
  .
  .
end constDemo;
```

This example seems to contradict the statement given earlier that constant declarations can have only a single value throughout the source file. The first usage of *sym* in the *usesSym* procedure loads the value 10 into EAX, just as in the earlier example. In the procedure *declaresLocalSym* we see a second declaration of *sym* with the value 25. When the code in this procedure references *sym*, HLA substitutes the value 25 for the symbol. This action seems to contradict the statement that a constant symbol has a fixed value throughout the compilation. However, the second declaration of *sym* is not a redeclaration of the original symbol (giving it a new value); instead, this is the declaration of a brand-new symbol that just happens to share the

same name (*sym*) as a symbol declared in the main program. The scope of this second symbol is limited to the procedure in which it is defined (and any procedures declared within it, though there are no such procedures in this example). The original symbol is not redefined, it is simply "hidden from view." At the end of *declaresLocalSym*, its local symbols are hidden and the global symbols are again visible. Note that the constant *sym* reverts to the value 10 at this point. Again, not meaning to sound redundant, it's important for you to understand that the two *sym* identifiers represent different constant objects whose visibility is controlled by the scope of those identifiers. The chapter on Procedures goes into greater detail about scope and how it affects the visibility of your symbols.

Unlike labels, you cannot create "external" constants. HLA **const** objects are *manifest* constants. This means that HLA substitutes the values of the **const** symbols wherever they appear in the source file before actually compiling the statements containing those symbols. In the object code file that HLA produces, the constant symbol no longer exists in any form; just the value of that symbol (unlike, say, an external label or procedure definition that passes the name of the symbol on to the linkage editor [linker] in order to properly combine object modules containing mutually-dependent symbols). If you want to use a constant symbol in multiple source files, the appropriate way to do this is to put the symbol into a header file and include that header file in all the source files that use the symbol.

10.2.3 The HLA VAL Declaration Section and the Compile-Time "?" Statement

The HLA **val** declaration section is very similar to the **const** declaration section insofar as you use it to declare symbol names that have a constant value at run time. The difference between the **val** and the **const** declaration sections is that you can reassign a different value to a **val** constant during the assembly/compilation process. This is useful for creating compile-time variables and handling a few other situations where **const** objects won't work. The syntax for the **val** section is either of the following:

```
val
    << value declarations >>
```

or

```
val
    << value declarations >>
endval;
```

As for the **const** section, either syntax is perfectly acceptable to HLA and either form neither form is particular preferred based on good programming style.

The syntax for the individual value declarations is identical to that of the constant declarations in a **const** section. There are two main differences: simple declarations without an assignment and value redefinitions.

The first difference is that a value declaration may consist of a constant identifier and a type identifier without the assignment of a constant expression. For example:

```
val
    sym :uns32;
```

This form creates the identifier without giving it an explicit value. HLA assumes that you are going to assign a value to this **val** constant before you use the value of that constant.

The second difference is that you can redefine the value of a value object multiple times in a program, for example:

```
program valDemo;
val
    sym :uns32;
```

```

        .
        .
        .
val
    sym := 10;
        .
        .
        .
val
    sym := sym + 1;
        .
        .
        .

begin valDemo;

        .
        .
        .
mov( sym, eax ); // Loads 11 into eax
        .
        .
        .

end valDemo;

```

Note that value redefinition in a **val** section only takes place when reassigning the value in the same scope as the original symbol definition. If you attempt to redefine the symbol at some point in the program that would have a different scope, then you will simply create a new object with the same name that is limited to the scope of the new definition. For example, consider the following code:

```

program t;
val
    i:=0;
endval;

    procedure u;
    val
        i := 1;
    begin u;

        #print( "i=", i )

    end u;

begin t;

    #print( "i=", i );

end t;

```

This example prints "i=1" and then "i=0" during compilation. The second declaration of *i* in procedure *u* is a local symbol (local to *i*), this declaration does not affect the original value of the *i* constant. To overcome this problem and provide a way to reassign the value of a **val** constant

anywhere in an HLA source file (including outside **val** declaration sections), HLA provides the compile-time assignment statement. An HLA compile-time assignment statement is legal anywhere a space is legal within the confines of an HLA **program** or **unit**. The HLA compile-time assignment statement takes one of the following two forms:

```
?valIdentifier := <<constant expression>>;
?valIdentifier :typeID := <<constant expression>>;
```

In these examples *valIdentifier* is either an undefined symbol or a constant identifier that was previously declared in a **val** declaration section or an HLA "?" compile-time assignment statement. In some respects, the HLA compile-time assignment statement is more flexible than the assignment of a value constant within a **val** section. Consider the following two programs that produce identical results:

```
program t1;
val
    i:=10;
begin t1;

    #print( "i=", i ); // Prints "10" at compile-time

end t1;

program t2;
?i:=10;
begin t2;

    #print( "i=", i ); // Prints "10" at compile-time

end t2;
```

There is, however, a major limitation to defining **val** constant identifiers in an HLA compile-time assignment statement: you cannot redefine the meaning of a symbol within some different scope (at a higher lex level) when using the compile-time assignment statement. For example, the following is illegal:

```
program t;
static
    i:uns32;

    procedure u;
    ?i := 1;// This is illegal!
    begin u;

        #print( "i=", i )

    end u;

begin t;
.
.
.
end t;
```

The problem here is that the HLA compile-time assignment statement only defines a new symbol if it was previously undefined. In this example the symbol *i* was already defined as a static

variable. As only value constant identifiers may appear in an HLA compile-time assignment statement, HLA will reject this program. Note, however, that the following is legal:

```

program t;

    procedure u;
    ?i := 1; // This is legal!
    begin u;

        #print( "i=", i )

    end u;

static
    i:uns32;

begin t;
    .
    .
    .
end t;

```

This program will compile (assuming you have something reasonable between the **begin** and **end** clauses) and print "i=1" during compilation. The difference here is that *i* was undefined at the point of the "?i := 1;" assignment statement so HLA was able to create a constant identifier (local to procedure *u*). At the end of procedure *u*, the symbol *i* was hidden from the rest of the compilation so the declaration of *i* in the main program does not produce a duplicate definition error. By the way, if you really needed to define *i* as a value constant with procedure *u* in the illegal example, you could do the following:

```

program t;
static
    i:uns32;

    procedure u;
    val
        i:uns32 := 1;
    begin u;

        #print( "i=", i )

    end u;

begin t;
    .
    .
    .
end t;

```

Value constants you declare in a **val** section are not subject to the restriction that the symbol must be (globally) undefined at the point of the declaration. In the example above, *i* is a local symbol in procedure *u* that just happens to be a **val** object with the value one.

Although **val** objects are syntactically similar to **const** objects, you use them in an HLA program in almost completely different ways. The main purpose for **val** objects is to create compile-time variables that you can use to control compilation via compile-time loops, conditional compilation, and macros. Comparing **const** and **val** objects at compile time is quite similar to

comparing **readonly** and **static** objects at run time. The following example demonstrates how you can use a **val** object in a program to unroll a loop at compile time:

```

program unroll;
static
    ary:uns32[16];

begin t;

    ?loopIndex :uns32 := 0;
    #while( loopIndex < 16 )

        mov( loopIndex, ary[ loopIndex*4 ] );
        ?loopIndex := loopIndex + 1;

    #endwhile
    .
    .
    .
end t;

```

Note that the **#while** loop executes at compile time, not at run time. The code between the **#while** and **#endwhile** compile-time statements is equivalent to the following 16 statements:

```

mov( 0, ary[ 0*4 ] );
mov( 1, ary[ 1*4 ] );
mov( 2, ary[ 2*4 ] );
mov( 3, ary[ 3*4 ] );
mov( 4, ary[ 4*4 ] );
mov( 5, ary[ 5*4 ] );
mov( 6, ary[ 6*4 ] );
mov( 7, ary[ 7*4 ] );
mov( 8, ary[ 8*4 ] );
mov( 9, ary[ 9*4 ] );
mov( 10, ary[ 10*4 ] );
mov( 11, ary[ 11*4 ] );
mov( 12, ary[ 12*4 ] );
mov( 13, ary[ 13*4 ] );
mov( 14, ary[ 14*4 ] );
mov( 15, ary[ 15*4 ] );

```

This is because each iteration of the **#while** loop at compile time compiles all of the statements between the **#while** and **#endwhile** statements. For more information on the **#while/#endwhile** statement and using **val** objects as compile-time variables, please see the chapter on the HLA Compile-Time Language.

10.2.4 The HLA TYPE Declaration Section

Examples of the HLA **type** declaration section have been so numerous in the chapter on HLA Language Elements that describing them here is almost redundant (please review that chapter for more details). Nevertheless, for completeness and for the sake of a reference guide, this section describes the syntax of a **type** declaration section. A type declaration section takes one of the following two forms:

```

type
    << type declarations >>

```

or

```
type
  << type declarations >>
endtype;
```

A type declaration takes one of the following forms:

```
newTypeID : typeID;
newTypeID : typeID [ list_of_array_dimensions ];
newTypeID : procedure (<<optional_parameter_list>>);
newTypeID : record <<record_field_declarations>> endrecord;
newTypeID : union <<union_field_declarations>> endunion;
newTypeID : class <<class_field_declarations>> endclass;
newTypeID : pointer to typeID;
newTypeID : enum{ <<list_of_enumeration_identifiers>> };
```

The purpose of the HLA type section is to declare a new type identifier that you can use when declaring **const**, **val**, **var**, **static**, **readonly**, and **storage** objects. You can also use type identifiers you declare in an HLA **type** section to define procedure prototypes in an HLA **proc** section. Each of the forms above deserves its own subsection to describe it, so the following subsections do just that.

Note that a type declaration only defines a type identifier you can use for declaring other objects in an HLA source file. A type declaration does not create a variable or constant object of the specified type. You can use the type identifier in some other declaration section (**const**, **val**, **var**, **static**, **readonly**, **storage**, etc.) to actually define an object of that type.

10.2.4.1 typeID

Before describing the valid syntax forms for the type declaration section, it's worthwhile to take a moment to describe the *typeID* item that appears in many of the type declarations. The *typeID* item is a single identifier whose classification is "type" (duh). This can be any of the HLA built-in types:

```
boolean
enum
uns8
uns16
uns32
uns64
uns128
byte
word
dword
qword
tbyte
lword
int8
int16
int32
int64
int128
char
wchar
real32
real64
```

```

real80
real128
string
zstring
u
nicode
cset

text
thunk.

```

The *typeID* can also be any user-defined type identifier you've previously declared in a type declaration section.

10.2.4.2 newTypeID : typeID;

The least complex type declaration is a simple type isomorphism, where you take an existing type and create a new type with all the same attributes except that you use a different name for the type. For example, suppose you want to use the identifier *integer* rather than **int32** in your programs. You could do this with the following type declaration:

```

type
    integer :int32;

```

Within the scope of this declaration, you can use the type name *integer* anywhere you want to declare a 32-bit signed integer object.

Warning: exercise care when using type isomorphisms of built-in types in an HLA program. If you're writing an HLA program, you can generally assume that people reading the source files you write are reasonably familiar with HLA's built-in types. By creating aliases of those type names, you make it harder for people who already know HLA to read and understand your programs because they have to mentally translate your new types to the more familiar type names. It might seem "cool" to use C++ type names or type names from some other programming language, but other people reading your programs might not share your enthusiasm for the renamed types.

One place where type isomorphisms might make sense is when you're creating a new type that is intended to be a subset of the full type (whose range you check at run time). For example, suppose you use a set of integers that must be in the range 0..31 for certain sections of your program. You could create a type definition like the following to let people know that variables of the specified type are supposed to lie in the range 0..31:

```

type
    smallInt_t :int8; // Holds values in the range 0..31

```

At run time you could use the bound instruction to verify that values you assign to a *smallInt_t* object are actually in the range 0..31:

```

    bound( eax, 0, 31 ); // Raises an ex.bound exception if not in the
range 0..31.
    mov( al, smallIntVar );

```

This example also demonstrates another common HLA programming convention: using an "_t" suffix on user-defined type identifiers.

10.2.4.3 newTypeID : typeID [list_of_array_bounds];

This form creates an array type with the specified number of elements. The *list_of_array_bounds* item is a list of one or more unsigned integer values that are greater than zero (and, generally, greater than one). If there are two or more array bound values, the type is a multi-

dimensional array type and the number of elements is the product of all of the array bound values. Note that HLA arrays are always indexed from zero to the array's bound value *minus one*. So an array declared as

10.2.4.4 **newTypeID : procedure (<<optional_parameter_list>>);**

A complete discussion of procedure pointer types appears in the chapter on procedures. Please see that document for a discussion of procedure pointer types. Like all pointer types, objects that are procedure pointers will consume four bytes in memory (and those four bytes typically hold the address of some procedure).

10.2.4.5 **newTypeID : record <<record_field_declarations>> endrecord;**

Please see the discussion of Record Data Types earlier in this document for examples of **record** type declarations, their syntax, and their use.

10.2.4.6 **newTypeID : union <<union_field_declarations>> endunion;**

Please see the discussion of Union Data Types earlier in this document for examples of **union** type declarations, their syntax, and their use.

10.2.4.7 **newTypeID : class <<class_field_declarations>> endclass;**

A complete discussion of **class** types appears in the chapter on Classes and Object-Oriented Programming in HLA. Please see that document for a discussion of class types.

10.2.4.8 **newTypeID : pointer to typeID;**

Please see the discussion of Pointer Data Types earlier in this document for examples of **pointer** type declarations, their syntax, and their use.

10.2.4.9 **newTypeID : enum{ <<list_of_enumeration_identifiers>> };**

Please see the discussion of Enumerated Data Types earlier in this document for examples of **enum** type declarations, their syntax, and their use.

10.2.5 **The HLA VAR Declaration Section**

The **var** section is where you declare automatic variables in an HLA **procedure**, **method**, **iterator**, or **program**. The HLA **var** section may not appear in the declaration section of an HLA **unit** or **namespace**. A **var** section may also appear in an HLA class, but the storage mechanism for class **var** objects is not the same as for procedures, methods, and iterators. Please see the chapter on Object-Oriented Programming for more details about **var** declarations in a class. The basic syntax for an HLA **var** section is the following:

```
var
    << variable declarations >>
```

or


```
var
  << variable declarations >>
endvar;
```

The syntax for the variable declarations is similar to type declarations; each variable declaration takes one of the following forms:

```
varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record <<record_field_declarations>> endrecord;
varID : union <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };
```

These statements will allocate sufficient storage on the stack for each variable (*varID*) that you declare in the **var** section.

The HLA **var** section also supports an **align** directive; the syntax for the **align** directive in a **var** section is the following:

```
align( constant_expression );
```

The *constant expression* must be a fully-defined constant expression that evaluates to a power of two that lies in the range 1..16. Technically, the only value that makes sense for the align expression is 4, as you will soon see.

The **var** section declares variables for which the HLA run-time code automatically allocates storage upon entry into a procedure (note: the HLA run-time system automatically allocates storage only if the **@frame** procedure option is enabled; otherwise it is the programmer's responsibility to actually allocate the storage). Automatic variable storage allocation is accomplished using a *standard entry sequence* into a procedure, such as the following

```
push( ebp );// Save old frame pointer
mov( esp, ebp );// Put new frame pointer value into EBP
sub( _vars_, esp );// Allocate storage for var variables on stack
```

Automatic variables (**var** objects, or *auto* variables) are referenced using negative offsets from the EBP (base pointer) register into the procedure's *stack frame* (also known as an *activation record*). The previous frame pointer (EBP) value is found at [EBP+0] and the auto variables are found on the stack below this location. Each variable is allocated some amount of storage (determined by the variables type) and the offset of the variable (from EBP) is computed by subtracting the variable's size from the offset of the previous object in the **var** section. Consider the following **var** declaration section; the comments tell you the offset to each of the objects from the EBP register (these offsets assume that there is no display):

```
var
  d :dword;// offset = ebp-4
  s :string;// offset = ebp-8
  u :uns32;// offset = ebp-12
  i :int32;// offset = ebp-16
  w :word; // offset = ebp-18
  b :byte; // offset = ebp-19
  c :char; // offset = ebp-20
```

The offset of each object is computed by subtracting the size of the object from the offset of the previous object in the **var** declaration section (the first object's offset is computed by subtracting the object size from zero, which is the offset of the saved EPB value).

Because the x86 supports a 1-byte offset (+/- 128 bytes) form of the "[EBP+offset]" addressing mode, your code will be slightly shorter if you group all your small variable objects at the beginning of the **var** declaration section and putting all your structured data types (e.g., arrays, records, unions, and character sets) near the end of **var** section. Consider the following two variable declaration sections:

```
var
  d  :dword;    // offset = ebp-4
  s  :byte[256]; // offset = ebp-260
```

versus

```
var
  s  :byte[256]; // offset = ebp-256
  d  :dword;    // offset = ebp-260
```

The instruction "mov(d, eax);" is three bytes shorter if you use the first set of declarations above (where *d*'s offset is -4) because HLA can encode the offset in one byte instead of a double word. By putting the declarations of the smaller objects at the beginning of the **var** section, you can increase the number of variables that you can reference with a 1-byte displacement.

As HLA processes each variable in a **var** section, it computes the offset of that variable by subtracting the variable's size from the offset of the previous variable. If you mix different sized variables in the **var** section, you may not get optimal addresses for each of the variables. Consider the following **var** section and the corresponding variable offsets:

```
var
  b  :byte;     // offset = ebp-1
  w  :word;     // offset = ebp-3
  d  :dword;    // offset = ebp-5
  q  :qword;    // offset = ebp-13
```

Assuming that EBP points at an address that is a multiple of four (and it usually will), all of these variables will be misaligned except for **b**, the byte variable. One solution to this problem is to use the **align** directive to align each variable at an offset that is a multiple of that variable's size:

```
var
  b  :byte;     // offset = ebp-1
  align(2);
  w  :word;     // offset = ebp-4
  align(4);
  d  :dword;    // offset = ebp-8
  align(8);
  q  :qword;    // offset = ebp-16
```

Unfortunately, sticking an align directive before each variable declaration is a pain. Fortunately, the **var** declaration supports the same alignment options as **record** declaration. Consider the following:

```
var[4];
  b  :byte;     // offset = ebp-4
  w  :word;     // offset = ebp-8
  d  :dword;    // offset = ebp-12
  q  :qword;    // offset = ebp-24
```

Of course, allocating four bytes for each automatic variable can be wasteful; you can also do the following:

```
var[4:1];
```

```

b  :byte;    // offset = ebp-1
w  :word;    // offset = ebp-4
d  :dword;  // offset = ebp-8
q  :qword;  // offset = ebp-16

```

See the discussion of record type declarations earlier in the chapter on HLA Language Elements for more details about the alignment options.

There is one big issue concerning the use of the **var** section **align** statement and the alignment options: generally you may only assume that the stack pointer is aligned to a 4-byte address upon entry into a subroutine. Therefore, alignment values other than 1, 2, or 4 may not achieve the desired memory alignment for your automatic variables. If you absolutely must have your automatic variables aligned on a boundary greater than four, you will have to explicitly guarantee that the variable is properly aligned in the activation record. There are a couple of different ways to do this.

The first way is to allocate storage on the stack (using `talloc`), create an address into that storage area that you've aligned to the desired boundary, and then save a pointer to that storage in another automatic variable. For example, to create a 16-byte aligned object (e.g., for SSE objects that require 16-byte alignment), you could do the following:

```

var
  ptrToAligned16:pointer to lword;
  .
  .
  .
sub( 16, esp );
and( $ffff_fff0, esp );
mov( esp, ptrToAligned16 );

```

The "`and($ffff_fff0, esp);`" instruction ensures that ESP is situated at an address that is aligned on a 16-byte boundary. Note that this instruction might actually allocate up to 15 additional bytes (or, if the stack is aligned properly to begin with, up to 12 additional bytes) in order to guarantee that the new address is aligned to a 16-byte boundary.

Whenever using this technique to allocate storage for an aligned object, you must allocate the storage before pushing any other data you need to retrieve onto the stack. Because you don't really know how much storage these three instructions will actually allocate on the stack, you won't know where any data might be that you've previously pushed onto the stack. As such, you wouldn't be able to pop that data later on. The best way to avoid this problem is to allocate aligned data immediately upon entry into a procedure, iterator, or method, before any other stack operations take place:

```

procedure hasAlignedStorage; @nostackalign;
var
  ptrToAligned16:pointer to lword;
begin hasAlignedStorage;

  sub( 16, esp );
  and( $ffff_fff0, esp );
  mov( esp, ptrToAligned16 );
  .
  .
  .

end hasAligned16;

```

Note the use of the `@nostackalign` option. If you're created aligned data on the stack, you're already aligning ESP to an address that is a multiple of four. Therefore, there is no need for HLA to emit this instruction for you.

If you are creating your own stack frame upon entry into the procedure (e.g., you're using the `@noframe` option), then you should allocate storage for your aligned objects after you've created the stack frame/activation record:

```

procedure hasAlignedStorage; @noframe;
var
    ptrToAligned16:pointer to lword;
begin hasAlignedStorage;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );
    sub( 16, esp );
    and( $ffff_fff0, esp );
    mov( esp, ptrToAligned16 );
    .
    .
    .

end hasAligned16;

```

Although this scheme allows you to allocate storage that is aligned on a 16-byte boundary (any other boundary that is a power of two is easy to achieve by modifying the `sub` and `and` instructions), this scheme isn't actually aligning the variables in the `var` section to a specific boundary. If you need to align the automatic variables themselves, it's going to take a bit more work to achieve. Achieving this goal requires that the stack be aligned to a given boundary *before* you call the subroutine. Unfortunately, you cannot simply align the stack pointer immediately upon entry into a subroutine, prior to building the activation record, because any parameters that the caller has pushed onto the stack must be accessible at fixed positions from EBP. If you align the stack upon entry into the code, you'll mess up the offsets to the parameters from EBP, thereby changing the assumptions HLA makes about where those parameters' values lie. Therefore, you must align the stack pointer to a desired address before pushing any parameters onto the stack. This means that the calling code will be responsible for aligning the stack and this has to be done on *each* call to the subroutine.

The task is to set up the stack pointer so that when it pushes EBP on the stack (while setting up the activation record) the address of the old EBP value on the stack is a multiple of whatever alignment you need. Unfortunately, you cannot simply align the stack pointer before the call because the subroutine's parameters, return address, and the EBP value itself consume space on the stack that may cause the alignment to change. Therefore, you will need to adjust to the stack pointer prior to the call so that ESP is aligned to an appropriate address after the caller has pushed the parameters, return address, and EBP has been preserved on the stack. For example, consider the following HLA procedure:

```

procedure p(i:int32);
var [16];
    b:byte;
    w:word;
    d:dword;
    l:lword;
begin p;
    .
    .
    .
end p;

```

The goal here is to align each of the automatic variables to an address that is a multiple of 16 bytes. Upon entry into the body of the procedure, there will be 12 bytes pushed onto the stack - four bytes for parameter *i*, four bytes for the return address, and four bytes with the old EBP value. Therefore, simply aligning ESP to some multiple of four before the call will not work because when the call to the procedure occurs, an additional 12 bytes wind up on the stack, leaving ESP misaligned. What has to be done is to align ESP to a multiple of 16 bytes and then drop the stack pointer down four bytes so that when the calling sequence pushes those 12 bytes onto the stack, ESP winds up properly aligned on a 16-byte boundary. This can be done with the following code sequence (that calls procedure *p*):

```
and( $FFFF_FFF0, esp ); // Align ESP to 16-byte boundary
sub( 4, esp );          // 4 + 12 bytes keeps it 16-byte aligned
p( 2 );                // Call p.
```

Alas, "4" is a magic number here that probably won't make much sense to the reader of this code. Furthermore, if you ever change the number or types of *p*'s parameters, "4" might no longer be the correct value to use here. Fortunately, HLA's compile-time language provides a compile-time function, **@parms**, that returns the number of parameter bytes for the procedure whose name you specify as an argument. So we can use the following generic version to properly align the stack on a 16-byte boundary:

```
and( $FFFF_FFF0, esp );
sub( 16 - ((@parms(p)+8) & $F), esp );
p( 2 );
```

The "**@parms(p)+8**" portion of the expression is the total number of bytes pushed on the stack up to the point where EBP will be pointing in the activation record. The "**((@parms(p)+8) & \$F)**" computes this value modulo 16 because we never need to push more than 15 bytes in order to align ESP to a 16-byte boundary. Finally, "**16 - ((@parms(p)+8) & \$F)**" computes the number of bytes we must drop the stack down in order to guarantee 16-byte alignment upon entry into the subroutine.

We could make one additional improvement to this code. On occasion, the expression "**16 - ((@parms(p)+8) & \$F)**" will evaluate to zero and there is no reason at all to execute the **sub** instruction. Because this is a constant expression, we can determine that it is zero at compile time and use conditional assembly to eliminate the **sub** instruction:

```
and( $FFFF_FFF0, esp );
#if( (16 - ((@parms(p)+8) & $F)) <> 0 )

    sub( 16 - ((@parms(p)+8) & $F), esp );

#endif
p( 2 );
```

Remember, you have to execute this instruction sequence before each call to procedure *p* in order to guarantee that *p*'s local variables are properly aligned on a 16-byte boundary. As it's easy to forget to execute this sequence prior to calling *p*, you might want to consider writing a macro to invoke that will automatically do this for you. Consider the following code:

```
#macro p( _i_ );
and( $FFFF_FFF0, esp );
#if( (16 - ((@parms(_p)+8) & $F)) <> 0 )

    sub( 16 - ((@parms(_p)+8) & $F), esp );

#endif
p( _i_ );
#endmacro
```

```

procedure _p(i:int32);
var [16];
    b:byte;
    w:word;
    d:dword;
    l:lword;
begin _p;
    .
    .
    .
end _p;
    .
    .
    .
p(2);

```

One problem with aligning the stack in this manner is that the code suffers from "stack creep". Each time you call procedure *p* it might drop the stack down as many as 12 bytes. If this isn't a problem (e.g., if you call *p* from within some other procedure that cleans up the stack upon returning to its caller) then you can ignore the stack creep. However, if you've pushed data onto the stack that you need to pop after the call to *p*, or if you're calling *p* within a loop and that would cause considerable stack creep, then you'll want to save ESP's value in a local (automatic) variable in the calling code and restore ESP upon return, e.g.,:

```

mov( esp, espSave );
p(2); // This is the macro from above!
mov( espSave, esp );

```

Be sure to use a local automatic, not a static, variable for *espSave*. Also, avoid the temptation to use **push** and **pop** to preserve ESP's value, remember that ESP is modified by the call to *p* and you won't be popping what you've pushed.

One last feature available in the **var** section is the ability to set the starting offset of the activation record. By default, HLA uses the offset zero as the base offset of the activation record. HLA assigns local (automatic) variables negative offsets from this base offset and parameters positive offsets from the base offset. Using the following syntax, you can change the base offset from zero to any other signed integer value you choose:

```

var:= <<signed integer expression>>;
    << var declaration section>>

```

The first local variable you declare in the << var declaration section >> will have the offset you specify by <<signed integer expression>>. *Note that HLA will not first subtract the size of the first object from your base offset as it normally does for the automatic variables you declare.* It uses the value you supply as the offset of the first variable you declare. Also note that this syntax does not change the offsets assigned to the parameters for the procedure. Therefore, EBP must point at the same location (at the old value of EBP immediately below the return address) it would if you didn't set the starting offset.

In general, this syntax is far more useful for **record** data structures than it is for **var** activation records, but it can be useful if you want to explicitly declare the saved EBP value and the display (if one is present). For example:

```

procedure p(i:int32);
var := 0;
    saveEBP:dword;
    display:dword[2];
    b          :byte;

```

```

        w          :word;
        d          :dword;
        l          :lword;
begin _p;
    .
    .
    .
end _p;

```

The var declaration section also supports the following (rarely-used) **@nostorage** syntax:

```

varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;

```

These declarations assign the current offset (after subtracting the size of the object) into the activation record to the variable you've declared, but they do not reserve any storage for such variables. As a result, variable declarations with the **@nostorage** option in the **var** section overlay the following variable declaration(s). Consider the following variable declarations:

```

var
    w:word;    @nostorage;
    b:byte;    @nostorage;
    d:dword;

```

The *b* variable will be sitting at offset -1, the *w* variable will be at offset -2, and the *d* variable will be sitting at offset -4. Note that these variables overlap one another in memory. Be very careful when using the **@nostorage** option in the **var** declaration section. If you declare a large object using the **@nostorage** option and you don't declare sufficient storage in variables after that object, accessing that object may wind up wiping data in "no man's land" on the stack.

Note that because offsets into the activation record are negative, the **@nostorage** option behaves differently in the **var** section from the way it works in the **static**, **storage**, and **readonly** sections. If you have a byte variable with an **@nostorage** option followed by a dword variable, the byte variable will be sitting in the H.O. byte of the dword object (rather than in the L.O. byte position, as it would in a **static**, **storage**, or **readonly** section). For this reason, you'll rarely see the **@nostorage** option used in a **var** section.

10.2.6 The HLA STATIC Declaration Section

The **static** section is where you declare static/data variables in an HLA **namespace**, **class**, **procedure**, **method**, **iterator**, or **program**. The basic syntax for an HLA **static** section is the following:

```

static
    << static variable declarations >>

or

static
    << static variable declarations >>
endstatic;

```

Each static variable declaration can take one of the following forms:

Uninitialized forms:

```

varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record <<record_field_declarations>> endrecord;
varID : union <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };

```

Initialized forms:

```

varID : typeID := <<constant expression>>;
varID : typeID [ list_of_array_dimensions ] := <<constant expression>>;
varID : procedure (<<optional_parameter_list>>) := <<constant
expression>>;
varID : pointer to typeID := <<constant expression>>;
varID : enum{ <<list_of_enumeration_identifiers>> } := <<constant
expression>>;

```

No allocation forms:

```

varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;

```

External forms:

```

varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;
varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );

```

The first set of declarations above creates non-initialized static variables. "Non-initialized" means that the program does not explicitly initialize these static variables before the program begins execution; in fact, the system initializes all non-initialized static objects to zero (or all zero bits) when the program loads into memory. Although you can safely assume that all non-initialized static variables contain zero bits, it's still wise to explicitly initialize static variables you expect to contain zero; leave the non-initialized forms of the static variable declaration for those variables whose initial value is completely irrelevant (e.g., because the program will initialize the value before ever using it).

Here are some examples of non-initialized variable declarations:

```

static
  i          :int32;
  user      :someUserType;
  ary       :char[ 3 ];
  usrAry    :someUserType[2];
  procPtr:procedure (cnt:uns32);
  quickRec:record
    a       :char;

```



```

        b :boolean;
        c :char;
    endrecord;
quickUn:union
    a :char;
    b :boolean;
    c :char;
endunion;
charPtr:pointer to char;
usrPtr :pointer to someUserType;
colors :enum{ red, green, blue };

```

Static variables also support initialization via some constant expression using the syntax from the second group of declarations given earlier. The type of the constant expression must be compatible with the type of the static variable you are declaring (that is, the type must match or HLA must be able to convert the constant's type to the specified type at compile time). Here are some examples:

```

type
    someUserType :record
        b:boolean;
        c:char;
        w:word;
        d:dword;
    endrecord;

procedure p( parameter:uns32 );
begin p;
    .
    .
    .
end p;

static
    i          :int32 := -4;
    user       :someUserType := someUserType:[ false, 'a', 0, 1];
    ary        :char[ 3 ] := ['a', 'b', 'c'];
    usrAry     :someUserType[2] :=
        [
            someUserType:[ false, 'a', 0, 1],
            someUserType:[ true, 'b', 1, 0]
        ];

    procPtr:procedure (cnt:uns32) := &p;
    charPtr:pointer to char := &ary[0];
    usrPtr :pointer to someUserType := &user;

```

You will notice that you cannot provide initializer constants for all of the static variable declarations. In particular, you cannot assign a constant to a static variable directly declared as a **record**, **union**, or **enum** object. However, as this last example demonstrates, this isn't a limitation because you can easily create a user-defined type that is a **record**, **union**, or **enum** type and use that type in a static variable declaration (e.g., as was done with *someUserType* in this example).

The third syntactical form (the "no allocation" forms) create a typed label in memory without explicitly allocating storage for that static variable. As a result, a variable with the **@nostorage** option will be sitting at the same memory location as the following variable(s) you declare in memory. Consider the following example:

```

static

```

```

b          :byte; @nostorage;
u          :uns32;@nostorage;
i          :int32;

```

The *b*, *u*, and *i* variables will all be sitting at the same starting address in memory; any modification to one of these variables will modify the others, as well. This declaration is almost equivalent to putting these three variables into a **union** data type.

The static declaration section also allows the declaration of external (and public) variables (see the "External Forms" syntax given earlier). Each external declaration can take one of two forms, one using the "external;" declaration and one using the "external("external_name");" declaration. For example:

```

static
  b          :byte; external;
  u          :uns32;external( "u_var" );

```

When **external** appears by itself, HLA will use the declared variable's name (e.g., *b* in this example) as the external name. Whenever you use the second form (with the string argument), HLA will use the declared name within the current source file and use the string name supplied as the external argument as the external name (e.g., *u_var* in place of *u* in this example).

When you compile and link your program, the system assumes that you have declared all external static variables in some other object module (that you link with the file containing the external declaration). The file containing the actual variable declaration must define the symbol as a *public* symbol. You create a public symbol by having both an external definition of the symbol and a regular declaration of that symbol, e.g.,

```

static
  b          :byte; external;
  b          :byte := 1;

  u          :uns32;external( "u_var" );
  u          :uns32 := 2;

```

All public and non-external variables in a **static** section will consume the corresponding amount of space in the executable program's disk file. This is true even if you don't explicitly assign a value to a **static** object using the initializer syntax. If you don't explicitly assign a value to a **static** variable, HLA will write a zero to the corresponding location on the disk. This is how the system initializes those variables when the program begins running: it simply copies the data from the disk file to the location in memory where the variable will be accessed. As non-initialized variables have zeros written to the disk file at their corresponding locations, the operating system will load those zeros into the memory locations reserved for such variables, thus initializing them to zero.

An HLA **static** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the static declaration section with an **endstatic** clause. Here's an example of such a declaration section:

```

procedure p;
begin p;

  .
  .
  .
  static
    b          :byte := 1;
    w          :word;
    d          :dword; external;

```

```

    endstatic;
    .
    .
    .
end p;

```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of static declaration section is to support variable declarations in macros that might need to declare static variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare static variables this way.

10.2.7 The HLA STORAGE Declaration Section

The **storage** section is where you declare uninitialized static variables in an HLA **namespace**, **class**, **procedure**, **method**, **iterator**, or **program**. The basic syntax for an HLA **storage** section is similar to that for **static** except you are not allowed to initialize any variables you declare in the **storage** section. The syntax is the following:

```

storage
    << storage variable declarations >>

or

storage
    << storage variable declarations >>
endstorage;

```

Each **storage** variable declaration can take one of the following forms:

Standard forms:

```

varID : typeID;
varID : typeID [ list_of_array_dimensions ];
varID : procedure (<<optional_parameter_list>>);
varID : record <<record_field_declarations>> endrecord;
varID : union <<union_field_declarations>> endunion;
varID : pointer to typeID;
varID : enum{ <<list_of_enumeration_identifiers>> };

```

No allocation forms:

```

varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;

```

External forms:

```

varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;

```

```

varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );

```

Other than you cannot assign an initial value to a **storage** variable, the declaration of the variables in the **storage** section is identical to that of the **static** section. Please see the discussion in the **static** section for more details.

As far as your program is concerned, **storage** variables are static objects exactly like variables you declare in a **static** section. The only real difference between variables you declare in a **storage** section and those you declare in a **static** section is that the disk file holding the program's data and code does not contain any data for the individual variables. Instead, HLA makes note of the number of bytes for all your **storage** variable declarations and stores this size in the object file it produces. When the operating system loads your program into memory, it makes note of this size and allocates a sufficient amount of space for these "BSS" (Block Started by a Symbol - an ancient assembly language term) variables and then writes zeros to that block of storage so that the variables are all initialized to zero when the program begins running. Although your program will take the same amount of storage in memory regardless of whether you declare your variables in the **storage** or **static** section, you may save some disk space in the executable file if you declare your uninitialized variables in the **storage** section rather than the **static** section.

An HLA **storage** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the static declaration section with an **endstorage** clause. Here's an example of such a declaration section:

```

procedure p;
begin p;

.
.
.
storage
    b          :byte;
    w          :word;
    d          :dword; external;
endstorage;
.
.
.
end p;

```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of storage declaration section is to support variable declarations in macros that might need to declare storage variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare storage variables this way.

10.2.8 The HLA READONLY Declaration Section

The **readonly** section is where you declare static read-only values in an HLA **namespace**, **class**, **procedure**, **method**, **iterator**, or **program**. The basic syntax for an HLA **readonly** section is the following:

```
readonly
```

```

    << readonly variable declarations >>

or

readonly
    << static variable declarations >>
endreadonly;

```

Each **readonly** object declaration can take one of the following forms:

Initialized forms:

```

varID : typeID := <<constant expression>>;
varID : typeID [ list_of_array_dimensions ] := <<constant expression>>;
varID : procedure (<<optional_parameter_list>>) := <<constant
expression>>;
varID : pointer to typeID := <<constant expression>>;
varID : enum{ <<list_of_enumeration_identifiers>> } := <<constant
expression>>;

```

No allocation forms:

```

varID : typeID; @nostorage;
varID : typeID [ list_of_array_dimensions ]; @nostorage;
varID : procedure (<<optional_parameter_list>>); @nostorage;
varID : pointer to typeID; @nostorage;
varID : enum{ <<list_of_enumeration_identifiers>> }; @nostorage;

```

External forms:

```

varID : typeID; external;
varID : typeID; external( "external_name" );
varID : typeID [ list_of_array_dimensions ]; external;
varID : typeID [ list_of_array_dimensions ]; external( "external_name" );
varID : procedure (<<optional_parameter_list>>); external;
varID : procedure (<<optional_parameter_list>>); external(
"external_name" );
varID : pointer to typeID; external;
varID : pointer to typeID; external( "external_name" );

```

Note that there are no non-initialized forms that have storage allocated for them. A **readonly** object must have an initializer attached to it, have the **@nostorage** attribute (in which case it inherits the initial value of the following **readonly** object you declare), or it must be an external declaration.

HLA places all objects you declare in a **readonly** section into memory that the operating system write protects. Any attempt to store data into a **readonly** object at run time will result in a segmentation/access violation fault. This is enforced by the operating system, not by HLA. You can create an instruction that will store data into a **readonly** object and HLA will compile the program just fine. When you try to run the program, however, it will generate an exception when you attempt to execute that instruction.

An HLA **readonly** declaration section can also appear in the body of a procedure or program. In such a case, you must explicitly terminate the **readonly** declaration section with an **endreadonly** clause. Here's an example of such a declaration section:

```

procedure p;
begin p;

```

```

.
.
.
readonly
  b      :byte := 1;
  w      :word := 2;
  d      :dword; external;
endreadonly;
.
.
.
end p;

```

As far as HLA is concerned, such declarations are treated as though you'd place them in the declaration section of that procedure (except you cannot access the variables until after they are declared). The main reason for allowing this type of **readonly** declaration section is to support variable declarations in macros that might need to declare **readonly** variables but are invoked within the body of the procedure. It would be unusual for you to explicitly declare **readonly** variables this way.

10.2.9 The HLA PROC Declaration Section

The **proc** section is where you declare "new style" procedures, iterators, and methods. The chapter on procedures goes into detail about the **proc** section, please see the discussion of the **proc** section in that chapter.

10.2.10 THE HLA NAMESPACE Declaration Section

HLA supports a special declaration section known as a **namespace**. A namespace is a collection of declarations that HLA gathers together under a single identifier (the **namespace** identifier). A namespace declaration uses the following syntax:

```

namespace userNamespaceID;

  << namespace declarations >>

end userNamespaceID;

```

userNamespaceID is an identifier you associate with the namespace declarations; you will use this identifier when referencing members of the namespace in your application. The body of the namespace, << *namespace declarations* >>, can be any of the following declaration sections:

```

const
val
type
static
readonly
storage
proc
old-style procedure, method, and iterator declarations

```

Note that **label** and **var** declaration sections are illegal in a namespace. In addition, you cannot nest **namespace** declarations (that is, **namespace** declarations are not legal in a **namespace**).

Namespace declarations should always appear an lex-level one in a **program** or **unit**. In HLA v2.x namespaces have a relatively kludged implementation and strange things might happen if you

declare namespaces within classes, procedures, methods, or iterators; namespace declarations have not been extensively tested in such cases and will probably fail to work properly.

An unusual feature about namespaces is that the **namespace** identifier does not have to be unique within its scope (that is, at lex level one). You can have multiple **namespace** declarations in a program with the same **namespace** identifier. HLA will simply combine these separate namespaces into a single unit. This is useful, for example, when you've got several different include files and each include file contains a common **namespace** declaration with the intent of constructing one big name space from the three separate ones. Although the **namespace** identifier need not be unique, all the declarations in a **namespace** with a given identifier must be unique. That is, you cannot declare two objects with the same name in a single name space.

One of the principle purposes of an HLA **namespace** is to prevent *name space pollution*. As your applications increase in size, and especially as you start to link in libraries of subroutines you (or other people) have created, it becomes difficult to avoid reusing names that other code is already using. For example, you might want to write a *put* macro or procedure to output data in some special way. However, *put* is a very common name (for example, the HLA Standard Library uses it) so you'd probably have to dream up a different name if you wanted to use this identifier. This is where name spaces come to the rescue. You can encapsulate every instance of the *put* identifier in a separate namespace and avoid the conflicts. For example, the HLA Standard Library uses the *put* identifier all over the place, but it's buried in the *stdout*, *stderr*, *fileio*, *str*, and other name spaces, so these identifiers don't conflict with one another.

To access an identifier that is a member of a namespace, you use the same *dot notation* that HLA uses for **record**, **union**, and **class** field access. To access a field from a **namespace** you specify the name space identifier, a period (dot), followed by the field name. For example, to invoke the *put* macro in the HLA Standard Library *stdout* namespace, you use the (very familiar) sequence *stdout.put*. If you create your own **namespace**, you simply substitute your name space identifier and the field name, e.g.,:

```
program nsDemo;

namespace myNamespace;

    static
        x:dword;

    procedure pp( p:dword );
    begin pp;
    end pp;

end myNamespace;

begin nsDemo;

    myNamespace.pp( myNamespace.x );

end nsDemo;
```

As noted above, namespaces in HLA have a somewhat kludged implementation. One artifact of this implementation is that within a **namespace** no global symbols (symbols declared outside the **namespace**) are directly visible. This includes some HLA-defined symbols (such as **true** and **false**) in addition to any symbols you've defined. If you need to reference any symbols defined outside the namespace within code (or expressions) inside the namespace, you will need to prepend the **@global:** string to the global symbol; otherwise, HLA will generate an *unknown symbol* error. Here is an example of using the **@global** modifier:

```
program nsDemo;
type
    array:byte[256];

namespace myNamespace;
```

```
static
    b:boolean := @global:true;

procedure pp( p:@global:array );
begin pp;
end pp;

end myNamespace;

static
    a :array;

begin nsDemo;

    myNamespace.pp( a );

end nsDemo;
```

A **namespace** declaration section may contain external- and forward-declared objects. Forward and public objects must be defined somewhere in the namespace within the current compilation, but you could have the external/forward definition in one instance of a particular namespace and the actual declaration of the object in another instance of that same namespace, e.g.,

```
program nsDemo;
namespace myNamespace;

    static
        b:boolean; external;

    procedure pp( p:dword ); forward;

end myNamespace;

// Assume some other code is here...

namespace myNamespace;

    static
        b:boolean;

    procedure pp( p:dword );
    begin pp;
    end pp;

end myNamespace;

begin nsDemo;
end nsDemo;
```

This example is rather trivial, but it's not hard to imagine a better one. The HLA Standard Library include files, for example, contain dozens of **namespace** declarations containing external entries. The actual source code for the HLA Standard Library contains the actual implementation within a **namespace** declaration section (in a **unit**).

One big advantage to using namespaces is that they improve HLA's compilation speed when dealing with a large number of symbols. Namespaces use a special symbol table lookup algorithm that is much faster than the standard symbol table lookup algorithms that HLA uses for symbols

defined outside a namespace. Using namespaces to encapsulate a large number of symbols can dramatically improve compile times. For example, the `w.hhf` header file (that encapsulates all of its identifiers in the `w` namespace) used to take about 45 seconds to process on a Pentium IV processor, prior to putting all the symbols into a namespace. After adding namespaces to HLA, the compile time was reduced to a couple of seconds. So if you're creating a large project with hundreds or thousands of data variables and other symbols, you might want to consider sticking those symbols into a namespace in order to reduce compilation time.

11 HLA Procedure Declarations and Procedure Calls

Note: this chapter discusses how HLA generates code for various procedure calls and procedure bodies. The examples given here should be treated as gross approximations only. Most of these examples come from early version of HLA v1.x and later versions have substantially improved the code generation. When in doubt, compile a test program with HLA emitting source code for your favorite assembler and view the output that HLA produces.

11.1 Procedure Declarations

HLA supports two different ways to declare procedures in a program: the "original style" ("old style") and the "new style". The "original style" was introduced in HLA v1.0; the "new style" of procedure declarations was introduced in HLA v2.0. Note that the "original style" (despite often being called the "old style") is not obsolete nor is it in danger of being deprecated. The original and new styles of procedure declarations are complementary. The new procedure declaration style simply adds some consistency and enhanced facilities to HLA.

11.1.1 Original Style Procedure Declarations

Original style procedure declarations are nearly identical to program declarations with two major differences: procedures are declared using the "procedure" reserved word and procedures may have parameters. The general syntax is:

```
procedure identifier ( optional_parameter_list ); procedure_options
    declarations
begin identifier;
    statements
end identifier;
```

Note that you may declare procedures inside other procedure in a fashion analogous to most block-structured languages (e.g., Pascal).

The optional parameter list consists of a list of **var**-type declarations taking the form:

```
optional_access_keyword identifier1 : identifier2 optional_in_reg
```

optional_access_keyword, if present, must be **val**, **var**, **valres**, **result**, **name**, or **lazy** and defines the parameter passing mechanism (pass by value, pass by reference, pass by value/result [or value/returned], pass by result, pass by name, or pass by lazy evaluation, respectively). The default is pass by value (**val**) if an access keyword is not present. For pass by value parameters, HLA allocates the specified number of bytes according to the size of that object in the activation record. For pass by reference, pass by value/result, and pass by result, HLA allocates four bytes to hold a pointer to the object. For pass by name and pass by lazy evaluation, HLA allocates eight bytes to hold a pointer to the associated thunk and a pointer to the thunk's execution environment (see the sections on parameters and thunks for more details).

The *optional_in_reg* clause, if present, corresponds to the phrase "in *reg*" where *reg* is one of the 80x86's general-purpose 8-, 16-, or 32-bit registers. You must take care when passing parameters through the registers as the parameter names become aliases for registers and this can create confusion when reading the code later (especially if, within a procedure with a register parameter, you call another procedure that uses that same register as a parameter).

HLA also allows a special parameter of the form:

```
var identifier : var
```

This creates an untyped reference parameter. You may specify any memory variable as the corresponding actual parameter and HLA will compute the address of that object and pass it on to the procedure without further type checking. Within the procedure, the parameter is given the **dword** type.

The *procedure_options* component above is a list of keywords that specify how HLA emits code for the procedure. There are several different procedure options available: `@noalignstack`, `@alignstack`, `@pascal`, `@stdcall`, `@cdecl`, `@align(int_const)`, `@use reg32`, `@leave`, `@noleave`, `@enter`, `@noenter`, and `@returns("text")`.

Option	Description
<code>@noframe</code> , <code>@frame</code>	By default, HLA emits code at the beginning of the procedure to construct a stack frame. The <code>@noframe</code> option disables this action. The <code>@frame</code> option tells HLA to emit code for a particular procedure if stack frame generation is off by default. HLA also uses these two special reserved words as a compile-time variable to set the default frame generation for all procedures. Setting <code>@frame</code> to true (or <code>@noframe</code> to false) turns on frame generation by default; setting <code>@frame</code> to false (or <code>@noframe</code> to true) turns off frame generation.
<code>@nodisplay</code> , <code>@display</code>	By default, HLA emits code at the beginning of the procedure to construct a display within the frame. The <code>@nodisplay</code> option disables this action. The <code>@display</code> option tells HLA to emit code to generate a display for a particular procedure if display generation is off by default. Note that HLA does not emit code to construct the display if <code>@noframe</code> is in effect, though it will assume that the programmer will construct this display themselves. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting <code>@display</code> to true (or <code>@nodisplay</code> to false) turns on display generation by default; setting <code>@display</code> to false (or <code>@nodisplay</code> to true) turns off display generation.
<code>@noalignstack</code> , <code>@alignstack</code>	By default (assuming <code>@frame</code> generation is active), HLA will emit an instruction to align ESP on a four-byte boundary after allocating local variables. Win32, *NIX, and other 32-bit OSes require the stack to be double-word-aligned (hence this option). If you know the stack will be double-word-aligned, you can eliminate this extra instruction by specifying the <code>@noalignstack</code> option. Conversely, you can force the generation of this instruction by specifying the <code>@alignstack</code> procedure option. HLA also uses these two special identifiers as a compile-time variable to set the default display generation for all procedures. Setting <code>@alignstack</code> to true (or <code>@noalignstack</code> to false) turns on stack alignment generation by default; setting <code>@alignstack</code> to false (or <code>@noalignstack</code> to true) turns off stack alignment code generation.

<p><code>@pascal</code>, <code>@cdecl</code>, <code>@stdcall</code></p>	<p>These options give you the ability to specify the parameter passing mechanism for procedures. By default, HLA uses the <code>@pascal</code> calling sequence for all procedures. This calling sequence pushes the parameters on the stack in a left-to-right order (i.e., in the order they appear in the parameter list). It also automatically cleans up the stack upon return from the procedure. The <code>@cdecl</code> procedure option tells HLA to pass the parameters from right-to-left so that the first parameter appears at the lowest address in memory and it is the user's responsibility to remove the parameters from the stack upon return from the procedure. The <code>@stdcall</code> procedure option is a hybrid of the <code>@pascal</code> and <code>@cdecl</code> calling conventions. It pushes the parameters in the right-to-left order (just like <code>@cdecl</code>) but <code>@stdcall</code> procedures automatically remove their parameter data from the stack (just like <code>@pascal</code>). Win32 API calls use the <code>@stdcall</code> calling convention.</p> <p>Note that iterators and methods always use the Pascal calling convention; you may only apply the <code>@cdecl</code> and <code>@stdcall</code> options to HLA procedures.</p>
<p><code>@align(int_constant)</code></p>	<p>The <code>@align(<i>int_constant</i>)</code> procedure option aligns the procedure on a 1, 2, 4, 8, or 16 byte boundary. Specify the boundary you desire as the parameter to this option. The default is <code>@align(1)</code>, which is unaligned; HLA also uses this special identifier as a compile-time variable to set the default procedure alignment. Setting <code>@align := 1</code> turns off procedure alignment while supplying some other value (which must be a power of two) sets the default procedure alignment to the specified number of bytes.</p>
<p><code>@use reg₃₂</code></p>	<p>When passing parameters, HLA can sometimes generate better code if it has a 32-bit general purpose register for use as a scratchpad register. By default, HLA never modifies the value of a register behind your back; so, it will often generate less than optimal code when passing certain parameters on the stack. By using the <code>@use</code> procedure option, you can specify one of the following 32-bit registers for use by HLA: EAX, EBX, ECX, EDX, ESI, or EDI. By providing one of these registers, HLA may be able to generate significantly better code when passing certain parameters.</p>
<p><code>@returns("text")</code></p>	<p>This option specifies the compile-time return value whenever a function name appears as an instruction operand. For example, suppose you are writing a function that returns its result in EAX. You should probably specify a "returns" value of "EAX" so you can compose that procedure just like any other HLA machine instruction (see the example below and the section on machine instructions for more details).</p>

<p><code>@leave</code>, <code>@noleave</code></p>	<p>These two options control the code generation for the standard exit sequence. If you specify the <code>@leave</code> option then HLA emits the x86 <code>leave</code> instruction to clean up the activation record before the procedure returns. If you specify the <code>@noleave</code> option, then HLA emits the primitive instructions to achieve this, e.g.,</p> <pre>mov(ebp, esp); pop(ebp);</pre> <p>The manual sequence is faster on some architectures, the <code>leave</code> instruction is always shorter.</p> <p>Note that <code>@noleave</code> occurs by default if you've specified <code>@noframe</code>. By default, HLA assumes <code>@noleave</code> but you may change the default using these special identifiers as a compile-time variable to set the default leave generation for all procedures. Setting <code>@leave</code> to true (or <code>@noleave</code> to false) turns on leave generation by default; setting <code>@leave</code> to false (or <code>@noleave</code> to true) turns off the use of the leave instruction.</p>
<p><code>@enter</code>, <code>@noenter</code></p>	<p>These two options control the code generation for a procedure's standard entry sequence. If you specify the <code>@enter</code> option then HLA emits the x86 <code>enter</code> instruction to create the activation record. If you specify the <code>@noenter</code> option, then HLA emits the primitive instructions to achieve this.</p> <p>The manual sequence is always faster, using the <code>enter</code> instruction is usually shorter.</p> <p>Note that <code>@noenter</code> occurs by default if you've specified <code>@noframe</code>. By default, HLA assumes <code>@noenter</code> but you may change the default using these special identifiers as a compile-time variable to set the default enter generation for all procedures. Setting <code>@enter</code> to true (or <code>@noenter</code> to false) turns on enter generation by default; setting <code>@enter</code> to false (or <code>@noenter</code> to true) turns off the use of the enter instruction.</p>

The following example demonstrates how the `@returns` option works:

```
program returnsDemo;
#include( "stdio.hhf" );

procedure eax0; @returns( "eax" );
begin eax0;

    mov( 0, eax );

end eax0;

begin returnsDemo;

    mov( eax0(), ebx );
```

```

    stdout.put( "ebx=", ebx, nl );

end returnsDemo;

```

11.1.2 "New Style" Procedure Declarations

HLA v2.0 added a new form of procedure declaration to make the syntax of procedure declarations more consistent with the other declaration sections (i.e., **const**, **val**, **type**, **var**, **static**, **readonly**, and **storage**). The new syntax uses the **proc** keyword to begin a procedure declaration section, e.g.,

```

proc
  << procedure declarations using new style syntax>>

```

You may optionally end a **proc** section with an **endproc** clause:

```

proc
  << procedure declarations using new style syntax>>
endproc;

```

A **procedure**, **iterator**, or **method** declaration appearing in a **proc** section is declared using one of the following forms:

```

identifier : procedure_type;
begin identifier;
  <<procedure body>>
end identifier;

identifier : procedure_type procedure_options;
begin identifier;
  <<procedure body>>
end identifier;

```

identifier is the name of the procedure you're declaring. This is a standard HLA identifier.

procedure_type is either a predefined (in the type section) procedure type or the reserved word **procedure** followed by an optional parameter list. Here are some examples of procedure declarations using both schemes:

```

type
  proc_t : procedure( i:int32; u:uns32 );

proc
  proc1:proc_t;
  begin proc1;
    <<procedure body>>
  end proc1;

  proc2:procedure( a:char );
  begin proc2;
    <<procedure body>>
  end proc2;
endproc;

```

The advantage of using a procedure type identifier to capture the parameter list is especially evident when defining several **forward** or **external** procedure declarations in a header file. If you have several procedures that all have the same parameter list (e.g., Win32 *winproc* type procedures), you can save a lot of typing by specifying a generic procedure type rather than repeating the same parameter list over and over again in the procedure declarations (using the original style declarations).

The syntax for procedure options in the new style procedure declarations is slightly different from the original style. Procedure options, if present, appear after the procedure type (or parameter list) and are surrounded by a pair of braces; they are separated by spaces or commas rather than terminated by semicolons. Here are some examples:

```

type
    proc_t : procedure( i:int32; u:uns32);

proc
    procx:proc_t; external;
    procy:procedure( d:dword ) { @returns( "@c" )}; forward;

    procl :proc_t {@noframe, @nodisplay};
    begin procl;
        <<procedure body>>
    end procl;

    proc2:procedure( var a:char )
        {@cdecl, @use eax, @returns( "eax" ), @noframe};
    begin proc2;
        <<procedure body>>
    end proc2;

```

Note that an original style **procedure** (or **iterator** or **method**) declaration will terminate a **proc** section. If you want to add some additional new style procedure declarations after an original style declaration, you will have to begin a new **proc** section by supplying another **proc** keyword definition.

HLA v1.x provided the ability to create a pointer to a procedure using syntax like the following:

```

procedure someProc( i:int32; u:uns32);
begin someProc;
    << procedure body >>
end someProc;
.
.
.
type
    someProc_t :pointer to someProc;

static
    ptrToSomeProc:someProc_t;

```

Objects of *someProc_t* will be pointers to procedures have two parameters (**int32** and **uns32**, as per the original *someProc* declaration). This syntax has always been a kludge (it was added at the request of an early HLA user) but the original HLA procedure declaration syntax really didn't allow anything that was better. Basing a type on an instance of an existing procedure declaration is ugly syntax. The better solution is to do the converse, base the procedure declaration on some procedure type. This is what the new procedure declaration syntax does.

```

type
    someProc_t :procedure( i:int32; u:uns32);

proc
    someProc :someProc_t;
begin someProc;
    << procedure body >>
end someProc;
.
.
.

static
    ptrToSomeProc:someProc_t;

```

11.2 Overloaded Procedure/Iterator/Method Declarations

Starting with HLA v2.5, the HLA language supports an **overloads** declaration in the **proc** declaration section. In previous versions of HLA it was possible to use the **overload** macro to create overloaded procedures, but that macro has some serious limitations. The new **overloads** declaration lifts many of the restrictions of the **overload** macro.

Because the **overloads** declaration replaces the functionality of the **overload** macro, the macro was moved out of the `hla.hhf` header file and into its own `overload.hhf` header file. The `stdlib.hhf` header file does not automatically include `overload.hhf`; therefore, if you are using the **overload** macro in your programs you will need to explicitly include the `overload.hhf` header file to compile without error. Better yet, convert your existing code to use the new **overloads** declaration.

Overloaded functions allows you to call a function (procedure, method, or iterator) based on a calling *signature* rather than just by the function's name. A call signature consists of the function's name and the number and types of its parameter list. Several functions can share the same name but have different signatures based on their parameter lists. For example, consider a **put** function with the following signatures:

```

procedure put( u:ins32 );
procedure put( i:int32 );
procedure put( c:char );

```

If you call **put** with an **uns32** parameter, it will invoke the version of the procedure that has an **uns32** argument; likewise, if you pass an **int32** or **char** parameter, the call to **put** will invoke the corresponding procedure.

The number of parameters also affects the function's signature; consider an extension of the **put** procedure signatures:

```

procedure put( u:ins32 );
procedure put( u:ins32; size:int32 );
procedure put( i:int32 );
procedure put( i:int32; size:int32 );
procedure put( c:char );
procedure put( c:char; size:int32 );

```

An invocation of the form `"put(u32Var);"` will call the first version of **put**; an invocation of the form `"put(u32Var, width);"` will call the second version above (assuming `u32Var` is an **uns32** object and `width` is an **int32** object).

Unlike some HLLs, HLA doesn't allow you to declare multiple procedures with the same name (but different signatures). The problem with that approach is that HLA has to generate internal ("mangled") names and this creates problems when you try to link overloaded procedures with other code (particularly if it isn't written in that same language). Instead, HLA's overloads declaration takes a more reasonable approach where you define the names of the individual functions (using standard procedure declarations with unique function names) and then use a set of overloads declarations to assign different signatures (with the same name) to these functions.

The **overloads** declaration appears in a **proc** section and uses the following basic syntax:

```
ovldName : overloads functionName;
ovldName : overloads functionName ( "string calling sequence" );
```

Here are a couple of overload example declarations:

```
procedure proc1( i:int32 ); external;
procedure proc2( i:int32; j:int32); external;
procedure proc3( i:int32; j:int32; k:int32); external;

proc
  prc : overloads proc1;
  prc : overloads proc2;
  prc : overloads proc3;
```

In this example, *prc* is the overloaded procedure name. Invoking *prc* will call *proc1*, *proc2*, or *proc3*, depending on the complete invocation signature. Note that you don't actually create a procedure named *prc*. The **overloads** declaration tells HLA that *ovldName* overloads the *functionName* in the overloads declaration.

In the **overloads** declaration, *functionName* must be the name of an already-defined **procedure**, **method**, or **iterator**. This can be a **forward**, **external**, or actual **procedure/iterator/method** declaration. Note that you cannot specify a procedure type name here; only actual **procedure/method/iterator** names are legal (as in the example above).

The overloaded name (e.g., *prc* in the example above) must be either an undefined identifier or an existing **overloads** name. **Overloads** identifiers follow all the usual HLA scoping rules with one extension: if you declare an **overloads** identifier inside a **procedure**, **method**, or **iterator** and that identifier is an overloads identifier outside that function, then the local declaration will extend the global declaration, not replace it. For example, consider this code:

```
program example;
  static
    i:int32;
    u:int32;

  proc
    proc1 :procedure( i:int32 ); external;
    proc2 :procedure ( u:uns32 ); external;

    prc : overloads proc1;

  procedure local;
  proc
    prc :overloads proc2;
  begin local;

    prc( i ); // Calls proc1
    prc( u ); // Calls proc2

  end local;
```

```

begin example;

    prc( i ); // calls proc1
    prc( u ); // Illegal, no valid signature for this call

end example;

```

Note how the signature that has an **uns32** argument is visible only inside *local*. Also note that the *prc* signature defined outside *local* is also usable inside *local*, despite the local declaration of *prc* inside the *local* procedure. Outside the *local* procedure, *prc* is still valid but the signature that has an **uns32** argument is no longer visible.

Note that signatures do not have to be unique. Consider the following declarations:

```

procedure proc1( i:int32 ); external;
procedure proc2( i:int32 ); external;

proc
    prc : overloads proc1;
    prc : overloads proc2;

```

Although *proc1* and *proc2* have the same argument list (that is, the number of parameters and types of the parameters match), the **overloads** declarations are legal. This, however, creates an ambiguity: if you invoke "prc(int32Var);" how will HLA differentiate the two calls? The answer is simple: HLA uses the last **overloads** declaration to disambiguate the declarations. This might seem confusing and a poor design decision, but this was a conscious design decision. To understand why HLA does this, consider the following example:

```

program example;
    static
        i:int32;

    proc
        proc1 :procedure( i:int32 ); external;
        proc2 :procedure ( i:int32); external;

        prc : overloads proc1;

    procedure local;
    proc
        prc :overloads proc2;
    begin local;

        prc( i ); // Calls proc2

    end local;

begin example;

    prc( i ); // calls proc1

end example;

```

Within a function, a local declaration of an **overloads** identifier can hide a global invocation that has the same signature. As the above example demonstrates, the global declaration is visible again beyond the body of the local function.

When you invoke an **overloads** function, HLA applies a multi-step signature-matching algorithm to determine which of the overloaded functions to call. First, the number of parameters must exactly agree with some **overloads** declaration or HLA will reject the invocation.

HLA compares the invocation parameter list against the list of **overloads** declaration. It scans the list backwards from the last **overloads** declaration (for a given **overloads** identifier) through to the first identifier. If an exact match to the parameter's types is found, then HLA will call that **procedure, method, or iterator**.

If HLA scans through the complete list of overloaded function names and doesn't come up with an exact signature match, then it will make a second pass through the list and relax the parameter matching requirements to see if a match is possible. The relaxation takes the following form:

Smaller `unsXX` types are "promoted" to larger types (e.g., an **uns8** actual parameter can be passed to a function with an **uns32** formal parameter).

Hexadecimal types (byte, word, dword, qword, tbyte, and lword) are compatible with all integer and unsigned types of the same size.

Note that the x86's registers are hexadecimal types.

Dwords are compatible with pointers.

Pointer constants are compatible with strings.

Because relaxing the type checking can produce ambiguity, the "last overloads to first overloads" disambiguation rule applies. Therefore, you should declare higher priority **overloads** declarations last in the **proc** section.

The overloads declaration takes two forms:

```
ovldName : overloads functionName;
ovldName : overloads functionName ( "string calling sequence" );
```

When you use the first form (as in all the examples to this point), invoking the overloaded name (*ovldName*) substitutes the actual function name (*functionName*) whose parameter list matches the actual call. That is, given a declaration of the form:

```
prc : overloads procl; // Assume: procedure procl( u:uns32 );
```

And given the invocation:

```
prc( uns32Var );
```

HLA will generate the following actual procedure call:

```
procl( uns32Var );
```

In certain circumstances, calling the function using the declared function name is insufficient. For example, if you have a class variable (an object), then you'll actually need to invoke the procedure using the actual object name, not by simply invoking the class procedure, method, or iterator name. The second form of the overloads declaration allows you to specify a string that HLA will expand when calling the actual function. Consider the following declarations inside a namespace:

```
namespace ns;

procedure a( u:uns32 );
begin u;
  //...
end u;

procedure b( i:int32 );
```

```

begin b;
    //...
end b;

proc
    ab:overloads a;
    ab:overloads b;

end ns;

```

This set of overloads declarations will not work outside the namespace. You would like to be able to invoke *ab* using "ns.ab(uns32Var);" or "ns.ab(int32Var);" but this will not work outside the namespace because an invocation of the form "ns.ab(uns32Var);" produces a call of the form "a(uns32Var);". Of course, outside the namespace this will either generate a syntax error (symbol not found or mismatched parameter list) or it will call the wrong procedure (a procedure named *a* outside the namespace). The solution is to use the second declaration form and explicitly specify the namespace name, as follows:

```

namespace ns;

    procedure a( u:uns32 );
    begin u;
        //...
    end u;

    procedure b( i:int32 );
    begin b;
        //...
    end b;

    proc
        ab:overloads a ( "@global:ns.a" );
        ab:overloads b ( "@global:ns.b" );

    end ns;

```

The "@global:" prefix exists in case you invoke the *ns.ab* overloaded procedure name inside another namespace.

Object references are another matter altogether. Consider the following declarations:

```

type
    class c;

        procedure a( u:uns32 );
        begin u;
            //...
        end u;

        procedure b( i:int32 );
        begin b;
            //...
        end b;

    proc
        ab:overloads a;
        ab:overloads b;
    end c;

```

```

    endclass;

static
    cv :c;

```

An invocation of the form "cv.ab(uns32Var);" produces the call "a(uns32Var);" which does not call the class procedure. Unfortunately, supplying a string operand of the form "c.a" in the **overloads** declaration will not solve the problem. Because a class procedure can be called using several different object variables (and, in the case of class procedures, using the class name), you have to supply the actual object name as part of the invocation string. Fortunately, HLA supplies a compile-time function, **@curVar**, which returns a string containing the current full variable name (including the overloaded name). The following declaration shows how to use **@curVar** to achieve this:

```

type
    c:class

        procedure a(u:uns32); external;
        procedure b(i:int32); external;

    proc
        ab :overloads a
        (
            "@text( @substr( @curvar, 0, @length( @curvar )-2)" +
              ""a" )"
        );
        ab :overloads b
        (
            "@text( @substr( @curvar, 0, @length( @curvar )-2)" +
              ""b" )"
        );
    endclass;

```

With this class declaration, an invocation of the form

```
cv.ab(uns32Var);
```

produces the call

```
@text(@substr( @curvar, 0, @length( @curvar )-2)+"a")(uns32Var);
```

which expands to

```
cv.a(uns32Var);
```

Note that **@curvar** in this example returns the string "cv.ab". The **@substring** function strips the last two characters from the string (the name of the overloaded function) and then appends "a" or "b" to the string to produce a call to the appropriate function. Because this string is a bit unwieldy, the HLA Standard Library's *hla.hhf* module includes a macro named "ovrldStr" that you can use to generate this text for you. This macro requires a single argument that is the name of the function you want to overload (a or b in this example). Here's what the above code would look like when using this macro:

```
type
```

```

c:class

    procedure a(u:uns32); external;
    procedure b(i:int32); external;

    proc
        ab :overloads a( hla.ovrldStr( a ));
        ab :overloads b( hla.ovrldStr( b ));
    endproc;

endclass;

```

Of course, you must `#include` the `hla.hhf` header file in order to use this macro.

For those who are interested in how things work internally, the `hla.ovrldStr` macro takes the following form:

```

#macro ovrldStr( string theFunc );

    "@text( @left( @curvar, @rindex( @curvar, 0, "."")+1) +
        "" + theFunc +"" )"

#endmacro

```

Because of the limitations of HLA's implementation languages (Flex and Bison), there are some limitations to how well HLA can recognize various parameter signatures. These problems will be corrected in HLA v3.0. In the meantime, if you run into any problems, you can easily work around the issue(s) by directly calling the procedure or by explicitly type-casting the actual arguments.

11.3 The `_vars_` and `_parms_` Constants and the `_display_` Array

To help those who insist on constructing the activation record themselves, HLA declares two local constants within each procedure: `_vars_` and `_parms_`. The `_vars_` symbol is an integer constant that specifies the number of bytes of local variables declared in the procedure. This constant is useful when allocating storage for your local variables. The `_parms_` constant specifies the number of bytes of parameters. You would normally supply this constant as the parameter to a `ret()` instruction to automatically clean up the procedure's parameters when it returns.

Example:

```

procedure demoVarsParms( parm1:int32; parm2:dword ); @nodisplay; @noframe;
var
    var1:dword;
    var2:dword;
begin demoVarsParms;

    push( ebp );
    mov( esp, ebp );
    sub( _vars_, esp );
    .
    .
    .
    mov( ebp, esp );
    pop( ebp );
    ret( _parms_ );

end demoVarsParms;

```

If you do not specify `@nodisplay`, then HLA defines a run-time variable named `_display_` that is an array of pointers to activation records. For more details on the `_display_` variable, see the section on lexical scope.

11.4 External Procedure Declarations

You can declare external procedures (procedures defined in other HLA units or written in languages other than HLA) using the following syntaxes:

Original Syntax:

```
procedure externProc1 (optional parameters) ; options; external;

procedure externProc2 (optional parameters) ; options; external (
"external_name" );
```

New Style Syntax:

```
proc
  externProc1 :procedure(optional parameters) {options}; external;
  externProc2 :procedure(optional parameters) {options}; external (
"external_name" );
endproc;
```

As with normal procedure declarations, the parameter list and the procedure options are optional. Note that `@external` and `external` are synonyms. `@external` is deprecated so you should use `external` in new code.

The first form is generally used for HLA-written functions. HLA will use the procedure's name (`externProc1` in this case) as external name.

The second form lets you refer to the procedure by one name within your HLA program (`externProc2` in this case) and by a different name ("`external_name`" in this example) in the generated object code. This second form has two main uses: (1) if you choose an external procedure name that just happens to conflict with a back-end assembler reserved word, the program may compile correctly but fail to assemble. Changing the external name to something else solves this problem. (2) When calling procedures written in external languages you may need to specify characters that are not legal in HLA identifiers. For example, Win32 API calls often use names like `WriteFile@24` containing illegal (in HLA) identifier symbols. The string operand to the external option lets you specify any name you choose. Of course, it is your responsibility to see to it that you use identifiers that are compatible with the external environment; HLA doesn't check these names.

External procedure declarations do not allow the same set of procedure options as regular procedure declarations. In particular, only those options that affect the calling sequence (rather than affecting the code generation within the procedure itself) are legal in an external declaration. The procedure options that are legal within an external procedure declaration are:

```
@use Reg32
@returns( "string" )
@cdecl
@pascal
@stdcall
```

If you declare an external procedure and then declare that same procedure in the same source file, that procedure name becomes *public* and is accessible to source files outside the current file. When declaring the public procedure body (after the appearance of the external procedure

declaration prototype earlier in the source file), the legal procedure options are those that affect the generation of code within the procedure; those that specify how HLA generates code to call the procedure are illegal (that is, the set of procedure options between the external declaration and the procedure's actual declaration are mutually exclusive. If an external procedure declaration appears in a source file, then the actual procedure declaration's procedure options are limited to the following:

```
@noframe, @frame
@nodisplay, @display
@noalignstack, @alignstack
@align( int_constant )
@leave, @noleave
@enter, @noenter
```

11.5 Forward Procedure Declarations

A forward procedure declaration provides a way to declare a procedure *prototype* that allows you to specify the calling syntax for a procedure before actually declaring the body of that procedure. In HLA, the syntax for a forward declaration (procedure prototype) is the following:

Original Syntax:

```
procedure forwardProc (optional parameters) ; options; forward;
```

New Style Syntax:

```
proc
  forwardProc :procedure(optional parameters) {options}; forward;
```

The forward declaration syntax is necessary because HLA requires all procedure symbols to be declared before they are used. In a few rare cases (where mutual recursion occurs between two or more procedures), it may be impossible to write your code such that every procedure is declared before the first call to the code. More commonly, sorting your procedures to ensure that all procedures are written before their first call may force an artificial organization on the source file, making it harder to read. The forward procedure declaration handles this situation for you. It lets you create a procedure prototype that describes how the procedure is to be called without actually specifying the procedure body. Later on in the source file, the full procedure declaration must appear.

Note: an external declaration also serves as a forward declaration. If you have an external definition at the beginning of your program (perhaps it appears in an include file), you do not need to provide a forward declaration as well.

As for external procedure declarations, forward declarations do not allow the same set of procedure options as regular procedure declarations. In particular, only those options that affect the calling sequence (rather than affecting the code generation within the procedure itself) are legal in a forward declaration. The procedure options that are legal within a forward procedure declaration are:

```
@use Reg32
@returns( "string" )
@cdecl
@pascal
@stdcall
```

When the procedure declaration appears later in the source file, the legal procedure options are


```

@noframe, @frame
@nodisplay, @display
@noalignstack, @alignstack
@align( int_constant )
@leave, @noleave
@enter, @noenter

```

Note that the presence of a forward declaration in a source file does not make the procedure name public. Also, note that you may not have both a forward and external procedure declaration for the same procedure name.

11.6 Setting Default Procedure Options

By default, HLA does the following:

- Creates a display for every procedure
- Emits code to construct the stack frame for each procedure
- Emits code to align ESP on a four-byte boundary upon procedure entry
- Assumes that it cannot modify any register values when passing (non-register) parameters
- The first instruction of the procedure is unaligned.

These options are the most general and "safest" for beginning assembly language programmers. However, the code HLA generates for this general case may not be as compact or as fast as is possible in a specific case. For example, few procedures will actually need a display data structure built upon procedure activation. Therefore, the code that HLA emits to build the display can reduce the efficiency of the program. Advanced programmers, of course, can use procedure options like **@nodisplay** to tell HLA to skip the generation of this code. However, if a program contains many procedures and none of them requires a display, continually adding the **@nodisplay** option can get annoying. Therefore, HLA allows you to treat these directives as "pseudo-compile-time-variables" to control the default code generation. E.g.,

```

?@display := true; // Turns on default display generation.
?@display := false; // Turns off default display generation.
?@nodisplay := true; // Turns off default display generation.
?@nodisplay := false; // Turns on default display generation.

?@frame := true; // Turns on default frame generation.
?@frame := false; // Turns off default frame generation.
?@noframe := true; // Turns off default frame generation.
?@noframe := false; // Turns on default frame generation.

?@alignstack := true; // Turns on default stk alignment code generation.
?@alignstack := false; // Turns off default stk alignment code generation.
?@noalignstack := true; // Turns off default stk alignment generation.
?@noalignstack := false; // Turns on default stk alignment generation.

?@enter := true; // Turns on default ENTER code generation.
?@enter := false; // Turns off default ENTER code generation.
?@noenter := true; // Turns off default ENTER code generation.
?@noenter := false; // Turns on default ENTER code generation.

?@leave := true; // Turns on default LEAVE code generation.
?@leave := false; // Turns off default LEAVE code generation.
?@noleave := true; // Turns off default LEAVE code generation.
?@noleave := false; // Turns on default LEAVE code generation.

```

```
?@align := 1; // Turns off procedure alignment (align on byte boundary).
?@align := int_expr; // Sets alignment, must be a power of two.
```

These directives may appear anywhere in the source file. They set the internal HLA default values and all procedure declarations following one of these assignments (up to the next, corresponding assignment) use the specified code generation option(s). Note that you can override these defaults by using the corresponding procedure options mentioned earlier.

11.7 Disabling HLA's Automatic Code Generation for Procedures

Before jumping in and describing how to use the high-level HLA features for procedures, the best place to start is with a discussion of how to disable these features and write "plain old fashioned" assembly language code. This discussion is important because procedures are the one place where HLA automatically generates a lot of code for you and many assembly language programmers prefer to control their own destinies; they don't want the compiler to generate any excess code for them. So disabling HLA's automatic code generation capabilities is a good place to start.

By default, HLA automatically emits code at the beginning of each procedure to do five things:

- (1) Preserve the pointer to the previous activation record (EBP);
- (2) build a display in the current activation record;
- (3) allocate storage for local variables;
- (4) load EBP with the base address of the current activation record;
- (5) adjust the stack pointer (downwards) so that it points at a double-word-aligned address.

When you return from a procedure, by default HLA will deallocate the local storage and return, removing any parameters from the stack.

To understand the code that HLA emits, consider the following simple procedure:

```
procedure p( j:int32 );
var
    i:int32;
begin p;
end p;
```

Here is a dump of the symbol table that HLA creates for procedure p:

```
p          <0,proc>:Procedure type (ID=p_hla_1) parms:4
-----
_vars_     <1,cons>:uns32, (4 bytes) =4
i          <1,var >:int32, (4 bytes, ofs:-12)
_parms_   <1,cons>:uns32, (4 bytes) =4
_display_ <1,var >:dword, (8 bytes, ofs:-4)
j         <1,valp>:int32, (4 bytes, ofs:8)
_finalize_ <1,val >:string, (0 bytes) =""
_initialize_ <1,val >:string, (0 bytes) =""
p         <1,proc>:
-----
```

The important thing to note here is that local variable *i* is at offset -12 and HLA automatically created an 8-byte local variable named *_display_* which is at offset -4 (note: the *_display_* variable uses negative indexes, so the two 4-byte elements are at offset -4 [index 0] and -8 [index -1]).

HLA, with the "-hla" and "-source" command-line parameters, produces the following pseudo-HLA code for the procedure above (annotations in italics are not emitted by HLA, this output is subject to changes in HLA code generation algorithms; the output is pure assembly language with no "hidden" or high-level code):

```

procedure p__hla_2;
begin p__hla_2;
    push( ebp );;Dynamic link (pointer to previous activation record)
    push( [ebp-4] );;Display for lex level 0
    lea( [esp+4], ebp );;Get frame ptr (point EBP at current
activation record)
    push( ebp );;Ptr to this proc's A.R. (part of display
construction)
    sub( 4, esp );;Local storage.
    and( -4, esp );;dword-align stack

xp__hla_2__hla_:
    mov( ebp, esp );;Deallocate local variables.
    pop( ebp );;Restore pointer to previous activation record.
    ret( 4 );;Return, popping parameters from the stack.
end p__hla_2;

```

Building the display data structure is not very common in standard assembly language programs. This is only necessary if you are using nested procedures and those nested procedures need to access non-local variables. Since this is a rare situation, many programmers will immediately want to tell HLA to stop emitting the code to generate the display. This is easily accomplished by adding the `@nodisplay` procedure option to the procedure declaration. Adding this option to procedure `p` produces the following:

```

procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
end p;

```

Compiling this procedure produces the following symbol table dump:

Symbol Table:

```

p                <0,proc>:Procedure type (ID=p__hla_1) parms:4
-----
_ vars_          <1,cons>:uns32, (4 bytes) =4
i                <1,var >:int32, (4 bytes, ofs:-4)
_ parms_        <1,cons>:uns32, (4 bytes) =4
j                <1,valp>:int32, (4 bytes, ofs:8)
_ finalize_     <1,val >:string, (0 bytes) =""
_ initialize_   <1,val >:string, (0 bytes) =""
p                <1,proc>:
-----

```

Note that the `_display_` variable is gone and the local variable `i` is now at offset -4. Here is the code that HLA emits for this new version of the procedure:

```

procedure p__hla_2;
begin p__hla_2;
    push( ebp );;Save ptr to previous activation record.
    mov( esp, ebp );;Point EBP at current activation record.
    sub( 4, esp );;Local storage.
    and( -4, esp );;Align stack on dword boundary.

```

```

; Exit point for the procedure:

xp__hla_2__hla_:
    mov( ebp, esp );;Deallocate local variables.
    pop( ebp );;Restore pointer to previous activation record.
    ret( 4 );;Return, popping parameters from the stack.
end p__hla_2;

```

As you can see, this code is smaller and a bit less complex. Unlike the code that built the display, it is common for an assembly language programmer to construct an activation record in a manner similar to this. Indeed, about the only instruction out of the ordinary above is the "AND" instruction that double-word-aligns the stack (OS calls require the stack to be double-word-aligned, and the system performance is much better if the stack is double-word aligned).

This code is still relatively inefficient if you don't pass parameters on the stack and you don't use automatic (non-static, local) variables. Many assembly language programmers pass their few parameters in machine registers and maintain local values in the registers. If this is the case, then the code above is pure overhead. You can inform HLA that you wish to take full responsibility for the entry and exit code by using the `@noframe` procedure option. Consider the following version of `p`:

```

procedure p( j:int32 ); @nodisplay; @noframe;
var
    i:int32;
begin p;
end p;

```

(This produces the same symbol table dump as the previous example).

HLA emits the following code for this version of `p`:

```

procedure p__hla_2;
begin p__hla_2;
end p__hla_2;

```

Whoa! There's nothing there! But this is exactly what the advanced assembly language programmer wants. With both the `@nodisplay` and `@noframe` options, HLA does not emit any extra code for you. You would have to write this code yourself.

By the way, you *can* specify the `@noframe` option without specifying the `@nodisplay` option. HLA still generates no extra code, but it will assume that you are allocating storage for the display in the code you write. That is, there will be an 8-byte `_display_` variable created and `i` will have an offset of -12 in the activation record. It will be your responsibility to deal with this. Although this situation is possible, it's doubtful this combination will be used much at all.

Note a major difference between the two versions of `p` when `@noframe` is not specified and `@noframe` is specified: if `@noframe` is not present, HLA automatically emits code to return from the procedure. This code executes if control falls through to the "end `p`;" statement at the end of the procedure. Therefore, if you specify the `@noframe` option, you must ensure that the last statement in the procedure is a `ret()` instruction or some other instruction that causes an unconditional transfer of control. If you do not do this, then control will fall through to the beginning of the next procedure in memory, probably with unintended results.

The `ret()` instruction presents a special problem. It is dangerous to use this instruction to return from a procedure that does not have the `@noframe` option. Remember, HLA has emitted code that pushes much data onto the stack. If you return from such a procedure without first removing this data from the stack, your program will probably crash. The correct way to return from a procedure without the `@noframe` option is to jump to the bottom of the procedure and run off the end of it. Rather than require you to explicitly put a label into your program and jump to this label, HLA provides the `exit procname`; instruction. HLA compiles the `exit` instruction into a `jmp` that

transfers control to the clean-up code HLA emits at the bottom of the procedure. Consider the following modification of `p` and the resulting assembly code produced:

```

procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;
    exit p;
    nop();
end p;

procedure p_hla_2;
begin p_hla_2;

    push( ebp );
    mov( esp, ebp );
    sub( 4, esp );
    and( -4, esp );
    jmp xp_hla_2_hla_;
    nop;
xp_hla_2_hla_:
    mov( ebp, esp );
    pop( ebp );
    ret( 4 );
end p_hla_2;

```

As you can see, HLA automatically emits a label to the assembly output file (`xp_hla_2_hla_` in this instance) at the bottom of the procedure where the clean-up code starts. HLA translates the "`exit p;`" instruction into a `jmp` to this label.

If you look back at the code emitted for the version of `p` with the `@noframe` option, you'll note that HLA did not emit a label at the bottom of the procedure. Therefore, HLA cannot generate a `jump` to this nonexistent label, so you cannot use the `exit` statement in a procedure with the `@noframe` option (HLA will generate an error if you attempt this).

Of course, HLA will *not* stop you from putting a `ret()` instruction into a procedure without the `@noframe` option (some people who know exactly what they are doing might actually want to do this). Keep in mind, if you decide to do this, that you must deallocate the local variables (that's what the "`mov esp, ebp`" instruction is doing), you need to restore EBP (via the "`pop ebp`" instruction above), and you need to deallocate any parameters pushed on the stack (the "`ret 4`" handles this in the example above). The following code demonstrates this:

```

procedure p( j:int32 ); @nodisplay;
var
    i:int32;
begin p;

    if( j = 0 ) then

        // Deallocate locals.

        mov( ebp, esp );

        // Restore old EBP

        pop( ebp );

```

```

        // Return and pop parameters

        ret( 4 );

    endif;
    nop();
end p;

procedure p_hla_2;
begin p_hla_2;

    push( ebp );
    mov( esp, ebp );
    sub( 4, esp );
    and( -4, esp );
    cmp( 0, (type dword [ebp+8]) );
    jne false_hla_3;
    mov( ebp, esp );
    pop( ebp );
    ret( 4 );
false_hla_3:
    nop;
xp_hla_2_hla_:
    mov( ebp, esp );
    pop( ebp );
    ret( 4 );
end p_hla_2;

```

If "real" assembly language programmers would generally specify both the `@noframe` and `@nodisplay` options, why not make them the default case (and use `@frame` and `@display` options to specify the generation of the activation record and display)? Well, keep in mind that HLA was originally designed as a tool to teach assembly language programming to beginning students. Those students have considerable difficulty comprehending concepts like activation records and displays. Having HLA generate the stack frame code and display generation code automatically saves the instructor from having to teach (and explain) this code. Even if the student never uses a display, it doesn't make the program incorrect to go ahead and generate it. The only real cost is a little extra memory and a little extra execution time. This is not a problem for beginning students who haven't yet learned to write efficient code. Therefore, HLA was optimized for the beginner at the expense of the advanced programmer. It is also worthwhile to point out that the behavior of the EXIT statement depends upon displays if you attempt to exit from a nested procedure; yet another reason for HLA's default behavior. Of course, you can always override HLA's default behavior by using the `@nodisplay` and `@noframe` compile-time variables.

If you are absolutely certain that your stack pointer is aligned on a four-byte boundary upon entry into a procedure, you can tell HLA to skip emitting the `and($FFFF_FFFC, ESP);` instruction by specifying the `@noalignstack` procedure option. Note that specifying `@noframe` also specifies `@noalignstack`.

11.8 Procedure Calls and Parameters in HLA

HLA's high-level support consists of three main features: HLL-like declarations, the HLL statements (IF, WHILE, etc), and HLA's support for procedure calls and parameter passing. This section discusses the syntax for procedure declarations and how HLA generates code to automatically pass parameters to a procedure.

The syntax for HLA procedure declarations was touched on earlier; however, it's probably a good idea to review the syntax as well as describe some options that previous sections ignored. There are several procedure declaration forms; the following examples demonstrate them all¹:

```
// Standard procedure declaration:

procedure procname (opt_parms); proc_options
begin procname;
    << procedure body >>
end procname;

// New style procedure declaration:

proc
    procname :procedure(opt_parms); proc_options
    begin procname;
        << procedure body >>
    end procname;

// External procedure declarations:

procedure extname (opt_parms); proc_options external;
procedure extname (opt_parms); proc_options external( "name");

// New style external procedure declarations:

proc
    extname :procedure(opt_parms) {proc_options}; external;
    extname :procedure(opt_parms) {proc_options}; external( "name");
endproc;

// Original forward procedure declarations:

procedure fwdname (opt_parms); proc_options forward;

// New style forward procedure declarations:

proc
    fwdname :procedure(opt_parms) {proc_options}; forward;
```

opt_parms indicates that the parameter list is optional; the parentheses are not present if there are no parameters present.

Proc_options is any combination (zero or more) of the procedure options (see the discussion earlier for these options)

11.9 Calling HLA Procedures

There are two standard ways to call an HLA procedure: use the **call** instruction or simply specify the name of the procedure as an HLA statement. Both mechanisms have their plusses and minuses.

1. This section only discusses procedure declarations. Other sections will describe iterators and methods.

To call an HLA procedure using the **call** instruction is exceedingly easy. Simply use either of the following syntaxes:

```
call( procName );
call procName;
```

Either form compiles into an 80x86 **call** instruction that calls the specified procedure. The difference between the two is that the first form (with the parentheses) returns the procedure's "returns" value, so this form can appear as an operand to another instruction. The second form above always returns the empty string, so it is not suitable as an operand of another instruction. Also, note that the second form requires a statement or procedure label, you may not use memory-addressing modes in this form; on the other hand, the second form is the only form that lets you "call" a statement label (as opposed to a procedure label); this form is useful on occasion.

If you use the **call** statement to call a procedure, then you are responsible for passing any parameters to that procedure. In particular, if the parameters are passed on the stack, you are responsible for pushing those parameters (in the correct order) onto the stack before the call. This is a lot more work than letting HLA push the parameters for you, but in certain cases you can write more efficient code by pushing the parameters yourself.

The second way to call an HLA procedure is to simply specify the procedure name and a list of actual parameters (if needed) for the call. This method has the advantage of being easy and convenient at the expense of a possible slight loss in efficiency and flexibility. This calling method should also prove familiar to most HLL programmers. As an example, consider the following HLA program:

```
program parameterDemo;

#include( "stdio.hhf" );

procedure PrtAplusB( a:int32; b:int32 ); @nodisplay;
begin PrtAplusB;

    mov( a, eax );
    add( b, eax );
    stdout.put( "a+b=", (type int32 eax ), nl );

end PrtAplusB;

static
    v1:int32 := 25;
    v2:int32 := 5;

begin parameterDemo;

    PrtAplusB( 1, 2 );
    PrtAplusB( -7, 12 );
    PrtAplusB( v1, v2 );

    mov( -77, eax );
    mov( 55, ebx );
    PrtAplusB( eax, ebx );

end parameterDemo;
```

This program produces the following output:

```
a+b=3
a+b=5
a+b=30
```



```
a+b=-22
```

As you can see, call `PrtAplusB` in HLA is very similar to calling procedures (and passing parameters) in a high-level language like C/C++ or Pascal. There are, however, some key differences between and HLA call and a HLL procedure call. The next section will cover those differences in detail. The important thing to note here is that if you choose to call a procedure using the HLL syntax (that is, the second method above), you will have to pass the parameters in the parameter list and let HLA push the parameters for you. If you want to take complete control over the parameter passing code, you should use the **call** instruction.

11.10 Parameter Passing in HLA, Value Parameters

The previous section probably gave you the impression that passing parameters to a procedure in HLA is nearly identical to passing those same parameters to a procedure in a high-level language. The truth is, the examples in the previous section were rigged. There are actually many restrictions on how you can pass parameters to an HLA procedure. This section discusses the parameter passing mechanism in detail.

The most important restriction on actual parameters in a call to an HLA procedure is that HLA only allows memory variables, registers, constants, and certain other special items as parameters. In particular, you cannot specify an arithmetic expression that requires computation at run-time (although a constant expression, computable at compile time is okay). The bottom line is this: if you need to pass the value of an expression to a procedure, you must compute that value prior to calling the procedure and pass the result of the computation; HLA will not automatically generate the code to compute that expression for you.

The second point to mention here is that HLA is a strongly typed language when it comes to passing parameters. This means that with only a few exceptions, the type of the actual parameter must exactly match the type of the formal parameter. If the actual parameter is an `int8` object, the formal parameter had better not be an `int32` object or HLA will generate an error. The only exceptions to this rule are the **byte**, **word**, **dword**, **qword**, **tbyte**, and **lword** types. If a formal parameter is of type **byte**, the corresponding actual parameter may be any one-byte data object. If a formal parameter is a **word** object, the corresponding actual parameter can be any two-byte object. Likewise, if a formal parameter is a **dword** object, the actual parameter can be any four-byte data type. And so on... Conversely, if the actual parameter is a **byte**, **word**, **dword**, **qword**, **tbyte**, or **lword** object, it can be passed without error to any one, two, four, eight, ten, or sixteen-byte actual parameter (respectively). Programmers who are lazy make all their parameters one of these hexadecimal types (at least, wherever possible). Programmers who care about the quality of their code use untyped parameters cautiously.

For efficiency reasons (dictated by the operating system and the Intel ABI), HLA procedure calls always pass all parameters as a multiple of four bytes. When passing a byte-sized parameter on the stack by value, the actual parameter value consumes the L.O. byte of the double word passed on the stack. The function ignores the H.O. three bytes of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. three bytes if it is not inconvenient to do so.

11.10.1 Passing Byte-Sized Parameters by Value

When passing a byte-sized constant, you should simply push a double-word containing the 8-bit value, e.g.,

```
pushd( 5 );  
call someSubroutine;
```

When passing the 8-bit value of the 8-bit registers AL, BL, CL or DL onto the stack, you should simply push the 32-bit register that holds the 8-bit register, e.g.,

```
push( eax ); // Pushes AL onto the stack  
call someSubroutine;  
push( ebx ); // Pushes BL onto the stack  
call someOtherSubroutine;
```

Note that this trick does not apply to the AH, BH, CH, or DH registers. The best code to use when you need to push these registers is to drop the stack down by four bytes and then move the desired register into the memory location you've just created on the stack, e.g.,

```
sub( 4, esp );
mov( AH, [esp] ); // Pushes AH onto the stack
call someSubroutine;
sub( 4, esp );
mov( BH, [esp] ); // Pushes BH onto the stack
call someOtherSubroutine;
```

Here's another way you can accomplish this (a little slower, but leaves zeros in the H.O. three bytes):

```
pushd( 0 );
mov( CH, [esp] ); // Pushes CH onto the stack
call someSubroutine;
```

When passing a byte-sized variable, you should try to push the variable's value and the following three bytes, using code like the following:

```
pushd( (type dword eightBitVar) );
call someSubroutine;
```

There is one drawback to the approach above. In three very rare cases the code above could cause a segmentation fault. If the 8-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, if a register is available, is to move the 8-bit value into AL, BL, CL, or DL and push the corresponding 32-bit register. If no registers are available, then you can write code like the following:

```
push( eax );
push( eax );
mov( byteVar, al );
mov( al, [esp+4] );
pop( eax );
call someSubroutine;
```

This code is ugly and slightly inefficient, but it will always work (assuming, of course, you don't get a stack overflow).

The HLA compiler will generate code similar to this last example if you pass a static byte variable as the actual parameter to a library function expecting an 8-bit value parameter:

```
someLibraryRoutine( byteVar );
```

Therefore, if efficiency is a concern to you, you should try to load the byte variable (byteVar in this example) into AL, BL, CL, or DL prior to calling someLibraryRoutine, e.g.,

```
mov( boolVar, al );
someLibraryRoutine( al );
```

Note that HLA will push a whole double-word if the actual parameter is an automatic variable or some other parameter (both of which are allocated on the stack). In this case, you're generally guaranteed that the three bytes following the byte variable in memory are in readable memory space. It is possible to set up the EBP register to violate this assumption, but HLA assumes that you're not trying to cause a segmentation fault and assumes that it's safe to access those extra three bytes beyond the byte variable. Here is an example of the code generation for various byte parameters:

```
program t;
static
```

```

    b:byte;

procedure p( b0:byte ); @nodisplay;

    procedure q( b1:byte; b2:byte; b3:byte; b4:byte; b5:byte; b6:byte;
b7:byte );
    begin q;
    end q;

var
    b8 :byte;
    w  :word;
    b9 :byte;

begin p;

    q( b, b0, b8, b9, al, ah, 255 );

end p;

begin t;
end t;

```

Here's the code generation for the call to the *q* procedure (MASM syntax, as output by HLA v2.2). Note that the comments in italics are not emitted by HLA:

```

    ; Push static variable b onto the stack

pushd( 0 );
push( eax );
mov( b_hla_1, al );
mov( al, [esp+4] );
pop( eax );

    ; Push parameter b0 onto the stack:

push( (type dword [ebp+8]) );

    ; Push automatic variable b8 onto the stack:

pushd( 0 );
push( eax );
mov( [ebp-1], al );
mov( al, [esp+4] );
pop( eax );

    ; Push automatic variable b9 onto the stack:

push( (type dword [ebp-4]) );

    ; Push AL onto the stack:

push( eax );

    ; Push AH onto the stack:

```

```

sub( 4, esp );
mov( ah, [esp] );

; Push 255 onto the stack:

pushd( 255 );

; Call q

call q__hla_3;

```

Note the difference in code generation between the *b8* and *b9* local variables. Because *b8* has an offset of -1 in the activation record and HLA is playing it safe and only accessing a single byte at [EBP-1]. This prevents any access to bytes that have a positive or zero index off of EBP. It generates slightly better code if the variable's index is -4 or less because it can safely push four bytes and all four bytes will have a negative offset from EBP. Moral of the story: if you intend to use the HLA HLL-like calling sequence and you intend to pass local (automatic) variables as byte parameters, try to ensure that those byte variables have an offset of -4 or less in the activation record.

If one of the EAX, EBX, ECX, or EDX registers is always free and available when calling a procedure with byte-sized parameters, you can often improve the quality of the code that HLA generates by attaching an **@uses** procedure option to the procedure's declaration. Consider the following modification to the above code:

```

program t;
static
    b:byte;

procedure p( b0:byte ); @nodisplay;

    procedure q
    (
        b1 :byte;
        b2 :byte;
        b3 :byte;
        b4 :byte;
        b5 :byte;
        b6 :byte;
        b7 :byte
    ); @use ecx;
    begin q;
    end q;

var
    b8 :byte;
    w  :word;
    b9 :byte;

begin p;

    q( b, b0, b8, b9, al, ah, 255 );

end p;

begin t;

```

```
end t;
```

The **@use ecx;** clause tells HLA that it can use the ECX register, wiping out its contents, if HLA finds it convenient to do so. Compare the code HLA generates for the call to *q* above against the earlier versions:

```
; Push static variable b onto the stack

mov( b__hla_1, cl );
push( ecx );

; Push parameter b0 onto the stack:

push( (type dword [ebp+8]) );

; Push automatic variable b8 onto the stack:

mov( [ebp-1], cl );
push( ecx );

; Push automatic variable b9 onto the stack:

push( (type dword [ebp-4]) );

; Push AL onto the stack:

push( eax );

; Push AH onto the stack:

sub( 4, esp );
mov( ah, [esp] );

; Push 255 onto the stack:

pushd( 255 );

; Call q

call q__hla_3;
```

If you want to use the high-level calling sequence, but you don't like the inefficient code HLA sometimes produces when generating code to pass your byte-sized parameters, you can always use the **#{...}#** sequence parameter to override HLA's code generation and substitute your own code for one or two parameters. Of course, it doesn't make any sense to pass all the parameters in a procedure using this trick, it would be far easier just to use the call instruction. Example:

```
q( b, b0, #{push( (type dword b8) );}#, b9, al, ah, 255 );
```

If efficiency is a concern to you and the **@use reg₃₂** procedure option isn't acceptable (perhaps because you can't guarantee that a register is always available), you should try to load a byte variable you want to pass as a parameter into AL, BL, CL, or DL prior to calling the subroutine, e.g.,

```
mov( byteVar, al );
```

```
someSubroutine( al );
```

11.10.2 Passing Word-Sized Parameters by Value

When passing a word-sized parameter on the stack by value, the actual parameter value consumes the L.O. two bytes of the double word passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

When passing a word-sized constant, you should simply push the double word containing the 16-bit value, e.g.,

```
pushd( 5 );
call someSubroutine;
```

When passing the 16-bit value of a 16-bit register (AX, BX, CX, DX, SI, DI, BP, or SP) onto the stack, you should simply push the 32-bit register that holds the 16-bit register, e.g.,

```
push( eax ); // Pushes AX onto the stack
call someSubroutine;
push( ebx ); // Pushes BX onto the stack
call someOtherSubroutine;
```

When passing a word-sized variable, you should try to push the variable's value and the following two bytes, using code like the following:

```
pushd( (type dword sixteenBitVar) );
call someSubroutine;
```

There is one drawback to the approach above. In three very rare cases, the code above could cause a segmentation fault. If the 16-bit variable is located on the last three bytes of a page in memory (4,096 bytes) and the next memory page is not readable, the system will generate a fault if you attempt to push all four bytes. In such a case, the next best solution, is to use two consecutive pushes:

```
pushw( 0 ); // H.O. word is zeros
push( sixteenBitVar );
call someSubroutine;
```

The HLA compiler will generate code similar to this last example if you pass a word variable as the actual parameter to a function expecting a 16-bit value parameter:

```
someSubroutine( wordVar );
```

Here is a more complete example:

```
program t;
static
  w:word;

procedure p( w0:word ); @nodisplay;

  procedure q
  (
    w1 :word;
    w2 :word;
```

```
        w3 :word;
        w4 :word;
        w5 :word;
        w6 :word;
        w7 :word
    );
    begin q;
    end q;

var
    w8 :word;
    w  :word;
    w9 :word;

begin p;

    q( w, w0, w8, w9, ax, si, 255 );

end p;

begin t;
end t;
```

This procedure produces the following (pseudo-HLA) assembly language output:

```
; Push w

pushw( 0 );
push( (type word [ebp-4]) );

; Push w0

push( (type dword [ebp+8]) );

; Push w8

pushw( 0 );
push( (type word [ebp-2]) );

; Push w9

pushw( 0 );
push( (type word [ebp-6]) );

; Push ax

push( eax );

; Push si

push( esi );

; Push 255

pushd( 255 );
```

```

; call q

call q_hla_3;

```

11.10.3 Passing Double-Word-Sized Parameters by Value

Because 32-bit double-word objects are the native x86 data type, there are only a few issues with passing 32-bit parameters on the stack to a standard library routine.

First, and this applies to all stack operations not just 32-bit pushes and pops, you should always keep the stack 32-bit aligned. That is, the value in ESP should always contain a value that is a multiple of four (i.e., the L.O. two bits of ESP must always contain zeros). If this is not the case, many OS API and standard library function calls will fail.

When passing a 32-bit value onto the stack, just about any mechanism you can use to push that value is perfectly valid. You can efficiently push constants, registers, and memory locations using a single push instruction, e.g.,

```

pushd( 12345 ); // Passing a 32-bit constant
push( mem32 ); // Passing a dword variable
push( eax ); // Passing a 32-bit register
call someRoutine;

```

Of course, you can always use the HLA high-level syntax to pass a 32-bit object to a subroutine. HLA automatically generates the appropriate code to pass the dword object as a parameter on the stack. Note that HLA automatically recognizes the lexeme "dx:ax" as a 32-bit value and will push these two registers (DX first, AX second) onto the stack.

11.10.4 Passing Quad-Word-Sized Parameters by Value

Because qword (64-bit) objects are a multiple of 32 bits in size, manually passing qword objects on the stack is very easy. All you need do is push two double-word values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. dword first and the L.O. dword second.

If the qword value is held in a register pair, then push the register containing the H.O. dword first and the L.O. dword second. For example, if EDX:EAX contains the 64-bit value, then you'd push the qword as follows:

```

push( edx ); // Push H.O. dword
push( eax ); // Push L.O. dword
call someLibraryRoutine;

```

If the qword value is held in a qword variable, then you must first push the H.O. dword of that variable followed by the L.O. dword, e.g.,

```

push( (type dword qwordVar[4])); // Push H.O. dword first
push( (type dword qwordVar)); // Push L.O. dword second
call someLibraryRoutine;

```

If the qword value you wish to pass is a constant, then you have to compute the L.O. and H.O. dword values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```

pushd( ((some64bitConst) >> 32);
pushd( ((some64bitConst) & $FFFF_FFFF );
call someLibraryRoutine;

```


If this is something you do frequently, you might want to create a macro to break up the 64-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 64-bit object to a subroutine. HLA automatically generates the appropriate code to pass the `qword` object as a parameter on the stack. Note that HLA automatically recognizes the lexeme `"edx:eax"` as a 64-bit value and will push these two registers (EDX first, EAX second) onto the stack.

11.10.5 Passing Tbyte-Sized Parameters by Value

For efficiency reasons, operating system APIs and HLA standard library routines always pass all parameters as a multiple of four bytes. When passing a **tbyte**-sized parameter on the stack by value, the actual parameter value consumes the L.O. ten bytes of the three double words passed on the stack. The function ignores the H.O. word of the value passed for this parameter, though by convention (to make debugging a little easier) you should try to pass zeros in the H.O. word if it is not inconvenient to do so.

The following code demonstrates how to pass a ten-byte object to a standard library routine:

```
pushw( 0 ); // Dummy H.O. word of zero
push( (type word tbyteVar[8])); // Push H.O. byte of tbyte object
push( (type dword tbyteVar[4])); // Push bytes 4-7 of tbyte object
push( (type dword tbyteVar[0])); // Push L.O. dword of tbyte object
call someLibraryRoutine;
```

If your **tbyte** object is not at the very end of allocated memory, you could probably combine the first two instructions in this sequence to produce the following (slightly more efficient) code:

```
push( (type dword tbyteVar[8])); // Pushes two extra bytes.
```

This pushes the two bytes beyond `tbyteVar` onto the stack but, presumably, the function will ignore all bytes beyond the tenth byte passed on the stack, so the actual values in those H.O. two bytes are irrelevant. Note the earlier discussion (in the section on pushing byte parameters) about the rare possibility of a memory access error when using this trick.

Of course, you can always use the HLA high-level syntax to pass an 80-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the **tbyte** object as a 12-byte parameter on the stack.

11.10.6 Passing Lword-Sized Parameters by Value

Because **lword** (128-bit) objects are a multiple of 32 bits in size, manually passing **lword** objects on the stack is very easy. All you need do is push four `dword` values. Because the stack grows downward in memory and the x86 is a little endian machine, you must push the H.O. `dword` first and the L.O. `dword` last.

If the **lword** value is held in an **lword** variable, then you must first push the H.O. `dword` of that variable followed by the lower-order `dwords`, down to the L.O. `dword`, e.g.,

```
push( (type dword qwordVar[12])); // Push H.O. dword first
push( (type dword qwordVar[8])); // Push bytes 8-11 second
push( (type dword qwordVar[4])); // Push bytes 4-7 third
push( (type dword qwordVar)); // Push L.O. dword last
call someRoutine;
```

If the **lword** value you wish to pass is a constant, then you have to compute the four `dword` values for that constant and push those. When using HLA, you can use the compile-time computational capabilities of HLA to do this for you, e.g.,

```
pushd( ((some128bitConst) >> 96);
pushd( ((some128bitConst) >> 64 & $FFFF_FFFF );
pushd( ((some128bitConst) >> 32 & $FFFF_FFFF );
```

```
pushd( ((some128bitConst) & $FFFF_FFFF );
call someRoutine;
```

If this is something you do frequently, you might want to create a macro to break up the 128-bit value and push it for you.

Of course, you can always use the HLA high-level syntax to pass a 128-bit object to a standard library routine. HLA automatically generates the appropriate code to pass the **lword** object as a parameter on the stack.

11.10.7 Passing Large Parameters by Value

When using a HLL-like call, HLA will automatically copy an actual value parameter into local storage for the procedure, regardless of the size of the parameter. If your value parameter is a one-million-byte array, HLA will allocate storage for 1,000,000 bytes and then copy that array in on each call. C/C++ programmers may expect HLA to automatically pass arrays by reference (as C/C++ does) but this is not the case. If you want your parameters passed by reference, you must explicitly state this.

If you're not using a HLL-like call, it is your responsibility to make room for large parameters and copy that parameter data to the stack before a call. Consider the following example that passes a fair-sized array (256 bytes) by value:

```
program t;
type
    array:byte[256];

procedure p( a:array );
begin p;
end p;

static
    theArray:array;

begin t;

    p( theArray );

end t;
```

Here is the code that HLA generates for the call to procedure *p* in the main program above:

```
// Reserve storage for the 256-byte array to be passed on the stack
lea( [esp-256], esp );

// Preserve the registers that rep.movsd uses:

push( esi );
push( edi );
push( ecx );
pushfd;

// Copy the array from the source location (theArray) to
// the storage just allocated on the stack:

cld;
lea( theArray__hla_2, esi );
```

```
mov( 64, ecx );
lea( [esp+16], edi );// Address of array on stack
rep movsd

// Restore the registers:

popfd;
pop( ecx );
pop( edi );
pop( esi );

// Call p
call p__hla_1;
```

The code HLA generates to copy value parameters, while not particularly bad, certainly isn't always optimal. If you need the fastest possible code when passing parameters by value on the stack, it would be better if you explicitly pushed the data yourself.

11.11 Parameter Passing in HLA, Reference, Value/Result, and Result Parameters

The one good thing about pass by reference, pass by value/result, and pass by result parameters is that the parameters are always four byte pointers, regardless of the size of the actual parameter. Therefore, HLA has an easier time generating code for these parameters than it does generating code for pass by value parameters.

In a procedure call HLA treats reference, value/result, and result parameters identically. The code within the procedure is responsible for differentiating these parameter types (value/result and result parameters generally require copying data between local storage and the actual parameter). The following discussion will simply refer to pass by reference parameters, but it applies equally well to pass by value/result and pass by result.

When passing a parameter by reference, you must push the address of the actual parameter (rather than its value) onto the stack. For static objects, you can use the push immediate instruction, e.g., (in HLA syntax):

```
pushd( &staticVar );
call someLibraryRoutine;
```

For automatic variables, or objects whose address is not a simple static offset (e.g., a complex pointer address involving registers and what-not), you'll have to use the LEA instruction to first compute the address and then push that register's value, e.g.,

```
lea( eax, anAutomaticVar ); // Variable allocated on the stack
push( eax );
call someLibraryRoutine;
```

If the variable's address is a simple offset from a single register (such as automatic variables declared in the stack frame and referenced off of the EBP register), you can push the address of the variable by pushing the base register and adding the offset of that variable to the value left on the stack, thusly:

```
push( ebp ); // anAutoVar is found at EPB+@offset(anAutoVar)
add( @offset( anAutoVar ), (type dword [esp]));
call someLibraryRoutine;
```

If the address you want to pass in a reference parameter is a complex address, you'll have to use the LEA instruction to compute that address and push it onto the stack. This, unfortunately, requires a free 32-bit register. If no 32-bit registers are free, you can use code like the following to achieve this:

```
sub( 4, esp ); // Reserve space for parameter on stack
push( eax );  // Preserve EAX
lea( eax, [ebp+@offset(autoVar)][ecx*4+3] );
mov( eax, [esp+4] ); // Store in parameter location
pop( eax );   // Restore EAX
call someLibraryRoutine;
```

Of course, it's much nicer to use the HLA high-level syntax for calls like this as the HLA compiler will automatically handle all the messy code generation details for you.

Like high-level languages, HLA places a whopper of a restriction on pass by reference parameters: they can only be memory locations. Constants and registers are not allowed since you cannot compute their address. Do keep in mind, however, that any valid memory-addressing mode is a valid candidate to be passed by reference; you do not have to limit yourself to static and local variables. For example, "[eax]" is a perfectly valid memory location, so you can pass this by reference (assuming you type-cast it, of course, to match the type of the formal parameter). The following example demonstrate a simple procedure with a pass by reference parameter:

```
program refDemo;

#include( "stdlib.hhf" )

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

begin refDemo;

    stdout.put( "(1) i=", i, nl );
    mov( 25, i );
    stdout.put( "(2) i=", i, nl );
    refParm( i );
    stdout.put( "(3) i=", i, nl );

end refDemo;
```

The output produced by this code is

```
(1) i=5
(2) i=25
(3) i=12345
```

As you can see, the parameter *a* in `refParm` exhibits pass by reference semantics since the change to the value *a* in `refParm` changed the value of the actual parameter (*i*) in the main program.

Note that HLA passes the address of `i` to `refParm`, therefore, the `a` parameter contains the address of `i`. When accessing the value of the `i` parameter, the `refParm` procedure must dereference the pointer passed in `a`. The two instructions in the body of the `refParm` procedure accomplish this.

Look at the code that HLA generates for the call to `refParm`:

```
    pushd( &(i__hla_1884+0) );
    call refParm__hla_1883;
```

(`i__hla_1884` is the back-end assembler compatible name that HLA generated for the static variable `i`.)

As you can see, this program simply computed the address of `i` and pushed it onto the stack. Now consider the following modification to the main program:

```
program refDemo;
#include( "stdlib.hhf" );

    procedure refParm( var a:int32 );
    begin refParm;

        mov( a, eax );
        mov( 12345, (type int32 [eax]));

    end refParm;

    static
        i:int32:=5;

    var
        j:int32;

begin refDemo;

    mov( 4, ebx );
    mov( 0, j );
    refParm( j );
    refParm( i[ebx] );
    lea( eax, j );
    refParm( [eax+ebx] );

end refDemo;
```

This version emits the following (pseudo-HLA syntax) code for the body of the main program:

```
// mov( 4, ebx );

mov( 4, ebx );

// mov( 0, j );

mov( 0, (type dword [ebp-8]) );

// refParm( j );
```

```

push( ebp );
add( -8, (type dword [esp]) );
call refParm_hla_1883;

// refParm( i[ebx] );

push( eax );
push( eax );
lea( i_hla_1884[ebx], eax );
mov( eax, [esp+4] );
pop( eax );
call refParm_hla_1883;

// lea( eax, j );

lea( [ebp-8], eax );

// refParm( [eax+ebx] );

push( eax );
push( eax );
lea( [eax+ebx*1], eax );
mov( eax, [esp+4] );
pop( eax );
call refParm_hla_1883;

```

As you can see, the code emitted for the last two calls is ugly. These calls would be good candidates for using the **call** instruction directly. Also see *Hybrid Parameters* elsewhere in this chapter. Another option is to use the **@use reg32** option to tell HLA it can use one of the 32-bit registers as a scratchpad. Consider the following:

```

procedure refParm( var a:int32 ); @use esi;
.
.
.

```

This sequence generates the following code (which is a little better than the previous example):

```

// mov( 4, ebx );

mov( 4, ebx );

// mov( 0, j );

mov( 0, (type dword [ebp-8]) );

// refParm( j );

lea( [ebp-8], esi );
push( esi );
call refParm_hla_1883;

// refParm( i[ebx] );

lea( i_hla_1884[ebx], esi );
push( esi );
call refParm_hla_1883;

```

```

// lea( eax, j );

lea( [ebp-8], eax );

// refParm( [eax+ebx] );

lea( [eax+ebx*1], esi );
push( esi );
call refParm__hla_1883;

```

As a rule, the type of an actual reference parameter must exactly match the type of the formal parameter. There are a couple exceptions to this rule. First, if the formal parameter is `DWORD`, then HLA will allow you to pass any four-byte data type as an actual parameter by reference to this procedure. Second, you can pass an actual `DWORD` parameter by reference if the formal parameter is a four-byte data type.

There is a third exception to the "the types must exactly match" rule. If the formal reference parameter is some data type HLA will allow you to pass an actual parameter that is a pointer to this type. Note that HLA will actually pass the *value* of the pointer, rather than the *address* of the pointer, as the reference parameter. This turns out to be convenient, particularly when calling Win32 API functions and other C/C++ code. Note, however, that this behavior isn't always intuitive, so be careful when passing pointer variables as reference parameters.

If you want to pass the value of a double word or pointer variable in place of the address of such a variable to a pass by reference, value/result, or result parameter, simply prefix the actual parameter with the `val` reserved word in the call to the procedure, e.g.,

```
refParm( val dwordVar );
```

This tells HLA to use the value of the variable rather than it's address.

You may also use the `val` keyword to pass an arbitrary 32-bit numeric value for a string parameter. This is useful in certain Win32 API calls where you pass either a pointer to a zero-terminated sequence of characters (i.e., a string) or a small integer "ATOM" value.

11.12 Untyped Reference Parameters

HLA provides a special formal parameter syntax that tells HLA that you want to pass an object by reference and you don't care what its type is. Consider the following HLA procedure:

```

procedure zeroObject( var object:byte; size:uns32 );
begin zeroObject;
  << code to write "size" zeros to "object" >
end zeroObject;

```

The problem with this procedure is that you will have to coerce non-byte parameters to a byte before passing them to `zeroObject`. That is, unless you're passing a byte parameter, you always have to call `zeroObject` thusly:

```
zeroObject( (type byte NotAByte), sizeToZero );
```

For some functions you call frequently with different types of data, this can get painful very quickly.

The HLA untyped reference parameter syntax solves this problem. Consider the following declaration of `zeroObject`:

```

procedure zeroObject( var object:var; size:uns32 );
begin zeroObject;
  << code to write "size" zeros to "object" >

```

```
end zeroObject;
```

Notice the use of the reserved word **var** instead of a data type for the object parameter. This syntax tells HLA that you're passing an arbitrary variable by reference. Now you can call `zeroObject` and pass any (memory) object as the first parameter and HLA won't complain about the type, e.g.,

```
zeroObject( NotAByte, sizeToZero );
```

Note that you may only pass untyped objects by reference to a procedure.

Note that untyped reference parameters always take the address of the actual parameter to pass on to the procedure, even if the actual parameter is a pointer. Normal pass by reference semantics in HLA will pass the value of a pointer, rather than the address of the pointer variable, if the base type of the pointer matches the type of the reference parameter. Sometimes you'll have the address of an object in a register or a pointer variable and you'll want to pass the value of that pointer object (i.e., the address of the ultimate object) rather than the address of the pointer variable. To do this, simply prefix the actual parameter with the **val** keyword, e.g.,

```
zeroObject( ptrVar );      // Passes the address of ptrVal
zeroObject( val ptrVar ); // Passes ptrVar's value.
```

11.13 Pass by Value/Result and Pass by Result Parameters

Although the behavior of pass by value/result and pass by result parameters is identical to pass by reference on a procedure call (that is, the caller passes the address of the object to the subroutine rather than the value), inside the procedure the behavior of these parameter-passing mechanisms is quite different. This section will discuss those differences and how you use pass by value/result and pass by result parameters in a program. The pass by result parameter-passing mechanism is actually a subset of the pass by value/result mechanism, so this section will fully describe the pass by value/result mechanism and then point out the difference between the two parameter-passing mechanisms.

One problem with the pass by reference calling sequence is that it is possible to create aliases of variables in a parameter list that lead to non-intuitive results in your program. Consider the following (very famous) example:

```
program refDemo;
#include( "stdlib.hhf" );

procedure famous( var a:int32; var b:int32 );
begin famous;

    // a := 5;

    mov( a, ebx );
    mov( 5, (type int32 [ebx]));

    // b := 10;

    mov( b, ebx );
    mov( 10, (type int32 [ebx] ) );

    // print a+b

    mov( a, ebx );
    mov( [ebx], eax );
    mov( b, ebx );
    add( [ebx], eax );
    stdout.put( "a+b=", (type int32 eax), nl );
```



```

        end famous;

static
    someVar:int32;

begin refDemo;

    famous( someVar, someVar );

end refDemo;

```

The output from this program is "a+b=20" which is somewhat counter-intuitive (the intuitive result, looking only at the code in *famous*, would be "a+b=15"). If you study the code above (no need to look at the low-level machine code), you'll discover the problem. In the call to *famous*, the main program passes the address of *someVar* in both parameter positions. Therefore, inside *famous*, both *a* and *b* contain the address of *someVar*. When *famous* stores the value 5 into the variable pointed at by *a*, it overwrites in *someVar* with 5. Because *b* also points at *someVar*, when *famous* stores the value 10 into the location pointed at by *b*, it overwrites the value 5 that it originally stored there. When *famous* accesses the values pointed at by *a* and *b* (to compute their sum), it retrieves the value 10 for both memory accesses and, therefore, computes the sum of 20.

The problem in this example is that *a* and *b* are aliases (different names for the same variable). The occurrence of aliases can sometimes create problems, as this example demonstrates. Note that the pass by value parameter-passing mechanism doesn't suffer from this problem because it makes a distinct copy of the parameter's value. Were you to use pass by value in the example above, you'd get the intuitive result of "a+b=15" because *a* and *b* would have both been separate variables in *famous*' activation record. The drawback to pass by value, of course, is that any modification to a pass by value parameter is not reflected in the actual parameter that was passed to the procedure.

The pass by value/result parameter-passing mechanism is a combination of the pass by value and pass by reference mechanisms: it provides a mechanism for passing values into and out of a procedure while avoiding the problem of aliases (by creating a local copy of the actual parameter's value in the activation record of the procedure). Here's how pass by value/result works:

- the caller passes in an address of the actual parameter
- the subroutine makes a copy of the actual parameter's data into a local (automatic) variable
- the subroutine manipulate the copy of the data just as it would a value parameter or any other local (automatic) variable
- before the subroutine returns, it copies the data from the local object to the memory location pointed at by the address that the caller passed to the procedure for the actual parameter.

Note that two copies of the data are made: one copy is made upon entry into the subroutine (from the actual parameter to the local copy) and one copy is made upon exit from the procedure (from the local copy to the actual parameter). Inside the procedure, however, all pass by value/result parameters have their own copy of their actual parameter's data, so there are no aliases. Were you to rewrite *famous* to use pass by value result, you'd get the intuitive result of "a+b=15".

The pass by value/result and pass by result parameter-passing mechanisms are unusual because their behavior is quite different when **@frame** or **@noframe** is specified for the procedure. Whenever you specify **@frame** (or if this is the default condition), then HLA will automatically allocate storage for the local copy of the data and *the formal parameter name will refer to that local data in the activation record*. You will not have direct access (via some sort of parameter name) to the address that the caller actually passed into the procedure. This is actually convenient and natural; you want to treat pass by value/result parameters as though they were simple values. Consider the following example:

```

program valresDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( valres a:int32; valres b:int32 );

```

```

begin p;
end p;

static
  someVar:int32;
  someVar2:int32;

begin valresDemo;

  p( someVar, someVar2 );

end valresDemo;

```

Here is the symbol table for procedure *p* in this code:

```

p          <0,proc>:Procedure type (ID=p__hla_1)
-----
_ vars_    <1,cons>:uns32, (4 bytes) =8
b          <1,var >:int32, (4 bytes, ofs:-8)
a          <1,var >:int32, (4 bytes, ofs:-4)
_ parms_   <1,cons>:uns32, (4 bytes) =8
a          <1,vrp >:int32, (4 bytes, ofs:12)
b          <1,vrp >:int32, (4 bytes, ofs:8)
_ finalize_ <1,val >:string, (0 bytes) =""
_ initialize_ <1,val >:string, (0 bytes) =""
p          <1,proc>:
-----

```

Note that the symbols *a* and *b* appear twice in *p*'s symbol table. The first pair (which HLA always finds when searching through the symbol table) corresponds to the local copies of these parameters that HLA allocates on the stack. The second pair (which are inaccessible to your program because HLA always finds the other pair first) corresponds to the actual addresses that the caller pushes onto the stack.

Here is the code that HLA generates for procedure *p* (including the code it automatically generates to copy the value/result parameter data to the local copy):

```

procedure p__hla_1;
begin p__hla_1;

  // Set up the activation record:

  push( ebp );
  mov( esp, ebp );

  // Allocate storage for locals (specifically, for the
  // local copies of a and b):

  sub( 8, esp );

  // Copy the actual values pointed at by a and b to
  // the local copies:

  push( esi );
  push( ecx );

```

```

    mov( [ebp+12], esi );// Get pointer to a
    mov( [esi], ecx );// Get a's value
    mov( ecx, [ebp-4] );// Store value into local copy

    mov( [ebp+8], esi );// Get pointer to b
    mov( [esi], ecx );// Get b's value
    mov( ecx, [ebp-8] );// Store value into local copy

    pop( ecx );
    pop( esi );

// Exit from procedure p:

xp_hla_1_hla_:

    // Copy the data from the local copies back to
    // the actual parameters:

    push( edi );
    push( ecx );

    mov( [ebp+12], edi );// Get pointer to a
    mov( [ebp-4], ecx );// Get a's local value
    mov( ecx, [edi] );// Store into actual a

    mov( [ebp+8], edi );// Get pointer to b
    mov( [ebp-8], ecx );// Get b's local value
    mov( ecx, [edi] );// Store into actual b

    pop( ecx );
    pop( edi );

    // Clean up the activation record and leave:

    mov( ebp, esp );
    pop( ebp );
    ret( 8 );
end p_hla_1;

```

If the procedure has the **@noframe** option, then the formal parameter name refers to the address passed by the caller on the stack. It is your responsibility to allocate storage for the local copy of the pass by value/result parameter and copy the data from the address specified to your local copy. Here's the code above with the **@noframe** option:

```

program valresDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( valres a:int32; valres b:int32 ); @noframe;
    begin p;
    end p;

static
    someVar:int32;
    someVar2:int32;

```

```
begin valresDemo;

    p( someVar, someVar2 );

end valresDemo;
```

Here's *p*'s symbol table for this code:

```
p          <0,proc>:Procedure type (ID=p__hla_1) parms:8
-----
_ vars_    <1,cons>:uns32, (4 bytes)  =0
_ parms_   <1,cons>:uns32, (4 bytes)  =8
a          <1,vrp >:int32, (4 bytes, ofs:12)
b          <1,vrp >:int32, (4 bytes, ofs:8)
_ finalize_ <1,val >:string, (0 bytes)  =""
_ initialize_ <1,val >:string, (0 bytes)  =""
p          <1,proc>:
-----
```

Note that there is only one set of *a* and *b* parameters in this symbol table and they refer to the addresses passed on the stack rather than to any local data. You are responsible for allocating the storage making a local copy of the data when the **@noframe** option is present. Essentially, when you have **valres** parameters in a procedure that has the **@noframe** option, the effect is the same as if you declared the parameters as pass by reference parameters.

Given the amount of code HLA generates to copy pass by value/result parameters to and from the actual parameter locations, you might question whether using pass by value/result is very efficient. As it turns out, if you're accessing the value/result objects frequently in a program, you can quickly recoup the cost of the data copy operation with the more efficient access to a local variable (versus the indirect access that takes place with a reference variable).

Pass by result parameters are very similar to pass by value/result. The only difference is that HLA-generated code (when **@frame** is active) does not copy any data into the local copy of the parameter variable. Pass by result parameters are more efficient than pass by value/result if you are only using such parameters to return a value to the caller.

Note that HLA uses callee-copying semantics for pass by value/result and pass by result parameters. It's also possible to use caller-copying semantics (though HLA doesn't directly support this). The way to achieve caller-copy semantics is to pass the parameters by value but not remove them from the stack upon return. When the procedure returns to the caller, the caller can pop the data off the stack and store it into the original actual parameter locations. Obviously, this scheme generates a lot more code if you call the procedure a large number of times, but it can be slightly more efficient in certain cases because the procedure won't need to preserve registers it uses to copy the data to and from the local copy of the parameter data.

11.14 Parameter Passing in HLA, Name and Lazy Evaluation Parameters

HLA provides a modicum of support for pass by name and pass by lazy evaluation parameters. A pass by name parameter consists of a thunk that returns the address of the actual parameter. A pass by lazy evaluation parameter is a thunk that returns the value of the actual parameter. Whenever you specify the **name** or **lazy** keywords before a parameter, HLA reserves eight bytes to hold the corresponding thunk in the activation record. It is your responsibility to create a thunk whenever calling the procedure.

There is a minor difference between passing a thunk parameter by value and passing a lazy evaluation or name parameter to a procedure. Pass by name/lazy parameters require an immediate thunk constant; you cannot pass a thunk variable as a pass by name or lazy parameter.

To pass a thunk constant as a parameter to a pass by name or pass by lazy evaluation parameter, insert the thunk's code inside `"#{...}#" sequence in the parameter list and preface the`

whole thing with the **thunk** reserved word. The following example demonstrates passing a thunk as a pass by name parameter:

```

program nameDemo;
#include( "stdio.hhf" );

    procedure passByName( name ary:int32; var ip:int32 );
    @nodisplay;
    const i:text := "(type int32 [ebx])";
    const a:text := "(type int32 [eax])";
    begin passByName;

        mov( ip, ebx );
        mov( 0, i );
        while( i < 10 ) do

            ary(); // Get address of "ary[i]" into eax.
            mov( i, ecx );
            mov( ecx, a );
            inc( i );

        endwhile;

    end passByName;

    procedure thunkParm( t:thunk );
    begin thunkParm;

        t();

    end thunkParm;

var
    index:int32;
    array:int32[10];
    th:thunk;

begin nameDemo;

    thunk th := #{ stdout.put( "Thunk Variable",nl ) }#;
    thunkParm( th );
    thunkParm( thunk #{ stdout.put( "Thunk Constant" nl ); }# );

    // passByName( th, index ); -- would be illegal;

passByName
(
    thunk
    #{
        push( ebx );
        mov( index, ebx );
        lea( eax, array[ebx*4] );
        pop( ebx );
    }#,
    index
);

```

```

mov( 0, ebx );
while( ebx < 10 ) do

    stdout.put
    (
        "array[",
        (type int32 ebx),
        "]= ",
        array[ebx*4],
        nl
    );
    inc( ebx );

endwhile;

end nameDemo;

```

This program produces the following output:

```

Thunk Variable
Thunk Constant
array[0]=0
array[1]=1
array[2]=2
array[3]=3
array[4]=4
array[5]=5
array[6]=6
array[7]=7
array[8]=8
array[9]=9

```

The main purpose of pass by name and pass by lazy evaluation parameters is to support deferred parameter evaluation. When you pass a value, reference, value/result, or result parameter to a procedure, the value of that parameter is evaluated exactly once, at the point of the procedure call. When you pass a pass by name or pass by lazy evaluation parameter to a procedure, you're passing a thunk that is called to evaluate the parameter's value whenever you want to reference that parameter's value. From HLA's perspective, pass by name and pass by lazy evaluation parameters are implemented exactly the same way. The intent is that the respective thunks for this parameters return the address (pass by name) of the object or the value (pass by lazy evaluation) of the actual parameter object.

11.15 Hybrid Parameter Passing in HLA

HLA's automatic code generation for parameters specified using the high-level language syntax isn't always optimal. In fact, sometimes it is downright inefficient (though, to be fair, the code generation has gotten much better over the past decade; there are only a few degenerate examples we can draw from today). This is because HLA makes very few assumptions about your program. For example, suppose you are passing a word parameter to a procedure by value. Since all parameters in HLA consume an even multiple of four bytes on the stack, HLA will zero extend the word and push it onto the stack. It does this using code like the following:

```

pushw    0
pushw    Parameter

```

Clearly, you can do better than this if you know something about the variable. For example, if you know that the two bytes following *Parameter* are in memory (as opposed to being in the next page of memory that isn't allocated, and access to such memory would cause a protection fault), you could get by with the single instruction:

```
push    dword ptr Parameter
```

Unfortunately, HLA cannot make these kinds of assumptions about the data because doing so could create malfunctioning code (actually, if *Parameter* is an automatic variable or a parameter passed in from some other call, then HLA will make this assumption - this is an example of how HLA's code generation has improved over the years).

One solution, of course, is to forego the HLA high-level language syntax for procedure calls and manually push all the parameters your self and call the procedure via the **call** instruction. However, this is a major pain involving lots of extra typing and produces code that is difficult to read and understand. Therefore, HLA provides a hybrid parameter passing mechanism that lets you continue to use a high-level language calling syntax yet still specify the exact instructions needed to pass certain parameters. This hybrid scheme works out well because HLA actually does a good job with most parameters (e.g., if they are an even multiple of four bytes, HLA generates efficient code to pass the parameters; it's only those parameters that have a weird size that HLA generates less than optimal code for).

If a parameter consists of the "#{" and "}" brackets with some corresponding code inside the brackets, HLA will emit the code inside the brackets in place of any code it would normally generate for that parameter. So if you wanted to pass a 16-bit parameter efficiently to a procedure named "Proc" and you're sure there is no problem accessing the two bytes beyond this parameter, you could use code like the following:

```
Proc( #{ push( (type dword WordVar) ); }# );
```

Notice the similarity to pass by name/eval parameters. However, no **think** reserved word prefaces the code in this instance.

Whenever you pass a non-static¹ variable as a parameter by reference, HLA generates the following (MASM-syntax) code to pass the address of that variable to the procedure:

```
pusheax
pusheax
lea    eax, Variable
mov    [esp+4], eax
pop    eax
```

It generates this particular code to ensure that it doesn't change any register values (after all, you could be passing some other parameter in the EAX register). If you have a free register available, you can generate slightly better code using a calling sequence like the following (assuming EBX is free):

```
HasRefParm
(
    #{
        lea( ebx, Variable );
        push( ebx );
    }#
);
```

Note that HLA will generate slightly better code for automatic variables and parameters that don't have an indexed addressing mode attached to them. Examining the HLA output code (using -

1. Static variables are those you declare in the static, readonly, and storage sections. Non-static variables include parameters, VAR objects, and anonymous memory locations.

source and `-hla` command-line options) is a good way to see exactly what HLA is doing with your procedure calls.

11.16 Parameter Passing in HLA, Register Parameters

HLA provides a special syntax that lets you specify that certain parameters be passed in registers rather than on the stack. The following are some examples of procedure declarations that use this feature:

```
procedure a( u:uns32 in eax ); forward;
procedure b( w:word in bx ); forward;
procedure d( c:char in ch ); forward;
```

Whenever you call one of these procedures, the code that HLA automatically emits for the call will load the actual parameter value into the specified register rather than pushing this value onto the stack. You may specify any general-purpose 8-bit, 16-bit, or 32-bit register after the **in** keyword following the parameter's type. Obviously, the parameter must fit in the specified register. You may only pass reference parameters in 32-bit registers; you cannot pass parameters that are not one, two, or four bytes long in a register.

You can get in to trouble if you're not careful when using register parameters, consider the following two procedure definitions:

```
procedure one( u:uns32 in eax; v:dword in ebx ); forward;
procedure two( a:uns32 in eax );
begin two;

    one( 25, a );

end two;
```

The call to *one* in procedure *two* looks like it passes the values 25 and whatever was passed in for *a* in procedure *two*. However, if you study the HLA output code, you will discover that the call to *one* passes 25 for both parameters. They reason for this is because HLA emits the code to load 25 into EAX in order to pass 25 in the *u* parameter. Unfortunately, this wipes out the value passed into *two* in the *a* variable, hence the problem. Be aware of this if you use register parameters often.

11.17 Instruction Composition and Parameter Passing in HLA

You can use HLA's instruction composition feature in calls to HLA procedures. Consider the following simple example:

```
program instrCompDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( var a:dword ); @noframe;
    begin p;
    end p;

var
    someVar:int32;

begin instrCompDemo;

    p( [lea( eax, someVar)] );

end instrCompDemo;
```


You can also use instruction composition to compute certain expression values to pass as value parameters to an HLA procedure:

```

program instrCompDemo;
?@nodisplay := true;
?@nostackalign:= true;

    procedure p( a:dword ); @noframe;
    begin p;
    end p;

var
    i   :int32;
    j   :int32;

begin instrCompDemo;

// Pass i+j as the value parameter:

p( add( mov( i, eax ), mov( j, ebx ) ) );

end instrCompDemo;

```

Here's the code that HLA generates for the procedure call in the last example:

```

mov( [ebp-8], eax );
mov( [ebp-12], ebx );
add( eax, ebx );
push( ebx );
call p__hla_1;

```

The `HLA returns` statement provides another mechanism for using instruction composition to pass parameters to an HLA function. Here is the basic syntax for the `returns` statement:

```
returns( { << HLA statements >> }, "string" )
```

HLA will compile the statements between the braces and then return the string operand as the "returns" value for the entire construct. If this string contains an x86 register, a memory location, or an appropriate constant, HLA will emit the appropriate push instruction for that string operand if the `returns` statement appears as a procedure parameter. Here is an example of this usage:

```

// Pass i+j as the value parameter:

p
(
    returns
    (
        {
            mov( i, eax );
            mov( j, ebx );
            add( eax, ebx );
        },
        "ebx"
    )
);

```

Note that this example emits the same code as the previous example. Though this example is a bit longer, it's also much easier to read and comprehend.

11.18 Lexical Scope

HLA is a block-structured language that enforces the scope of local identifiers. HLA uses lexical scope to determine when and where an identifier is visible to the program. Identifiers declared within a procedure are always visible within that procedure and to any local procedures declared after the identifier. Local identifiers are never visible outside the procedure declaration. The scoping rules are similar to languages like Pascal, Ada, and Modula-2. As an example, consider the following code:

```
program scopeDemo;

#include( "stdio.hhf" );

var
  i:int32;
  j:int32;
  k:int32;

  procedure lex1;
  var
    i:int32;
    j:int32;

    procedure lex2;
    var
      i:int32;
    begin lex2;

      mov( i, eax ); //1
      mov( ebx:j, eax ); //2
      mov( ecx:k, eax ); //3

    end lex2;

  begin lex1;

    mov( i, eax ); //4
    mov( j, eax ); //5
    mov( ecx:k, eax ); //6

  end lex1;

  procedure alsolex1;
  var
    i:int32;
    m:int32;
  begin alsolex1;

    mov( i, eax ); //4
    mov( m, eax ); //5
    mov( ecx:k, eax ); //6

  end alsolex1;
```

```

begin scopeDemo;

    mov( i, eax );           //7
    mov( j, eax );           //8
    mov( k, eax );           //9

end scopeDemo;

```

Note: the purpose of the `ebx::` and `ecx::` prefixes on certain variables will become clear in a moment. Also note that this code is not functional, it was written only as an illustration.

In this example, you will note that `lex2` is nested within `lex1`, which is nested within the main program. The `alsolex1` procedure is nested within the main program but inside no other procedure. To describe this arrangement, compiler writers use the term *lex level* to denote the depth of nesting. HLA defines the main program to be *lex level zero*. Each time you nest a procedure you increase its lex level. So `lex1` is at lex level one since it is directly nested inside the main program at lex level zero. The `lex2` procedure is at lex level two because it is nested inside the `lex1` procedure. Finally, `alsolex1` is also at lex level one because it is nested inside the main program (which is lex level zero).

Within a given procedure (or the main program), all identifiers must be unique. That is, you cannot have two symbols named *i* in the same procedure. In different procedures, however, you may reuse the names. If all procedures were written at lex level one, then no procedure would be able to directly access the local variables in any other procedure (this is the case with the C/C++ language). In block-structured languages, like HLA, it is possible to access certain non-local variables in other procedures if the current procedure (whose code is attempting to access said variable) is nested within the other procedure.

In the example above, `lex2` accesses three variables: `i`, `j`, and `k`. The `i` variable is local to `lex2`, so there is nothing surprising here. The `j` variable is local to `lex1` and global to `lex2`. Likewise, the `k` variable is global to both `lex1` and `lex2` yet `lex2` can access it. Whenever a procedure is nested within another (either directly or indirectly), the nested procedure can access all variables in the global, nesting, procedures (including the main program)¹ unless the procedure declares a local name with the same name as a global name (the local name always takes precedence in this case). The term "scope" refers to the visibility of these names.

Being able to use a name during compilation is one thing, accessing the memory location associated with that name at run-time is something else entirely. Most block-structured high-level languages (HLLs) emit lots of extra code to access these "intermediate" and global variables for you. Why the extra code? Well remember, local procedure variables are accessed on the stack by indexing off the EBP register (which points at a procedure's "activation record"). When a procedure like `lex1` above calls a local procedure like `lex2`, the `lex2` procedure promptly saves the value in EBP (that points at `lex1`'s activation record) and it points EBP at the new activation record for `lex2`. Unfortunately, `lex2` no longer has access to `lex1`'s local variables since EBP no longer points at `lex1`'s locals. This creates a bit of a problem.

"But wait!" you exclaim. "EBP is pointing at the pointer to `lex1`'s activation record, why not just use double indirection to get the pointer to `lex1`'s locals?" This is a good idea, but it fails if `lex2` is recursive. There are two or three general solutions to this problem; HLA uses a *display* to access non-local values.

A *display* is nothing more than an array of pointers. `Display[0]` is a pointer to the most recent activation record at lex level zero, `Display[1]` is a pointer to the most recent activation record at lex level one, `Display[2]` is a pointer to the most recent activation record at lex level two, etc. (note the use of the phrase *most recent*. This ensures that displays work properly even when recursion occurs). With a *display*, to access a non-local variable, you just go to the memory location specified by `Display[varlex] + varoffset` where `varlex` is the lex level of the symbol you wish to access and `varoffset` is the offset into the activation record where the variable's data can be found.

Sound complex? Actually, HLA simplifies this quite a bit. First, as long as you don't specify the `@nodisplay` procedure option, HLA automatically emits the code to build a *display* at the

1. Strictly speaking, this isn't true. The nested procedure has access to all global variables that were declared before the procedure's declaration.

start of the procedure's code¹. HLA also defines a run-time variable, `_display_`, that points at (the end of) this array of pointers. To access a non-local variable requires two instructions, one to fetch the address of the variable's activation record and one to access the variable. Correcting the previous program, the code would look something like this:

```

procedure lex2;
var
  i:int32;
begin lex2;

  mov( i, eax );

  // access non-local variable j
  // at lex level 1.

  mov( _display_[-1*4], ebx );
  mov( ebx::j, eax );

  // access non-local variable k
  // at lex level 0.

  mov( _display_[0], ecx );
  mov( ecx::k, eax );

end lex2;

```

There are two things to note about the display: first, the entries are stored at negative indices in the array (0, -1, -2, etc) rather than at positive indices (this is due to Intel's implementation of displays). Second, don't forget that this is a run-time array of double-words so you must multiply each index by the array element size, which is four in this case.

Once you've loaded the address into a register, the `reg::var` syntax tells HLA to use the specified register rather than EBP as the pointer to the variable's activation record. The "mov(ecx::k,eax);" instruction, for example, compiles to "mov eax, [ecx+koffset]" where `koffset` represents the offset of `k` in the main program's activation record.

In general, few programs take advantage of nested procedures and access to local variables, so it is very common to find programmers putting `@nodisplay` after all their procedures. Of course, if you do this, HLA does not generate display and access to non-local variables (declared in the `var` section) is not possible. Of course, static variables are not allocated in the activation record, so you always have access to non-local static variables even if you don't generate the code for a display.

1. It is important that all nested procedures construct the display. You couldn't use the `@nodisplay` option in `lex1` and expect `lex2` to properly build the display. In general, unless you know exactly what you are doing, your procedures should all have the `@nodisplay` option, or none of them should have it.

12 HLA Classes and Object-Oriented Programming

12.1 Class Data Types

HLA supports object-oriented programming via the class data type. A class declaration takes the following form:

```
class
    << declarations >>
endclass;
```

Classes allow **const**, **val**, **var**, **static**, **readonly**, **storage**, **procedure**, **iterator**, and **method** declarations. In general, just about everything allowed in a program declaration section except **labels**, **types**, and **namespaces** are legal in a class declaration.

Unlike C++ and Object Pascal, where the class declarations are nearly identical to the record/struct declarations, HLA class declarations are noticeably different than HLA records because you supply **const**, **var**, **static**, etc., declaration sections within the class. As an example, consider the following HLA class declaration:

```
type
    SomeClass:
        class

            var
                i:int32;

            const
                pi:=3.14159;

            method incrementI;

        endclass;
```

Unlike records, you must put each declaration into an appropriate section. In particular, data fields must appear in a **static**, **readonly**, **storage**, or **var** section.

Note that the body of a procedure or method does not appear in the class declaration. Only prototypes (forward declarations) appear within the class definition itself. The actual procedure or method is declared elsewhere in the code.

12.2 Classes, Objects, and Object-Oriented Programming in HLA

HLA provides support for object-oriented program via classes, objects, and automatic method invocation. Indeed, supporting method calls requires HLA to violate an important design principle (that HLA generated code does not disturb values in any registers except ESP and EBP). Nevertheless, supporting object-oriented programming and automatic method calls was so important, an exception was made in this instance. More on that in a moment.

It is worthwhile to review the syntax for a class declaration. First, class declaration may only appear in a **type** section within an HLA program. You cannot define classes in the **var**, **static**, **storage**, or **readonly** sections and HLA does not allow you to create class constants¹. Within the **type** section, a class declaration takes one of the following forms:

type

```

baseClass:
    class
        Declarations, including const,
        val, var, and static sections, as
        well as procedures, methods, and
        macros.
    endclass;

derivedClass:
    class inherits( baseClass )
        Declarations, including const,
        val, var, and static sections, as
        well as procedure and method prototypes, and
        macros.
    endclass;

```

Note that you may not include **type** sections or **namespace** sections in a class. Allowing **type** sections in a class creates some special problems (having to do with the possibility of nested class definitions). Name spaces are illegal because they allow **type** sections internally (and there is no real need for name spaces within a class).

Note that you may only place **procedure**, **iterator**, and **method** prototypes in a class definition. Procedure and method prototypes look like a forward declaration without the forward reserved word; they use the following syntax:

```

procedure procName(optional_parameters); options
method methodName(optional_parameters); options
iterator iterName( optional_parameters ); optional_external

```

procName, *iterName*, and *methodName* are the names you wish to assign to these program units. Note that you do *not* preface these names with the name of the class and a period.

If the procedure, iterator, or method has any parameters, they immediately following the procedure/iterator/method name enclosed in parentheses. The parentheses must not be present if there are no parameters. A semicolon immediately follows the parameters, or the procedure/method name if there are no parameters.

12.3 The THIS and SUPER Reserved Words

Within a class method, procedure, or iterator, you will often need to access one of the class fields of the current object. Upon entry into a class method or iterator, the ESI register will always be pointing at the class object's data. Upon entry into a class procedure, the ESI register will either contain NULL (if you call the class procedure directly, specifying the class name rather than an object name) or a pointer to the object's data (if you call the class procedure using an object name or object pointer name). You can use HLA's type coercion operation to access the object's data fields or call other methods in the class, e.g.,:

```

method someClass.SomeMethod;
begin SomeMethod;

    mov( (type someClass [esi]).someField, eax );
    (type someClass [esi]).someOtherMethod( eax );

```

-
1. Of course, you may create class variables (objects) by specifying the class type name in the var or static sections.

```
end SomeMethod;
```

Of course, you must take care not to overwrite the value passed in ESI to the method (or iterator or procedure) when using it in this fashion.

HLA offers a special reserved word, **this**, that simplifies accessing fields of the current object. The **this** keyword automatically expands to “(type current_object_class [esi])”, so you could write the previous code thusly:

```
method someClass.SomeMethod;
begin SomeMethod;

    mov( this.someField, eax );
    this.someOtherMethod( eax );

end SomeMethod;
```

Note that calling a class function associated with any other object will load ESI with the address of that object’s data; so if you make such a call within a method the current value in ESI may be replaced. Using **this** after such a call will produce undefined results:

```
method someClass.aMethod;
begin aMethod;

    someOtherObject.itsMethod( 0 );
    mov( this.someField, eax ); // Incorrect! ESI is wiped out!

end aMethod;
```

On occasion, a method may need to call the base class’ version of that method in order to handle some operations done by the base class. The intent might be like the following (incorrect example):

```
method derivedClass.someFunction;
begin someFunction;

    // Attempt to call the base class’ method:

    (type baseClass [esi]).someFunction();

    // Do some work specific to this class:
    .
    .
    .

end someFunction;
```

This won’t work as intended. The code above will likely end up in an infinite loop because the current object’s virtual method table (VMT) entry for **someFunction** points at the **derivedClass.someFunction** method. Simply coercing the type of [esi] won’t change this (indeed, this is how polymorphism in object-oriented programming works). If you really want to call the base class’ method, you should use the **super** keyword. The **super** keyword is similar to **this** except that it is only valid for method calls. Consider the following example:

```
method derivedClass.someFunction;
begin someFunction;
```

```

// Attempt to call the base class' method:

super.someFunction();

// Do some work specific to this class:
.
.
.

end someFunction;

```

The difference between **this** and **super** is that the **super** keyword loads the EDI register (which points at the VMT) with the address of the base class' VMT rather than the current classes' VMT. This forces the call to the base class' method rather than to the current (derived) class' method. See the discussion of the **override** keyword later in the chapter for more details on derived and base class methods.

12.4 Class Procedure and Method Prototypes

Class procedure and method prototypes allow two options: an **@returns** clause and/or an **external** clause. The **@pascal**, **@cdecl**, **@stdcall**, **@nodisplay** and **@noframe** options are not allowed in the prototype. See the section on procedures for more details on the **@returns** and **external** clauses. The iterator only allows the **external** option.

You can also use new style procedure declarations in an HLA class to declare procedures, iterators, and macros. Here is a simple example of a class using the new style syntax:

```

type
  myClass:
    class

      proc
        classProc:procedure( i:int32 );
        classMethod:method( j:int32 );
        classIterator:iterator( k:int32 );
      endproc;

    endclass;

```

Unlike procedures and methods, if you define a macro within a class you must supply the body of the macro within the class definition.

Consider the following example of a class declaration:

```

type
  baseClass:
    class

      var
        i:int32;

      procedure create; @returns( "esi" );
      procedure geti; @returns( "eax" );
      method seti( ival:int32 ); @external;

    endclass;

```


By convention, all classes should have a class procedure named `create`. This is the constructor for the class. The `create` procedure should return a pointer to the class object in the ESI register, hence the `@returns("esi");` clause in this example.

This procedure includes two accessor functions, `geti` and `seti`, that provide access to the class variable `i`. Note that HLA classes do not support the public, private, and protected visibility options found in HLLs like C++ and Delphi. HLA's design assumes that assembly language programmers are sufficiently disciplined such that they will not access fields that should be private¹.

Of course, the class' procedures and methods must be defined at one point or another. Here are some reasonable examples of these class definitions (a full explanation will appear later):

```
procedure baseClass.create;
begin create;

    push( eax );
    if( esi = 0 ) then

        malloc( @size( baseClass ) );
        mov( eax, esi );

    endif;
    mov( baseClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

end create;

procedure baseClass.geti; @nodisplay; @noframe;
begin geti;

    mov( this.i, eax );
    ret();

end geti;

method baseClass.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );
    mov( eax, this.i );
    pop( eax );

end seti;
```

These procedure and method declarations look almost like regular procedure declarations with one important difference: the class name and a period precede the procedure or method name on the first line of the procedure/method declaration. Note, however, that only the procedure or method name appears after the **begin** and **end** clauses.

Another important difference is the procedure options. Only the `@nodisplay/@display`, `@noalignstack/@alignstack`, and `@noframe/@frame` options are legal here (the converse

1. Actually, HLA was designed this way because far too often programmers make fields private and other programmers decide they really needed access to those fields, software engineering be damned. HLA relies upon the discipline of the programmers to stay out of trouble on this matter.

of the class procedure/method prototype definitions which only allow `external` and `@returns`). Note that class procedures, methods, and iterators do not support the `@pascal`, `@cdecl`, or `@stdcall` procedure options (they always use the Pascal calling convention).

Class procedures and methods must be defined at the same lex level and within the same scope as the class declaration. Usually class declarations are a lex level zero (i.e., inside the main program or within a unit), so the corresponding procedure and method declarations must appear at lex level zero as well. Of course, it is legal to declare a class type within some other procedure (at lex level one or higher). If you do this, the class procedure and method declarations must appear at the same level.

Note that class declarations also support the new procedure declaration syntax with a `proc` section. Here is the previous example using the new style procedure declarations:

```

type
  baseClass:
    class

        var
            i:int32;

        proc
            create :procedure { @returns( "esi" ) };
            geti  :procedure { @returns( "eax" ) };
            seti  :method( ival:int32 ); external;

    endclass;

proc
  baseClass.create: procedure;
  begin create;

    push( eax );
    if( esi = 0 ) then

        malloc( @size( baseClass ) );
        mov( eax, esi );

    endif;
    mov( baseClass._VMT_, this._pVMT_ );
    pop( eax );
    ret();

  end create;

  baseClass.geti :procedure: @nodisplay @noframe;
  begin geti;

    mov( this.i, eax );
    ret();

  end geti;

  baseClass.seti :method( ival:int32 ); @nodisplay;
  begin seti;

    push( eax );
    mov( ival, eax );
    mov( eax, this.i );

```

```

    pop( eax );

end seti;

```

12.5 Inheritance

HLA classes support inheritance using the **inherits** reserved word. Consider the following class declaration that inherits the fields from the *baseClass* declaration in the previous section:

```

derivedClass:
    class inherits( baseClass )

        var
            j:int32;
            f:real64;

    endclass;

```

This class inherits all the fields from *baseClass* and adds two new fields, *j* and *f*. This declaration is roughly equivalent to:

```

derivedClass:

    var
        i:int32;

    procedure create; @returns( "esi" );
    procedure geti; @returns( "eax" );
    method seti( ival:int32 ); @external;

    var
        j:int32;
        f:real64;

    endclass;

```

It is "roughly" equivalent because there is no need to create the `derivedClass.create` and `derivedClass.geti` procedures or the `derivedClass.seti` method. This class inherits the procedures and methods written for *baseClass* along with the field definitions.

Like records, it is possible to "override" the **var** fields of a base class in a derived class. To do this, you use the **overrides** keyword. Note that this keyword is valid only for **var** fields in a class, you may not override static objects with this keyword. Example:

```

derivedClass:
    class inherits( baseClass )

        procedure create; @returns( "esi" );
        procedure geti; @returns( "eax" );
        method seti( ival:int32 ); @external;

    var
        overrides i: dword;    // New copy of i for this class.
        j:int32;

```

```

        f:real64;

    endclass;

```

While on the subject of class **var** objects, you should be aware that class **var** objects are not (necessarily) allocated on the stack in an activation record as are local **var** variables in a **procedure**, **method**, or **iterator**. Class **var** objects are allocated in storage associated with a class object, that actual memory could be on the stack, in static memory, or on the heap.

Occasionally, you may want to override a procedure in a base class. For example, it is very common to supply a new constructor in each derived class (since the constructor may need to initialize fields in the derived class that are not present in the base class). The `override`¹ keyword tells HLA that you intend to supply a new procedure or method declaration and you do not want to call the corresponding functions in the base class. Consider the following modifications to `derivedClass` that override the `create` procedure and `seti` method:

```

derivedClass:
    class inherits( baseClass )

        var
            j:int32;
            f:real64;

        override procedure create;
        override method seti;

    endclass;

```

When you override a procedure or method, you are not allowed to specify any parameters or procedure options except the `external` option. This is because the parameters and `@returns` strings must exactly match the declarations in the base class. So even though `seti` in this derived class doesn't have an explicit parameter declared, the `ival` parameter is still required in a call to `seti`.

Of course, once you override procedures and methods in a derived class, you must provide those program units in your code. Here is an example of a section of a program that provides overridden procedures and methods along with their declarations:

```

type

    base:    class

        var
            i:int32;

        procedure create;
        method geti;
        method seti( ival:int32 );

    endclass;

    derived:class inherits( base )

        var

```

1. Note that the syntax is `override`, not `overrides` as is used for overriding data fields. This is an unfortunate consequence of HLA's grammar.

```
        j:int32;

        override procedure create;
        override method setj;

        method getj;
        method setj( jval:int32 );

    endclass;

procedure base.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        malloc( @size( base ));
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;

method base.geti; @nodisplay; @noframe;
begin geti;

    mov( this.i, eax );
    ret();

end geti;

method base.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );
    mov( eax, this.i );
    pop( eax );

end seti;

procedure derived.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );
```

```
endif;

// Do any initialization done by the base class:
call base.create;

// Do our own specific initialization.
mov( &derived._VMT_, this._pVMT_ );
mov( 1, this.j );

// Return
pop( eax );
ret();

end create;

method derived.seti( ival:int32 ); @nodisplay;
begin seti;

    push( eax );
    mov( ival, eax );

    // call inherited code to do whatever it does:
    (type base [esi]).seti( ival );

    // Now handle the code that we do specially.
    mov( eax, this.j );

    // Okay, return to caller.
    pop( eax );

end seti;

method derived.setj( jval:int32 ); @nodisplay;
begin setj;

    push( jval );
    pop( this.j );

end setj;

method derived.getj; @nodisplay; @noframe;
begin getj;

    mov( this.j, eax );
    ret();

end getj;
```

12.6 Abstract Methods

Sometimes you will want to create a base class as a template for other classes. You will never create instances (variables) of this base class, only instances of classes derived from this class. In object-oriented terminology, we call this an *abstract* class. Abstract classes may contain certain methods that will always be overridden in the derived classes. Hence, there is no need to actually supply the method for this base class. HLA, however, always checks to verify that you supply all methods associated with a class. Therefore, you normally have to supply some sort of method, even if it's just an empty method, to satisfy the compiler. In those instances where you really don't need such a method, this is an annoyance. HLA's *abstract methods* provide a solution to this problem.

You declare an abstract method in a class declaration as follows:

```
type
  c: class

      method absMethod( parameters: uns32 ); abstract;

      proc
        anotherAbsMethod:method( parms:uns32 ) {@returns( "eax" )};
abstract;

      endclass;
```

The `abstract` keyword must follow the `@returns` option if the `@returns` option is present. In the new style procedure syntax, the `abstract` option must follow the declaration.

The `abstract` keyword tells HLA not to expect an actual method associated with this class. Instead, it is the responsibility of all classes derived from "c" to override this method. If you attempt to call an abstract method, HLA will raise an exception and abort program execution.

12.7 Classes versus Objects

An *object* is an instance of a class. In plain English, this means that a class is only a data type while an object is a variable whose type is some class type. Therefore, actual objects may be declared in the **var**, **static**, **readonly**, or **storage** declaration section. Here are a couple of typical examples:

```
var
  b: base;

static
  d: derived;
```

Each of these declarations reserves storage for all the data in the specified class type.

For reasons that will shortly become clear, most programmers use pointers to objects rather than directly declared objects. Pointer declarations look like the following:

```
var
  ptrToB: pointer to base;

static
  ptrToD: pointer to derived;
```

Of course, if you declare a pointer to an object, you will need to allocate storage for the object (call the HLA Standard Library `mem.alloc` routine) and initialize the pointer variable with the

address of the allocated storage. As you will soon see, the class constructor typically handles this allocation for you.

12.8 Initializing the Virtual Method Table Pointer

Whether you allocate storage for an object statically (in the **static** section), automatically (in the **var** section), or dynamically (via a call to `mem.alloc`), it is important to realize that the object is not properly initialized and must be initialized before making any method calls. Failure to do so will most likely cause your program to crash when you attempt to call a method or access other data in the class.

The first four bytes of every object contain a pointer to that object's *virtual method table*. The virtual method table, or VMT, is an array of pointers to the code for each method in the class. To help you initialize this pointer, HLA automatically adds two fields to every class you create: `_VMT_` which is a static double-word entry (the significance of this being a static entry will become clear later) and `_pVMT_` which is a **var** field of the class whose type is pointer to dword. `_pVMT_` is where you must put a pointer to the virtual method table. The pointer value to store here is the address of the `_VMT_` entry. This initialization can be done using the following statement:

```
mov( &ClassName._VMT_, ObjectName._pVMT_ );
```

`ClassName` represents the name of the class and `ObjectName` represents the name of the **static** or **var** variable object. If you've allocated storage for an object pointer using `mem.alloc`, you'd use code like the following:

```
mov( ObjectPtr, ebx );
mov( &ClassName._VMT_, (type ClassName [ebx])._pVMT_ );
```

In this example, `ObjectPtr` represents the name of the pointer variable. `ClassName` still represents the name of the class type.

Typically, the initialization of the pointer to the virtual method table takes place in the class' constructor procedure (it must be a procedure, not a method!). Consider the example from the previous section:

```
procedure base.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;
```

As you can see here, this example uses the keyword `this._pVMT_` rather than `(type derived [esi])._pVMT_`. That's because **this** is a shorthand for using the ESI register as a pointer to an object of the current class type.

12.9 Creating the Virtual Method Table

For various technical reasons (related to efficiency), HLA does not automatically create the virtual method table for you; you must explicitly tell HLA to emit the table of pointers for the virtual method table. You can do this in either the **static** or the **readonly** declaration sections. The simple way is to use a statement like the following in either the **static** or **readonly** section:

```
VMT( classname );
```

If you intend to reference a VMT outside the source file in which you declare it, you can use the **external** option to make the symbol accessible, e.g.,

```
VMT( classname ); external;
```

Note that an external declaration of this form is optional. HLA always makes the VMT name for a class an external symbol. If you actually declare the VMT (using the first declaration above), HLA also makes the VMT symbol public.

If you need to be able to access the pointers in this table, there are two ways to do this. First, you can refer to the **classname.VMT_** double-word variable in the class. Another way is to directly attach a label to the VMT you create using a declaration like the following:

```
vmtLabel: VMT( classname );
```

The `vmtLabel` label will be a static object of type **dword**.

As for unnamed VMT declarations, HLA will automatically make the VMT symbol (and the `vmtLabel` symbol) external and public. If you want to explicitly specify a named external VMT declaration, you can do so with either of the following statements:

```
vmtLabel: VMT( classname ); external;
vmtLabel: VMT( classname ); external( "externalVmtLabelName" );
```

12.10 Calling Methods and Class Procedures

Once the virtual method table of an object is properly initialized, you may call the methods and procedures of that object. The syntax is very similar to calling a standard HLA procedure except that you must prefix the procedure or method name with the object name and a period. For example, assume you have some objects with the following types (*base* is the type in the examples of the previous sections):

```
var
  b: base;
  pb: pointer to base;
```

With these variable declarations, and some code to initialize the pointers to the *base* virtual method table, the calls to the *base* procedures and methods might look like the following:

```
b.create();
b.geti();
b.seti( 5 );

pb.create();
pb.geti();
pb.seti( eax );
```

Note that HLA uses the same syntax for an object call regardless of whether the object is a pointer or a regular variable.

Whenever HLA encounters a call to an object's procedure or method, HLA emits some code that will load the address of the object into the ESI register. **This is the one place HLA emits code that modifies the value in a general-purpose register!** You must remember this and not expect to be able to pass any values to an object's procedure or methods in the ESI register. Likewise, don't expect the value in ESI to be preserved across a call to an object's procedure or method. **As you will see shortly, HLA may also emit code that modifies the EDI register as well as the ESI register.** Therefore, don't count on the value in EDI, either.

The value in ESI, upon entry into the procedure or method, is that object's **this** pointer. This pointer is necessary because the exact same object code for a procedure or method is shared by all object instances of a given class. Indeed, the **this** reserved word within a method or class procedure is really nothing more than shorthand for "(type *ClassName* [esi])".

Perhaps an obvious question is "What is the difference between a class procedure and a method?" The difference is the calling mechanism. Given an object **b**, a call to a class procedure emits a call instruction that directly calls the procedure in memory. In other words, class procedure calls are very similar to standard procedure calls with the exception that HLA emits code to load ESI with the address of the object¹. Methods, on the other hand, are called indirectly through the virtual method table. Whenever you call a method, HLA actually emits three machine instructions: one instruction that load the address of the object into ESI, one instruction that loads the address of the virtual method table (i.e., the first four bytes of the object) into EDI, and a third instruction that calls the method indirectly through the virtual method table. For example, given the following four calls:

```
b.create();
b.geti();

pb.create();
pb.geti();
```

HLA emits the following 80x86 assembly language code:

```
lea( esi, [ebp-12]); //b
call classname.create;

lea( esi, [ebp-12] ); //b
mov( [esi], edi );
call( (type dword ptr [edi+geti_offset_in_VMT]); //geti

mov( [ebp-16], esi ); //pb
call classname.create

mov( [ebp-16], esi ); //b
mov( [esi], edi );
call((type dword [edi+geti_offset_in_VMT] ); //geti
```

HLA class procedures roughly correspond to C++'s *static member functions*. HLA's methods roughly correspond to C++'s *virtual member functions*. Read the next few sections on the impact of these differences.

If you call a method within some other method using the **super** keyword, the code does not fetch the VMT pointer from the current object. Instead, the code directly loads EDI with the address of the appropriate VMT:

1. When calling a class procedure, HLA never disturbs the value in the EDI register. EDI is only tweaked when you call methods.

```
super.someMethod();
```

generates x86 code like the following:

```
lea( edi, baseClass_VMT );
call( (type dword ptr [edi+methodOffsetInVMT]));
```

12.11 Accessing VMT Fields

The VMT is basically an array of pointers. Offsets zero through $(n-1)*4$, where n is the number of methods in a class (including inherited methods), hold pointers to each of the methods associated with the class. The previous section described how HLA emits a call to a class method. You can manually do this by simulating the same code that HLA emits. The `@offset` compile-time function, when supplied with the name of a class method as its operand, will return an index into the VMT where the address of that method is found. Therefore, you could manually call a method using code like the following:

```
mov( objectPtr, esi ); // or lea( esi, objectVar );
mov( [esi], edi );    // Get VMT pointer into EDI
call( [edi+@offset( derivedClass.methodToCall )]);
```

In this example, `derivedClass` is the name of the class and `methodToCall` is the name of some method in that class. Note that you must supply the full `classname.methodname` identifier to the `@offset` compile-time function so HLA can properly identify the method. Of course, it's generally easier to call the method using `objectPtr.methodToCall`, but for those who insist on calling the method using low-level code, this is how it is done.

You might be tempted to streamline the code above to something like the following:

```
mov( objectPtr, esi ); // or lea( esi, objectVar );
call( derivedClass._VMT_[@offset( derivedClass.methodToCall )]);
```

Resist the temptation to do this at all costs! First, this defeats polymorphism; `objectPtr` might actually contain a pointer to some other class that was derived from `derivedClass`. The code immediately above will always call `derivedClass.methodToCall`, even if it actually should be calling `some_class_derived_from_derivedClass.methodToCall`. The former example will handle this correctly.

Before the `super` keyword was added to HLA, the accepted way to call a base class' version of some method was to manually call the method, as was done in the first example of this section (though `ESI` usually contained the `THIS/object` pointer, so you didn't normally need to load it into `ESI`).

In HLA v2.8 and v2.9, several new fields were added to the VMT at negative offsets from the VMT's base address. At offset -4 there is a pointer to the parent class' VMT (this field contains `NULL` if this is a base class that has no parent class). At offset -8 is the size, in bytes, of an object of the class' type. At offset -12 is a string object that contains the name of the class associated with the VMT. The HLA Standard Library `hla.hhf` header file contains a record definition you can use to access these fields in a VMT:

```
namespace hla;

vmtRec:
    record := -12;

        vmtName      :string;
        vmtSize      :uns32;
        vmtParent    :pointer to dword;

    endrecord;
```

```
end hla;
```

Using the record definition above, you could load the class' name into EAX with a statement like this:

```
mov( (type hla.vmtRec derivedClass._VMT_).vmtName, eax );
```

Don't forget to include the *hla.hhf* header file in order to gain access to the declaration of the `vmtRec` record.

12.12 Non-object Calls of Class Procedures

In addition to the difference in the calling mechanism, there is another major difference between class procedures and methods: you can call a class procedure without an associated object. To do so, you would use the class name and a period, rather than an object name and a period, in front of the class procedure's name. E.g.,

```
base.create();
```

Since there is no object here (remember, `base` is a type name, not a variable name, and types do not have any storage allocated for them at run-time), HLA cannot load the address of the object into the ESI register before calling `create`. This situation can create some big problems in your code if you attempt to use the **this** pointer within a class procedure. Remember, an instruction like "`mov(this.i, eax);`" really expands to "`mov((type base [esi]).i, eax);`" The question that should come to mind is "where is ESI pointing when one makes a non-object call to a class procedure?"

When HLA encounters a non-object call to a class procedure, HLA loads the value zero (NULL) into ESI immediately before the call. Therefore, ESI doesn't contain junk but it does contain the NULL pointer. If you attempt to dereference NULL (e.g., by accessing `this.i`) you will probably bomb the program. Therefore, to be safe, you must check the value of ESI inside your class procedures to verify that it does not contain zero.

The `base.create` constructor procedure demonstrates a great way to use class procedures to your advantage. Take another look at the code:

```
procedure base.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ) );
        mov( eax, esi );

    endif;

    mov( &base._VMT_, this._pVMT_ );
    mov( 0, this.i );
    pop( eax );
    ret();

end create;
```

This code follows the standard convention for HLA constructors with respect to the value in ESI. If ESI contains zero (NULL), this function will allocate storage for a brand new object, initialize that object, and return a pointer to the new object in ESI¹. On the other hand, if ESI

contains a non-null value, then this function does not allocate memory for a new object, it simply initializes the object at the address provided in ESI upon entry into the code.

Certainly, you do not want to use this trick (automatically allocating storage if ESI contains **NULL**) in all class procedures; but it's still a real good idea to check the value of ESI upon entry into every class procedure that accesses any fields using ESI or the **this** reserved word. One way to do this is to use code like the following at the beginning of each class procedure in your program:

```
if( ESI = NULL ) then
    raise( AttemptToDerefZero );
endif;
```

If this seems like too much typing, or if you are concerned about efficiency once you've debugged your program, you could write a macro like the following to solve your problem:

```
#macro ChkESI;
    #if( CheckESI )
        if( ESI = 0 ) then

            raise( AttemptToDerefZero );

        endif;
    #endif
#endmacro
```

Now all you have to do is stick an innocuous `ChkESI` macro invocation at the beginning of your class procedures (maybe on the same line as the **begin** clause to further hide it) and you're in business. By defining the boolean constant `CheckESI` to be **true** or **false** at the beginning of your code, you can control whether this "inefficient" code is generated into your programs.

12.13 Static Class Fields

There exists only one copy, shared by all objects, of any **static**, **readonly**, or **storage** data objects in a class. Since there is only one copy of the data, you do not access variables in the class' static section using the object name or the **this** pointer. Instead, you preface the field name with the class name and a period.

For example, consider the following class declaration that demonstrates a very common use of static variables within a class:

```
program DemoOverride;

#include( "memory.hhf" )
#include( "stdio.hhf" )
type

    CountedClass:
        class

            static
                CreateCnt:int32 := 0;
```

-
1. Of course, it is the caller's responsibility to save this pointer away into an object pointer variable upon return from the class procedure.

```
        procedure create;
        procedure DisplayCnt;

    endclass;

procedure CountedClass.create; @nodisplay; @noframe;
begin create;

    push( eax );
    if( esi = 0 ) then

        mem.alloc( @size( base ));
        mov( eax, esi );

    endif;
    mov( &CountedClass._VMT_, this._pVMT_ );
    inc( this.CreateCnt );
    pop( eax );
    ret();

end create;

procedure CountedClass.DisplayCnt; @nodisplay; @noframe;
begin DisplayCnt;

    stdout.put( "Creation Count=", CountedClass.CreateCnt, nl );
    ret();

end DisplayCnt;

var
    b: CountedClass;
    pb: pointer to CountedClass;

begin DemoOverride;

    CountedClass.DisplayCnt();

    b.create();
    CountedClass.DisplayCnt();

    CountedClass.create();
    mov( esi, pb );
    CountedClass.DisplayCnt();

end DemoOverride;
```

In this example, a static field (`CreateCnt`) is incremented by one for each object that is created and initialized. The `DisplayCnt` procedure prints the value of this static field. Note that `DisplayCnt` does not access any non-static fields of `CountedClass`. This is why it doesn't bother to check the value in `ESI` for zero.

There is a big issue with respect to static fields in a class. If you include the header file containing the class definition in more than one HLA source file (that is part of a single project), HLA will create one copy of the static object for each source file. This can produce linkage errors if you attempt to link those files together. The solution to this problem is to create an external symbol in the class declaration:

type

```
CountedClass:
    class

        static
            CreateCnt:int32;
            external( "CountedClass_CreateCnt" );

        procedure create;
        procedure DisplayCnt;

    endclass;
```

The external declaration in this example expects you to provide an external `int32` object named `CountedClass_CreateCnt`. You can do this (in one of the HLA source files) using code like the following:

```
static
    CreateCnt :int32; external( "CountedClass_CreateCnt" );
    CreateCnt :int32 := 0;
```

12.14 Taking the Address of Class Procedures, Iterators, and Methods

You can use the static address-of operator ("`&`") to obtain the memory address of a class procedure, method, or iterator by applying this operator to the class procedure/method/iterator's name with a *classname* prefix. E.g.,

type

```
c : class
    procedure p;
    method m;
    iterator i;
endclass;

procedure c.p; begin p; end p;
method c.m; begin m; end m;
iterator c.i; begin i; end i;

.
.
.
mov( &c.p, eax );
mov( &c.m, ebx );
mov( &c.i, ecx );
```

Please note that when you apply the address-of operator ("&") to a class procedure/method/iterator you must specify the class name, not an object name, as the prefix to the procedure/method/iterator name. That is, the following is illegal given the class definition for `c`, above:

```
static
  myClass: c;
  .
  .
  .
  mov( &myClass.p, eax );
```

12.15 Program Unit Initializers and Finalizers

HLA does not automatically call an object's constructor like C++ does. There is no code associated with a unit that automatically executes to initialize that unit as in (Turbo) Pascal or Delphi. Likewise, HLA does not automatically call an object's destructor. However, HLA does provide a mechanism by which you can automatically execute initialization and shutdown code without explicitly specifying the code to execute at the beginning and end of each procedure. This is handled via the HLA `_initialize_` and `_finalize_` strings. All programs, procedures, methods, and iterators have these two predeclared string constants (**val** strings, actually) associated with them. Whenever you declare a program unit, HLA inserts these constants into the symbol table and initializes them with the empty string.

HLA expands the `_initialize_` string immediately before the first instruction it finds after the **begin** clause for a program, procedure, iterator, or method. Likewise, it expands the `_finalize_` string immediately before the **end** clause in these program units. Since, by default, these string constants hold the empty string, they usually have no effect. However, if you change the values of these constants within a declaration section, HLA emits the corresponding code at the appropriate point. Consider the following example:

```
procedure HasInitializer;
  ?_initialize_ := "mov( 0, eax );";
begin HasInitializer;

  stdout.put( "EAX = ", eax, nl );

end HasInitializer;
```

This program will print "EAX = 0000_0000" since the `_initialize_` string contains an instruction that moves zero into EAX.

Of course, the previous example is somewhat irrelevant since you could have more easily put the **mov** instruction directly into the program. The real purpose of the initialize and finalize strings in an HLA program is to allow macros and include files to slip in some initialization code. For example, consider the following macro:

```
#macro init_int32( initValue ):theVar;

  :forward( theVar );
  theVar: int32
  ?_initialize_ = _initialize_ +
                  "mov( " +
                  @string:initValue +
                  ", " +
                  @string:theVar +
                  " );";

#endmacro
```

Now consider the following procedure:


```

procedure HasInitedVars;
var
  i: init_int32( 0 );
  j: init_int32( -1 );
  k: init_int32( 1 );

begin HasInitedVars;

  stdout.put( "i=", i, " j=", j, " k=", k, nl );

end HasInitedVars;

```

The first *init_int32* macro above expands to (something like) the following code:

```

i: forward( _1002_ );
_1002_: int32
?_initialize_ := _initialize_ +
                "mov( " +
                "0" +
                ", " +
                "i" +
                " );";

```

Note that the last statement is equivalent to:

```
?_initialize_ := _initialize_ + "mov( 0, i );"
```

Also note that the text object *_1002_* expands to "i".

If you take a step back from this code and look at it from a high level perspective, you can see that what it does is initialize a **var** variable by emitting a **mov** instruction that stores the macro parameter into the **var** object. This example makes use of the **forward** declaration clause in order to make a copy of the variable's name for use in the **mov** instruction. The following is a complete program that demonstrates this example (it prints "i=1", if you're wondering):

```

program InitDemo;
#include( "stdlib.hhf" )

  #macro init_int32( initVal ):theVar;

    forward( theVar );
    theVar:int32;
    ?_initialize_ :=
      _initialize_ +
      "mov( " +
      @string:initVal +
      ", " +
      @string:theVar +
      " );";
  #endmacro

var
  i:init_int32( 1 );

begin InitDemo;

```

```

        stdout.put( "i=", i, nl );
    end InitDemo;

```

Note how this example uses string concatenation to append an initialization string to the end of the existing string. Although `_initialize_` and `_finalize_` start out as the empty string, there may be more than one initialization string required by the program. For example, consider the following modification to the code above:

```

var
    i:init_int32( 1 );
    j:init_int32( 2 );

```

The two macro invocations above produce the initialization string "mov(1, i);mov(2,j);". Had the macro not used string concatenation to attach its string to the end of the existing `_initialize_` string and then only the last initialization statement would have been generated.

You can put any number of statements into an initialization string, although the compiler tools used to write HLA limit the length of the string to something less than 32,768 characters. In general, you should try to limit the length of the initialization string to something less than 4,096 characters (this includes all initialization strings concatenated together within a single procedure).

Two very useful purposes for the initialization string include automatic constructor invocation and Unit initialization code invocation. Let's consider the **unit** situation first. Associated with some unit you might have some code that you need to execute to initialize the code when the program first loads in to memory, e.g.,

```

unit NeedsInit;
#include( "NeedsInit.hhf" )
static
    i:uns32;
    j:uns32;

    procedure InitThisUnit;
    begin InitThisUnit;

        mov( 0, i );
        mov( 1, j );

    end InitThisUnit;
    .
    .
    .
end NeedsInit;

```

Now suppose that the *NeedsInit.hhf* header file contains the following lines:

```

procedure InitThisUnit; @external;
?_initialize_ := _initialize_ + "InitThisUnit()";

```

When you include the header file in your main program (that uses this unit), the statement above will insert a call to the `InitThisUnit` procedure into the main program. Therefore, the main program will automatically call the `InitThisUnit` procedure without the user of this unit having to explicitly make this call.

You can use a similar approach to automatically invoke class constructors and destructors in a procedure. Consider the following program that demonstrates how this could work:

```
program InitDemo2;
#include( "stdlib.hhf" )

type
  _MyClass:
    class
      procedure create;
      var
        i: int32;

    endclass;

  #macro MyClass:theObject;
    forward( theObject );
    theObject: _MyClass;
    ?_initialize_ := _initialize_ +
                    @string:theObject +
                    ".create()";
  #endmacro

  procedure _MyClass.create;
  begin create;

    push( eax );
    if( esi = 0 ) then

      mem.alloc( @size( _MyClass ) );
      mov( eax, esi );

    endif;
    mov( &_MyClass._VMT_, this._pVMT_ );
    mov( 12345, this.i );
    pop( eax );

  end create;

  procedure UsesMyClass;
  var
    mc:MyClass;

  begin UsesMyClass;

    stdout.put( "mc.i=", mc.i, nl );

  end UsesMyClass;

static
  vmt( _MyClass );

begin InitDemo2;

  UsesMyClass();
```

```
end InitDemo2;
```

The variable declaration `mc:MyClass` inside the `UsesMyClass` procedure (effectively) expands to the following text:

```
mc: _MyClass;
?_initialize_ := _initialize_ + "mc.create()";
```

Therefore, when the `UsesMyClass` procedure executes, the first thing it does is call the constructor for the `mc/_MyClass` object. Notice that the author of the `UsesMyClass` procedure did not have to explicitly call this routine.

You can use the `_finalize_` string in a similar manner to automatically call any destructors associated with an object.

Note that if an exception occurs and you do not handle the exception within a procedure containing `_finalize_` code, the program will not execute the statements emitted by `_finalize_` (any more than the program will execute any other statements within a procedure that an exception interrupts). If you absolutely, positively, must ensure that the code calls a destructor before leaving a procedure (via an exception), then you might try the following code:

```
?_initialize_ :=
    _initialize_ +
    <<string to call constructor>> +
    "try ";

?_finalize_ :=
    _finalize_ +
    "anyexception push(eax); " +
    <<string to call destructor>> +
    "pop(eax); raise( eax ); endtry; " +
    <<string to call destructor>>;
```

This version slips a **try..endtry** block around the whole procedure. If an exception occurs, the **anyexception** handler traps it and calls the associated destructor, then re-raises the exception so the caller will handle it. If an exception does not occur, then the second call to the destructor above executes to clean up the object before control transfers back to the caller. Note that this is not a perfect solution because it does not prevent the programmer from slipping in their own **try..endtry** statement with an `anyexception` clause that doesn't bother to execute the `_finalize_` code.

13 The HLA Compile-Time Language

13.1 HLA Compile-Time Language, Macros, and Pragmas

This topic section describes one of HLA's more impressive features - the compile time language. Combined with the macro preprocessor, the HLA compile-time language lets you customize the HLA language in almost an infinite variety of ways.

Compile-time programs are just that- programs that execute while HLA is compiling your source file. You embed compile-time language statements directly in your HLA source files and these short program fragments control how HLA compiles your assembly code.

This section doesn't fully explain the HLA compile-time language because you've already seen some major parts of it. For example, **val** constants in the HLA source file are equivalent to compile-time variables. The "=" statement is the compile-time assignment statement. This topic section, therefore, builds on the material that appears elsewhere in HLA Reference Manual.

13.2 Viewing the Output of the HLA Compile-Time Language

The HLA compile-time language can generate assembly language statements during the compilation of an HLA program. Because it isn't always obvious what code the compile-time language is generating, you'll sometimes need the ability to view the output of the HLA compiler. This is easily accomplished using the HLA command-line option "-hla". This command-line option tells HLA to produce an assembly language output file that uses a pseudo-HLA syntax. The result is not compilable under HLA, but it will show you the "pure" assembly language output that HLA produces from your original source file. Here's an example source file:

```
program demoCTLoutput;
var
    array:dword[11];
begin demoCTLoutput;

    #for( i := 0 to 10 )

        mov( i, array[ i*4 ] );

    #endfor;

end demoCTLoutput;
```

Here is the output that the HLA compiler produces for the main program when you supply the "-hla" command-line option:

```
begin _HLAMain;
procedure start;
begin start;
end start;

    call BuildExcepts__hla_;
    pushd( 0 );
    push( ebp );
    push( ebp );
    lea( [esp-4], ebp );
    sub( 44, esp );
    and( -4, esp );

    mov( 0, (type dword [ebp-48]) );
    mov( 1, (type dword [ebp-44]) );
    mov( 2, (type dword [ebp-40]) );
```

```

    mov( 3, (type dword [ebp-36]) );
    mov( 4, (type dword [ebp-32]) );
    mov( 5, (type dword [ebp-28]) );
    mov( 6, (type dword [ebp-24]) );
    mov( 7, (type dword [ebp-20]) );
    mov( 8, (type dword [ebp-16]) );
    mov( 9, (type dword [ebp-12]) );
    mov( 10, (type dword [ebp-8]) );
QuitMain__hla_::
    pushd( 0 );
    call( (type dword __imp__ExitProcess@4) );
end _HLAMain;

```

13.3 #linker Directive

The **#linker** directive passes a single string argument along to the linker. This is typically done to specify the name of some object or library file to link with the current file during the link edit phase. This directive has the following syntax:

```
#linker( "linker directive or file" )
```

For example, under Linux the following **#linker** commands tell HLA to have the linker link in part of the C Standard Library:

```

#if( os.linux )
    #linker( "-I /lib/ld-linux.so.2" )
    #linker( "-lc" )
#endif

```

As you can see from this example, each **#linker** string argument is really just a command-line argument that is passed along to the GNU ld linker (which is exactly how the **#linker** command operates under any OS). If multiple **#linker** commands appear in a source file (as is the case in this example), HLA concatenates the command-line arguments together (with a space separating them) prior to passing the command-line arguments to the linker.

13.4 The #Include Directive

Like most languages, HLA provides a source inclusion directive that inserts some other file into the middle of a source file during compilation. HLA's **#include** directive is very similar to the pragma of the same name in C/C++ and you primarily use them both for the same purpose: including library header files into your programs.

HLA's include directive has the following syntax:

```
#include( string_expression )
```

Note that any arbitrary compile-time string expression is legal. You are not limited to a literal string constant.

The **#include** directive is legal anywhere whitespace is legal. The string specifies a filename that HLA will insert into the program during compilation at the point the **#include** appears. If HLA cannot find the file specified by the string constant in the current directory (or in the directory specified if the string contains a pathname), then HLA tries to find the file in the location specified by the "hlainc" environment variable. If HLA still doesn't find the file, HLA will report an error.

Although you can use the **#include** directive to insert any arbitrary text at an arbitrary point in your program, the vast majority of the time you will use **#include** to include a library header file (either an HLA Standard Library header file or a library header file you've written) into your program. HLA requires that you compile all external files at lex level zero. Therefore, if you are

including some declarations into your program, the **#include** directive should be just inside the main program. Convention dictates that **#include** directives that include library headers should appear immediately after the **program** or **unit** header in a file.

13.5 The #IncludeOnce Directive

When composing complex header files, particularly when constructing library header files, you may find it necessary to insert a **#include("file")** directive into some other header files. Generally, this is not a problem, HLA certainly allows nested include files (up to 256 files deep). However, unless you are very careful about how you organize your files, it is very easy to create an "include loop" where one header file includes another and that other header file includes the first. Attempting to compile a program that includes either header file results in an infinite "include loop" during compilation; clearly, this is not desirable.

The standard way to handle this situation is to surround all the statements in an include file with a **#if** statement as follows:

```
#if( !@defined( headerfilename_hhf ))

?headerfilename_hhf := true;

    << Statements associated with this header file go here >>

#endif
```

The first time HLA includes this file the symbol "headerfilename_hhf" is not defined, so HLA processes the statements in the body of the **#if** statement. The very first statement defines this "headerfilename_hhf" symbol (the value and type of this symbol are irrelevant for our purposes; only the fact that the symbol exists is important). Thereafter, if some other header file includes this file a second (or additional) time, the "headerfilename_hhf" symbol is defined, so HLA skips all the statements in the header file since the value of the boolean expression in the **#if** statement is false. Therefore, HLA only processes the statements of this header file (at least those inside the **#if** statement) the first time it encounters this particular header file.

A drawback to this scheme is that HLA must still open the header file and read every line from the file, even if it ignores all the lines in the file. For large header files, (e.g., the "stdlib.hhf" header file) this can consume a significant amount of time during compilation. The **#includeonce** directive provides a solution for this problem.

You use the **#includeonce** directive just like the **#include** directive. The only difference between the two is that HLA keeps track of all files it has processed using the **#include** or **#includeonce** directives and will not process a header file a second time if you attempt to include it using the **#includeonce** directive.

Whenever HLA processes the **#includeonce** directive, it first compares its string operand with a list of strings appearing in previous **#include** or **#includeonce** directives. If it matches one of these previous strings, then HLA ignores the **#includeonce** directive; if the include filename does not appear in HLA's internal list, then HLA adds this filename to the list and includes the file.

Note that HLA's **#includeonce** directive only compares strings for equality. If you use two separate filenames for the same file, HLA will not detect this and it will include the file a second time. E.g., if the current directory is "C:\hlafiles" then the following sequence will include the file "whoops.hhf" twice:

```
#IncludeOnce( "whoops.hhf" )
#includeOnce( "c:\whoops.hhf" )
```

Also note that the **#include** directive will include its file regardless of whether the program previously included that file with a **#includeonce** directive, e.g., the following sequence also includes "whoops.hhf" twice:

```
#IncludeOnce( "whoops.hhf" )
#include( "whoops.hhf" )
```

For these two reasons, it's still a good idea to protect all header files using the `#if` technique mentioned earlier, even if you use the `#includeonce` directive throughout.

13.6 Macros

HLA has one of the most powerful macro expansion facilities of any programming language. HLA's macros are the key to extending the HLA language. The following subsections describe HLA's powerful macro processing facilities.

13.6.1 Standard Macros

HLA provides powerful macro capabilities. You can declare macros almost anywhere whitespace is allowed in a program using the following syntax:

```
#macro identifier ( optional_parameter_list ) ;
    statements
#endmacro
```

Note that a semicolon does not follow the `#endmacro` clause.

Example:

```
#macro MyMacro;
    ?i = i + 1;
#endmacro
```

The optional parameter list must be a list of one or more identifiers separated by commas. Unlike procedure declarations, you do not associate a type with macro parameters. HLA automatically associates the type "text" with macro parameters (except for two special cases noted below). Example:

```
#macro MacroWParms( a, b, c );
    ?a = b + c;
#endmacro
```

Optionally, the last (or only) name in the identifier list may take the form `identifier[]`. This syntax tells the macro that it may allow a variable number of parameters and HLA will create an array of string objects to hold all the parameters (HLA uses a string array rather than a text array because text arrays are illegal). Example:

```
#macro MacroWVarParms( a, b, c[] );
    ?a = b + @text( c[0] ) + c[1] );
#endmacro
```

If the macro does not allow any parameters, then you follow the identifier with a semicolon (i.e., no parentheses or parameter identifiers). See the first example in this section for a macro without any parameters.

When using the array form (variable parameters) in a macro argument list, HLA will parse the remaining actual parameters and shove them into the array, one (perceived) parameter per string array element. Sometimes, however, you might want to handle the parameter parsing chores yourself (for example, to allow commas as part of an actual macro parameter) rather than have HLA handle this task for you. HLA provides an option to tell it to grab all remaining (or simply all) parameter text passed in the actual parameter list and stores all this data into a compile-time string object. To achieve this, you prefix the last (or only) formal macro parameter with the reserved word **string**, e.g.,

```
#macro MacroWStringParms( a, b, string c );
```



```
<<macro body>>
#endmacro
```

In this example, the first two actual parameters will be assigned to the text objects *a* and *b* within the macro. Any remaining parameters will be collected as a single string and stored into the *c* formal parameter as a string.

One very useful purpose for string macro parameters is to allow you to grab a list of parameters you want to pass on to some other macro or procedure as a single object. E.g.,

```
procedure abc( a:byte; b:word; c:dword );
begin abc;
.
.
.
end abc;

#macro CallsAbc( string abcParms );
.
.
.
abc( @text( abcParms ) );
.
.
.
#endmacro
.
.
.
CallsAbc( 1, 2, 3 );
```

The final macro invocation in this sequence passes the three parameters "1,2,3" to the *abc* function.

Occasionally you may need to define some symbols that are local to a particular macro invocation (that is, each invocation of the macro generates a unique symbol for a given identifier). The local identifier list allows you to do this. To declare a list of local identifiers, simply following the parameter list (after the parenthesis but before the semicolon) with a colon (":") and a comma separated list of identifiers, e.g.,

```
#macro ThisMacro( parm1 ):id1,id2;
...
```

HLA automatically renames each symbol appearing in the local identifier list so that the new name is unique throughout the program. HLA creates unique symbols using some form such as XXXX_HLA where XXXX is some hexadecimal numeric value. To guarantee that HLA can generate unique symbols, you should avoid defining symbols of this form in your own programs (in general, symbols that begin and end with an underscore are reserved for use by the compiler and the HLA standard library). Example:

```
#macro LocalSym : i,j;

j: cmp(ax, 0)
   jne( i )
   dec( ax )
   jmp( j )
i:
#endmacro
```

Without the local identifier list, multiple expansions of this macro within the same procedure would yield multiple statement definitions for *i* and *j*. With the local statement present, however, HLA substitutes symbols similar to `_0001_HLA_` and `_0002_HLA_` for *i* and *j* for the first invocation and symbols like `_0003_HLA_` and `_0004_HLA_` for *i* and *j* on the second invocation, etc. This avoids duplicate symbol errors if you do not use (poorly chosen) identifiers like `_0001_HLA_` and `_0004_HLA_` in your code.

The statements section of the macro may contain any legal HLA statements (including definitions of other macros). However, the legality of such statements is controlled by where you expand the macro.

To invoke a macro, you simply supply its name and an appropriate set of parameters. Unless you specify a variable number of parameters (using the array syntax) then the number of actual parameters must exactly match the number of formal parameters. If you specify a variable number of parameters, then the number of actual parameters must be greater than or equal to the number of formal parameters (not counting the array parameter).

During macro expansion, HLA automatically substitutes the text associated with an actual parameter for the formal parameter in the macro's body. The array parameter, however, is a string array rather than a text array so you will have to force the expansion yourself using the `@text` function:

```
#macro example( variableParms[] );
    ?@text(variableParms[0]) := @text(variableParms[1]);
#endmacro
```

Actual macro parameters consist of a string of characters up to, but not including a separate comma or the closing parentheses, e.g.,

```
example( v1, x+2*y )
```

"v1" is the text for parameter #1, "x+2*y" is the text for parameter #2. Note that HLA strips all leading whitespace and control characters before and after the actual parameter when expanding the code in-line. The example immediately above would expand to the following:

```
?v1 := x+2*y;
```

If (balanced) parentheses appear in some macro's actual parameter list, HLA does not count the closing parenthesis as the end of the macro parameter. That is, the following is legal:

```
example( v1, ((x+2)*y) )
```

This expands to:

```
?v1 := ((x+2)*y);
```

If you need to embed commas or unmatched parentheses in the text of an actual parameter, use the HLA literal quotes `#(` and `)#` to surround the text. Everything (except surrounding whitespace) inside the literal quotes will be included as part of the macro parameter's text. Example:

```
example( v1, #( array[0,1,i] )# )
```

The above expands to:

```
?v1 := array[0,1,i];
```

Without the literal quote operator, HLA would have expanded the code to

```
?V1 := array[0;
```

and then generated an error because (1) there were too many actual macro parameters (four instead of two) and (2) the expansion produces a syntax error.

Of course, HLA's macro parameter parser does not consider commas appearing inside string or character constants as parameter separators. The following is legal, as you would expect:

```
example( charVar, ', ' )
```

As you may have noticed in these examples, a macro invocation does not require a terminating semicolon. Macro expansion occurs upon encountering the closing parenthesis of the macro invocation. HLA uses this syntax to allow a macro expansion *anywhere* in an HLA source file. Consider the following:

```
#macro funny( dest )
    , dest );
#endmacro

mov( 0 funny( ax )
```

This code expands to "mov(0, ax);" and produces a legal machine instruction. Of course, this is a truly horrible example of macro use (the style is really bad), but it demonstrates the power of HLA macros in your program. This "expand anywhere" philosophy is the primary reason macro invocations do not end with a semicolon.

13.6.2 Where You Declare a Macro Affects its Visibility

You may declare a macro almost anywhere whitespace is allowed in a program. This increases the utility of macros as part of the HLA Compile-time Language. However, there are some issues of which you should be aware when declare macros at arbitrary points; this section will discuss those issues so you can avoid some pitfalls of this new flexibility.

First, unless you have good reason to do otherwise, you really should declare your macros in a declaration section of your program. Long-time HLA programmers have grown used to finding them there and, by placing your macros in a declaration section (e.g., wherever a procedure declaration is allowed), you'll make your programs easier to read because other programmers can look for such declarations in a few known locations. Arbitrarily scattering your macro declarations all over the place can make your programs harder to read. In addition, it should be understood that you must declare a macro before the first invocation.

Like other identifiers in an HLA program, macro identifiers have a scope that limits their visibility. If you declare a macro within a procedure, then that macro's identifier is only visible within that procedure and you cannot invoke (call) the macro outside of the procedure (that is, beyond the `end` statement associated with the procedure). Note that this is true even if you declare the macro in the body of the procedure, outside the procedure's declaration section, e.g.,

```
procedure SomeProc;
begin SomeProc;

    #macro mov0eax;
        mov( 0, eax )
    #endmacro

    mov0eax; // legal here

end SomeProc;

mov0eax; // undefined symbol here.
```

If you declare a macro in a namespace or within an HLA class, you may invoke that macro from outside the namespace or class declaration by prefixing the macro identifier with the

namespace or class or object identifier using the normal dot-notation for access to fields of the namespace or class. Note that you may invoke namespace or class macros within the namespace or class without the namespace prefix (just as you may access other symbol types within the namespace or class without the prefix).

You may also embed macro definitions within records and unions. However, when you do this HLA will insert the macro's symbol into the field list for the record or union. Because HLA does not provide a way to access anything other than variable fields of a record or union outside the declaration of that type, you will not be able to invoke the macro from outside the record or union declaration. However, you may invoke that macro within the same record/union declaration that contains the macro definition, e.g.,

```
type
  r :record

      i:int32;
      #macro inrec;
          k:int32;
      #endmacro
      j:int32;
      inrec; // Legal expansion here
  endrecord;

var
  r.inrec; // this is not legal here. Use a namespace or class to do
  this.
```

Because of some limitations of the HLA implementation language (Flex/Bison), there is an important peculiarity you should know when declaring macros. In particular, HLA may process a macro declaration before it finishes processing whatever occurs immediately before the macro. Therefore, if the successful definition of a macro depends on whatever appears immediately before the macro, the declaration may fail. Though this is rare, it does occur occasionally. Should this happen to you, try an insert an innocuous syntactical item (like a semicolon) before the macro declaration.

13.6.3 Multi-part (Context Free) Macro Invocations:

HLA macros provide some very powerful facilities not found in other macro assemblers. One of the unique features that HLA macros provide is support for multi-part (or context-free) macro invocations. This feature is accessed via the `#terminator` and `#keyword` reserved words. Consider the following macro declaration:

```
program demoTerminator;

#include( "stdio.hhf" );

#macro InfLoop:TopOfLoop, LoopExit;
  TopOfLoop:
#terminator endInfLoop;
  jmp TopOfLoop;
  LoopExit:
#endmacro;

static
  i:int32;

begin demoTerminator;

  mov( 0, i );
  InfLoop
```

```

        stdout.put( "i=", i, nl );
        inc( i );

    endInfLoop;

end demoTerminator;

```

The `#terminator` keyword, if it appears within a macro, defines a second macro that is available for a one-time use after invoking the main macro. In the example above, the `endInfLoop` macro is available only after the invocation of the `InfLoop` macro. Once you invoke the `EndInfLoop` macro, it is no longer available (though the macro calls can be nested, more on that later). During the invocation of the `#terminator` macro, all local symbols declared in the main macro (`InfLoop` above) are available (note that these symbols are not available outside the macro body. In particular, you could refer to neither `TopOfLoop` nor `LoopExit` in the statements appearing between the `InfLoop` and `endInfLoop` invocations above). The code above, by the way, emits code similar to the following:

```

_0000_HLA_:
    stdout.put( "i=", i, nl );
    inc( i );
    jmp _0000_HLA_;
_0001_HLA_:

```

The macro expansion code appears in italics. This program, therefore, generates an infinite loop that prints successive integer values.

These macros are called multi-part macros for the obvious reason: they come in multiple pieces (note, though, that HLA only allows a single `#terminator` macro). They are also referred to as *Context-Free macros* because of their syntactical nature. Earlier, this document claimed that you could refer to the `#terminator` macro only once after invoking the main macro. Technically, this should have said "you can invoke the terminator once for each outstanding invocation of the main macro." In other words, you can nest these macro calls, e.g.,

```

    InfLoop

        mov( 0, j );
        InfLoop

            inc( i );
            inc( j );
            stdout.put( "i=", i, " j=", j, nl );

        endInfLoop;

    endInfLoop;

```

The term *Context-Free* comes from automata theory; it describes this nestable feature of these macros.

As should be painfully obvious from this `InfLoop` macro example, it would be nice if one could define more than one macro within this context-free group. Furthermore, the need often arises to define limited-scope macros that can be invoked more than once (limited-scope means between the main macro call and the terminator macro invocation). The `#keyword` definition allows you to create such macros.

In the `InfLoop` example above, it would be nice if you could exit the loop using a statement like `brkLoop` (note that **break** is an HLA reserved word and cannot be used for this purpose). The `#keyword` section of a macro allows you to do exactly this. Consider the following macro definition:

```
#macro InfLoop:TopOfLoop, LoopExit;
  TopOfLoop:
#keyword brkLoop;
  jmp LoopExit;
#terminator endInfLoop;
  jmp TopOfLoop;
  LoopExit:
#endmacro;
```

As with the `#terminator` section, the `#keyword` section defines a macro that is active after the main macro invocation until the terminator macro invocation. However, `#keyword` macro invocations do not terminate the multi-part invocation. Furthermore, `#keyword` invocations may occur more than once. Consider the following code that might appear in the main program:

```
mov( 0, i );
InfLoop

  mov( 0, j );
  InfLoop

    inc( j );
    stdout.put( "i=", i, " j=", j, nl );
    if( j >= 10 ) then

      brkLoop;

    endif

  endInfLoop;
  inc( i );
  if( i >= 10 ) then

    brkLoop;

  endif;

endInfLoop;
```

The `brkLoop` invocation inside the "if(j >= 10)" statement will break out of the inner-most loop, as expected (another feature of the context-free behavior of HLA's macros). The `brkLoop` invocation associated with the "if(i >= 10)" statement breaks out of the outer-most loop. Of course, the HLA language provides the **forever..endfor** loop and the **break** and **breakif** statements, so there is no need for this `InfLoop` macro, nevertheless, this example is useful because it is easy to understand. If you are looking for a challenge, try creating a statement similar to the C/C++ switch/case statement; it is perfectly possible to implement such a statement with HLA's macro facilities, see the HLA Standard Library for an example of the switch statement implemented as a macro.

The discussion above introduced the `#keyword` and `#terminator` macro sections in an informal way. There are a few details omitted from that discussion. First, the full syntax for HLA macro declarations is actually:

```
#macro identifier ( optional_parameter_list ) :optional_local_symbols;
  statements

#keyword identifier ( optional_parameter_list ) :optional_local_symbols;
  statements
```

note: additional `#keyword` declarations may appear here

```
#terminator identifier ( optional_parameter_list )
:optional_local_symbols;
  statements
#endmacro
```

There are three things that should immediately stand out here: (1) you may define more than one `#keyword` within a macro. (2) `#keywords` and `#terminators` allow optional parameters. (3) `#keywords` and `#terminators` allow their own local symbols.

The scope of the parameters and local symbols isn't particularly intuitive (although it turns out that the scope rules are exactly what you would want). The parameters and local symbols declared in the main macro declaration are available to all statements in the macro (including the statements in the `#keyword` and `#terminator` sections). The `InfLoop` macro used this feature since the `JMP` instructions in the `brkLoop` and `endInfLoop` sections referred to the local symbols declared in the main macro. The `InfLoop` macro did not declare any parameters, but had they been present, the `brkLoop` and `endInfLoop` sections could have used those parameters as well.

Parameters and local symbols declared in a `#keyword` or `#terminator` section are local to that particular section. In particular, parameters and/or local symbols declared in a `#keyword` section are not visible in other `#keyword` sections or in the `#terminator` section.

One important issue is that local symbols in a multipart macro are visible in the main code between the start of the multipart macro and the terminating macro. That is, if you have some sequence like the following:

```
InfLoop
    jmp LoopExit;

endInfLoop;
```

The HLA substitutes the appropriate internal symbol (e.g., `_xxxx_HLA_`) for the `LoopExit` symbol. This is somewhat unintuitive and might be considered a flaw in HLA's design. Future versions of HLA may deal with this issue; in the meantime, however, some code takes advantage of this feature (to mask global symbols) so it's not easy to change without breaking a lot of code. Be forewarned before taking advantage of this "feature", however, that it will probably change in HLA v3.x. An important aspect of this behavior is that macro parameter names are also visible in the code section between the initial macro and the terminator macro. Therefore, you must take care to choose macro parameter names that will not conflict with other identifiers in your program. E.g., the following will probably lead to some problems:

```
static
  i:int32;

#macro parmi(i);
  mov( i, eax );
#terminator endParmi;
  mov( eax, i );
#endmacro
.
.
.
  parmi( xyz );
  mov( i, ebx );// actually moves xyz into ebx, since the parameter i
                // overrides the global variable i here.
endParmi;
```

13.6.4 Macro Invocations and Macro Parameters:

As mentioned earlier, HLA treats all non-array macro parameters as text constants that are assigned a string corresponding to the actual parameter(s) passed to the macro. I.e., consider the following:

```
#macro SetI( v );
    ?i := v;
#endmacro

SetI( 2 );
```

The above macro and invocation is roughly equivalent to the following:

```
const
    v : text := "2";
    ?i := v;
```

When utilizing variable parameter lists in a macro, HLA treats the parameter object as a string array rather than a text array (because HLA does not support text arrays). For example, consider the following macro and invocation:

```
#macro SetI2( v[] );
    ?i := v[0];
#endmacro

SetI2( 2 );
```

Although this looks quite similar to the previous example, there is a subtle difference between the two. The former example will initialize the constant (value) `i` with the `int32` value 2. The second example will initialize `i` with the string value "2".

If you need to treat a macro array parameter as text rather than as a string object, use the HLA `@text` function that expands a string parameter as text. E.g., the former example could be rewritten as:

```
#macro SetI2( v[] );
    ?i := @text( v[0] );
#endmacro

SetI2( 2 );
```

In this example, the `@text` function tells HLA to expand the string value `v[0]` (which is "2") directly as text, so the "SetI2(2)" invocation expands as

```
?i := 2;
rather than as
?i := "2";
```

On occasion, you may need to do the converse of this operation. That is, you may want to treat a standard (non-array) macro parameter as a string object rather than as a text object. You can accomplish this by using the `@string(text_object)` function. When HLA encounters this construct, it will substitute a string constant for the identifier. The following example demonstrates one possible use of this feature:

```
program demoString;
```



```

#macro seti3( v );
    #print( "i is being set to " + @string( v ) )
    ?i := v;
#endmacro

begin demoString;

    seti3( 4 )
    #print( "i = " + string( i ) )
    seti3( 2 )
    #print( "i = " + string( i ) )

end demoString;

```

Note that HLA supports a second, deprecated, form: **@string:identifier**. Though you might see this form in older source code, you should not use this form in HLA v2.x programs as this feature will probably be eliminated from a future version of HLA.

If an identifier is a **text** constant (e.g., a macro parameter or a const/value identifier of type **text**), special care must be taken to modify the string associated with that text object. A simple **val** expression like the following won't work:

```
?textVar:text := "SomeNewText";
```

The reason this doesn't work is subtle: if `textVar` is already a text object, HLA immediately replaces `textVar` with its corresponding string; this includes the occurrence of the identifier immediately after the "?" in the example above. So were you to execute the following two statements:

```
?textVar:text := "x";
?textVar:text := "1";
```

the second statement would not change `textVar`'s value from "x" to "1". Instead, the second statement above would be converted to:

```
?x:text := "1";
```

and `textVar`'s value would remain "x". To overcome this problem, HLA provides a special syntactical entity that converts a text object to a string and then returns the text object ID. The syntax for this special form is **@tostring:identifier**. The example above could be rewritten as:

```
?textVar:text := "x";
?@tostring:textVar:text := "1";
```

In this example, *textVar* would be a text object that expands to the string "1".

13.6.5 Processing Macro Parameters

As described earlier, HLA processes as parameters all text between a set of matching parentheses after the macro's name in a macro invocation. HLA macro parameters are delimited by the surrounding parentheses and commas. That is, the first parameter consists of all text beyond the left parenthesis up to the first comma (or up to the right parenthesis if there is only one parameter). The second parameter consists of all text just beyond the first comma up to the second comma (or right parenthesis if there are only two parameters); and so on. The last parameter consists of all text from the last comma to the closing right parenthesis. Within a single parameter, any text appearing between "(" and ")", "[" and "]", or "{" and "}" will be considered as a single parameter, even if there are commas present. Therefore, procedure calls (with parameters), array element accesses, character set constants, macro invocations, and other such items will constitute a single macro parameter. The follow macro invocation, for example, has three arguments because the *parmMacro* invocation counts as a single parameter:

```
ThreeParms( a, parmMacro( 0, 1 ), b );
```

This example demonstrates another feature of HLA's macro processing system - HLA uses *deferred macro parameter expansion*. That is, the text of a macro parameter is expanded when HLA encounters the formal parameter within the macro's body, *not* while HLA is processing the actual parameters in the macro invocation (which would be *eager* evaluation).

There are three exceptions to the rule of deferred parameter evaluation: (1) text constants are always expanded in an eager fashion (that is, the value of the text constant, not the text constant's name, is passed as the macro parameter). (2) The **@text** function, if it appears in a parameter list, expands the string parameter in an eager fashion. (3) The **@eval** function immediately evaluates its parameter; the discussion of **@eval** appears a little later.

In general, there is very little difference between eager and deferred evaluation of macro parameters. In some rare cases, there is a semantic difference between the two. For example, consider the following two programs:

```
program demoDeferred;
#macro two( x, y ):z;
  ?z:text := "1";
  x+y
#endmacro

const
  z:string := "2";

begin demoDeferred;

  ?i := two( z, 2 );
  #print( "i=" + string( i ) )

end demoDeferred;
```

In the example above, the code passes the actual parameter "z" as the value for the formal parameter "x". Therefore, whenever HLA expands "x" it gets the value "z", which is a local symbol inside the "two" macro, that expands to the value "1". Therefore, this code prints "3" ("1" plus y's value which is "2") during assembly. Now consider the following code:

```
program demoEager;
#macro two( x, y ):z;
  ?z:text := "1";
  x+y
#endmacro

const
  z:string := "2";

begin demoEager;

  ?i := two( @text( z ), 2 );
  #print( "i=" + string( i ) )

end demoEager;
```

The only differences between these two programs are their names and the fact that `demoEager` invocation of "two" uses the **@text** function to eagerly expand z's text. As a result, the formal parameter "x" is given the value of z's expansion ("2") and HLA ignores the local value for "z" in macro "two". This code prints the value "4" during assembly. Note that changing "z" in the main program to a text constant (rather than a string constant) has the same effect:

```

program demoEager;
#macro two( x, y ):z;
    ?z:text:="1";
    x+y
#endmacro

const
    z:text := "2";

begin demoEager;

    ?i := two( z, 2 );
    #print( "i=" + string( i ) )

end demoEager;

```

This program also prints "4" during assembly.

One place where deferred vs. eager evaluation can get you into trouble is with some of the HLA built-in functions. Consider the following HLA macro:

```

#macro DemoProblem( Parm );

    #print( string( Parm ) )

#endmacro
.
.
.
DemoProblem( @linenumber );

```

(The **@linenumber** function returns, as an `uns32` constant, the current line number in the file).

When this program fragment compiles, HLA will use deferred evaluation and pass the text **@linenumber** as the parameter `Parm`. Upon compilation of this fragment, the macro will expand to `#print(string(@linenumber))` with the intent, apparently, being to print the line number of the statement containing the *DemoProblem* invocation. In reality, that is not what this code will do. Instead, it will print the line number, in the macro, of the `#print(string(Parm))` statement. By delaying the substitution of the current line number for the **@linenumber** function call until inside the macro, deferred execution produces the wrong result. What is really needed here is eager evaluation so that the **@linenumber** function expands to the line number string before being passed as a parameter to the *DemoProblem* macro. The **@eval** built-in function provides this capability. The following coding of the *DemoProblem* macro invocation will solve the problem:

```
DemoProblem( @eval( @linenumber ) );
```

Now the compiler will execute the **@linenumber** function and pass that number as the macro parameter text rather than the string `"@linenumber"`. Therefore, the **#print** statement inside the macro will print the actual line number of the *DemoProblem* statement rather than the line number of the `#print` statement.

Keep these minor differences in mind if you run into trouble using macro parameters.

13.7 Built-in Functions:

HLA provides several built-in functions that take constant operands and produce constant results. It is important that you differentiate these compile-time functions from run-time functions. These functions do not emit any object code, and therefore do not exist while your program is running. They are only available while HLA is compiling your program. Note that many of these

functions are trivial to implement in assembly language or have counterparts in the HLA standard library. Therefore, the fact that they are not available at run-time shouldn't prove to be much of a problem.

13.8 Constant Type Conversion Functions

`boolean(const_expr)`

The expression must be an ordinal or string expression. If `const_expr` is numeric, this function returns **false** for zero and **true** for everything else. If `const_expr` is a character, this function returns true for "T" and false for "F". It generates an error for any other character value. If `const_expr` is a string, the string must contain "true" or "false" else HLA generates an error.

`int8(const_expr)`
`int16(const_expr)`
`int32(const_expr)`
`int64(const_expr)`
`int128(const_expr)`
`uns8(const_expr)`
`uns16(const_expr)`
`uns32(const_expr)`
`uns64(const_expr)`
`uns128(const_expr)`
`byte(const_expr)`
`word(const_expr)`
`dword(const_expr)`
`qword(const_expr)`
`lword(const_expr)`

These functions convert their parameter to the specified integer. For real operands, the result is truncated to form a numeric operand. For all other numeric operands, the result is range checked. For character operands, the ASCII code of the specified character is returned. For boolean objects, zero or one is returned. For string operands, the string must be a sequence of decimal characters that are converted to the specified type. Note that **byte**, **word**, and **dword**, ..., types are synonymous with **uns8**, **uns16**, and **uns32**, ..., for the purposes of range checking.

`real32(const_expr)`
`real64(const_expr)`
`real80(const_expr)`

These functions are similar to the earlier integer functions, except these functions produce the obvious real results. Only numeric and string parameters are legal.

`char(const_expr)`

`const_expr` must be an ordinal or string value. This function returns a character whose ASCII code is that ordinal value. For strings, this function returns the first character of the string.

`string(const_expr)`

This function produces a reasonable string representation of the parameter. Almost all data types are legal.

`cset(const_expr)`

The parameter must be a character, string, or character set. For character parameters, this function returns the singleton set containing only the specified character. For strings, each character in the string is unioned into the set and the function returns the result. If the parameter is a character set, this function makes a copy of that character set.

13.8.1 Bitwise Type Transfer Functions

The type conversion functions of the previous section will automatically convert their operands from the source type to the destination type. Sometimes you might want to change the type of some object without changing its value. For many "conversions" this is exactly what takes place. For example, when converting an `uns8` object to an `uns16` value using the `uns16(---)` function, HLA does not modify the bit pattern at all. For other conversions, however, HLA may completely change the underlying bit pattern when doing the conversion. For example, when converting the `real32` value 1.0 to a `dword` value, HLA completely changes the underlying bit pattern (`$3F80_0000`) so that the `dword` value is equal to one. On occasion, however, you might actually want to copy the bits straight across so that the resulting `dword` value is `$3F80_0000`. The HLA bit-transfer type conversion compile-time functions provide this facility.

The HLA bit-transfer type conversion functions are the following:

```
@int8( const_expr )
@int16( const_expr )
@int32( const_expr )
@int64( const_expr )
@int128( const_expr )
@uns8( const_expr )
@uns16 const_expr )
@uns32( const_expr )
@uns64( const_expr )
@uns128( const_expr )
@byte( const_expr )
@word( const_expr )
@dword( const_expr )
@qword( const_expr )
@lword( const_expr )
@real32( const_expr )
@real64( const_expr )
@real80( const_expr )
@char( const_expr )
@cset( const_expr )
```

The above functions extract eight, 16, 32, 64, or 128 bits from the constant expression for use as the value of the function. Note that supplying a string expression as an argument isn't particularly useful since the functions above will simply return the address of the string data in memory while HLA is compiling the program. The `@byte` function provides an additional syntax with two parameters; see the next section for details.

```
@string( const_expr )
```

HLA string objects are pointers (in both the language as well as within the compiler). So simply copying the bits to the internal string object would create problems since the bit pattern probably is not a valid pointer to string data during the compilation. With just a few exceptions, what the `@string` function does is takes the bit data of its argument and translates this to a string (up to 16 characters long). Note that the actual string may be between zero and 16 characters long since the HLA compiler (internally) uses zero-terminated strings to represent string constants. Note that the first zero byte found in the argument will end the string.

If you supply a string expression as an argument to `@string`, HLA simply returns the value of the string argument as the value for the `@string` function. If you supply a text object as an argument to the `@string` function, HLA returns the text data as a string without first expanding the text value. If you supply a pointer constant as an argument to the `@string` function, HLA returns the string that HLA will substitute for the static object when it emits the assembly file.

13.8.2 General functions

```
@abs( numeric_expr )
```

Returns the absolute equivalent of the numeric value passed as a parameter.

```
@byte( integer_expr, which )
```

The *which* parameter is a value in the range 0..15. This function extracts the specified byte from the value of the *integer_expression* parameter. (This is an extension of the **@byte** type transfer function.)

```
@byte( real32_expr, which )
```

The *which* parameter is a value in the range 0..3. This function extracts the specified byte from the value of the *real32_expression* parameter.

```
@byte( real64_expr, which )
```

The *which* parameter is a value in the range 0..7. This function extracts the specified byte from the value of the *real64_expression* parameter.

```
@byte( real80_expr, which )
```

The *which* parameter is a value in the range 0..9. This function extracts the specified byte from the value of the *real80_expression* parameter.

```
@ceil( real_expr )
```

This function returns the smallest integer value larger than or equal to the expression passed as a parameter. Note that although the result will be an integer, this function returns a **real80** value.

```
@cos( real_expr )
```

The *real* parameter is an angle in radians. This function returns the cosine of that angle.

```
@date
```

This function returns a string of the form "YYYY/MM/DD" containing the current date.

```
@env( string_expr )
```

This function returns a string containing the value of a system environment variable (whose name you pass as the *string* parameter). If the specified environment variable does not exist, this function returns the empty string.

```
@exp( real_expr )
```

This function returns a **real80** value that is the result of the computation e^{**real_expr} (i.e., e raised to the specified power).

```
@extract( cset_expr )
```

This function returns a character from the specified character set constant. Note that this function doesn't actually remove the character from the set, if you want to do that, then you will need to explicitly remove the character yourself. The following code demonstrates how to do this:

```
program extractDemo;

val
    c:cset := {'a'..'z'};

begin extractDemo;

    #while( c <> {} )

        ?b := @extract( c );
        #print( "b=" + b )
        ?c := c - {b};
```

```
#endwhile
```

```
end extractDemo;
```

```
@floor( real_expr )
```

This function returns the largest integer value less than or equal to the supplied expression. Note that the returned result is of type **real80** even though it has no fractional component.

```
@isalpha( char_expr )
```

This function returns **true** if the character expression is an upper or lower case alphabetic character.

```
@isalphanum( char_expr )
```

This function returns **true** if the parameter is an alphabetic or numeric character. It returns false otherwise.

```
@isdigit( char_expr )
```

This function returns **true** if the character expression is a decimal digit.

```
@islower( char_expr )
```

This function returns **true** if the character expression is a lower case alphabetic character.

```
@isspace( char_expr )
```

This function returns **true** if the character expression is a "whitespace" character. Typically, this would be spaces, tabs, newlines, returns, linefeeds, etc.

```
@isupper( char_expr )
```

This function returns **true** if the character expression is an upper case alphabetic character.

```
@isxdigit( char_expr )
```

This function returns **true** if the supplied character expression is a hexadecimal digit.

```
@log( real_expr )
```

This function returns the natural (base e) logarithm of the supplied parameter.

```
@log10( real_expr )
```

This function returns the base-10 logarithm of the supplied parameter.

```
@max( comma_separated_list_of_ordinal_or_real_values )
```

This function returns the largest value from the specified list.

```
@min( comma_separated_list_of_ordinal_or_real_values )
```

This function returns the least of the values in the specified list.

```
@odd( int_expr )
```

This function returns **true** if the integer expression is an odd number.

```
@random( int_expr )
```

This function returns a random **uns32** value.

```
@randomize( int_expr )
```

This function uses the integer expression passed as a parameter as the new seed value for the random number generator.

```
@sin( real_expr )
```

This function returns the sine of the angle (in radians) passed as a parameter.

```
@sort(array_expr, int_expr, left_compare_id, right_compare_id, str_expr)
```

This function returns an array containing the elements of *array_expr* sorted in ascending order. The second parameter (*int_expr*) specifies the number of elements in the array to sort (sorting always begins with element zero and continues for *int_expr* elements). Note that **@sort** always returns an array that is the same size as *array_expr*, but only the first *int_expr* elements are sorted.

Because *array_expr* elements can be an arbitrary type, you must supply a mechanism for comparing individual elements of the array. This is accomplished using the last three parameters to **@sort**. First, you must supply the names of two HLA **val** objects as the *left_compare_id* and *right_compare_id* parameters. These two value objects must be the same type as an element of *array_expr*. The last parameter must be a string constant holding the name of a macro that will compare the values in these two identifiers and return true if *left_compare_id* is less than *right_compare_id* (This has to be a string constant so that HLA won't attempt to immediately expand the macro when encountering the name).

Though it shouldn't matter much, the current implementation of **@sort** uses a quick-sort algorithm. There is no guarantee that this function will continue to use quicksort in the future, however.

Here's a quick example:

```
#macro abcmp;
    (a < b)
#endmacro

val
    a:int32;
    b:int32;
    array:int32[8] := [8,7,6,5,4,3,2,1];
    sortedArray:int32[8] := @sort( array, @elements(array), a, b, "abcmp"
);
```

```
@sqrt( real_expr )
```

This function returns the square root of the parameter.

```
@system( string_expr )
```

This function executes the system command specified by the string (i.e., a command-line operation for a shell interpreter). It captures all the output sent to the standard output device by this command and returns that data as a string value.

```
@tan( real_expr )
```

This function returns the tangent of the angle (in radians) passed as a parameter.

```
@time
```

This function returns a string of the form "HH:MM:SS xM" (x= A or P) denoting the time at the point this function was called (according to the system clock).

13.8.3 String functions:

`@delete(str_expr, int_start, int_len)`

This function returns a string consisting of the `str_expr` passed as a parameter with (possibly) some characters removed. This function removes `int_len` characters from the string starting at index `int_start` (note that strings have a starting index of zero).

`@index(str_expr1, int_start, str_expr2)`

This function searches for `str_expr2` within `str_expr1` starting at character position `int_start` within `str_expr1`. If the string is found, this function returns the index into `str1_expr1` of the first match (starting at `int_start`). This function returns -1 if there is no match.

`@insert(str_expr1, int_start, str_expr2)`

This function inserts `str_expr2` into `str_expr1` just before the character at index `int_start`.

`@left(str_expr, int_length)`

This function returns the left-most `int_length` characters of the specified string.

`@leftdel(str_expr, int_length)`

This function deletes the left-most `int_length` characters of the specified string and returns the result.

`@length(str_expr)`

This function returns the length of the specified string.

`@lowercase(str_expr, int_start)`

This function returns a string of characters from `str_expr` with all uppercase alphabetic characters converted to lower case. Only those characters from `int_start` on are copied into the result string.

`@replace(str_expr1, str_expr2, str_expr3)`

This function searches for every occurrence of `str_expr2` found within `str_expr1`. It replaces each occurrence found with `str_expr3` and returns the resultant string.

`@replace(str_expr1, str_array_expr2)`

An extended version of the `@replace` function. The second argument must be an array of strings with an even number of elements. For each pair of strings this function will replace each occurrence of the first string of the pair found in `str_expr1` with the value of the second string in the pair. E.g., `?resultStr := @replace("Hello there world", [{"there", ""}, [{" world", "world"}]);`

`@right(str_expr, int_length)`

This function returns the right-most `int_length` characters of the specified string.

`@rightdel(str_expr, int_length)`

This function deletes the right-most `int_length` characters of the specified string and returns the result.

`@rindex(str_expr1, int_start, str_expr2)`

Similar to the `index` function, but this function searches for the last occurrence of `str_expr2` in `str_expr1` rather than the first occurrence.

`@strbrk(str_expr, int_start, cset_expr)`

This function returns the index of the first character beyond `int_start` in `str_expr` that is a member of the `cset_expr` parameter. This function returns -1 if none of the characters are in the set.

```
@strset( char_expr, int_len )
```

This function returns a string consisting of `int_len` copies of `char_expr`.

```
@strspan( str_expr, int_start, cset_expr )
```

This function returns the index of the first character beyond position `int_start` in `str_expr` that is not a member of the `cset_expr` parameter. This function returns -1 if all of the characters are in the set.

```
@substr( str_expr, int_start, int_len )
```

This function returns the substring specified by the starting position and length in `str_expr`.

```
@tokenize( str_expr, int_start, cset_delims, cset_quotes )
```

This function returns an array of strings obtained by doing a lexical scan of the `str_expr` passed as a parameter (starting at character index `int_start`). Each element of this array consists of all characters between any sequences of delimiter characters (specified by the `cset_delims` parameter). The only exceptions are strings appearing between bracketing (quoting) symbols. The fourth parameter specifies the possible bracketing characters. If `cset_quotes` contains a quotation mark (") then all sequences of characters between a pair of quotes will be treated as a single string. Similarly, if `cset_quotes` contains an apostrophe, then all characters between a pair of apostrophes will be treated as a single string. If the `cset_quotes` parameters contains one of the pairs "(" / ")", "{" / "}", or "[" / "]" (both characters from a given pair must be present), then `@Tokenize` will consider all characters between these bracketing symbols to be a single string.

You should use the `@elements` function to determine how many strings are present in the resulting array of strings (this will always be a one-dimensional array, although it is possible for it to have zero elements).

```
@trim( str_expr, int_start )
```

This function returns a string consisting of the characters in `str_expr` starting at position `int_start` with all leading and trailing whitespace removed.

```
@uppercase( str_expr, int_start )
```

This function returns a string consisting of the characters in `str_expr` starting at position `int_start` with all lower case alphabetic character converted to uppercase.

13.8.4 String/Pattern matching functions

The HLA string/pattern matching functions all attempt to match a string against a pattern. These functions all return a boolean result indicating success or failure (i.e., whether the string matches the pattern).

Most of these functions have two optional parameters: *Remainder* and *Matched*. If the function succeeds it generally copies the matched string into the **val** string constant specified by the *Matched* parameter and it copies all the characters in the `InputStr` parameter the follow the matched text into the *Remainder* parameter. You may specify the *Remainder* parameter without also specifying the *Matched* parameter, but if you need the *matched* result, you must specify all the parameters. The *Remainder* and *Matched* parameters appear in italics in all of the following functions to denote that they are optional.

If the function fails, the values of the *Remainder* and *Matched* parameters are generally undefined.

```
@peekCset( InputStr, charSet, Remainder, Matched )
```

This function checks the first character of `InputStr` to see if it is a member of `charSet`. The function returns **true/false** depending on the result of the set membership test. If the function succeeds it copies the value of the `InputStr` parameter to `Remainder` and creates a single character string from the first character of `InputStr` and stores this into `Matched`.

```
@oneCset( InputStr, charSet, Remainder, Matched )
```

This function checks the first character of `InputStr` to see if it is a member of `charSet`. The function returns **true/false** depending on the result of the set membership test. If the function succeeds, it copies all characters but the first of `InputStr` parameter to `Remainder` and copies the first character of `InputStr` into `Matched`.

```
@uptoCset( InputStr, charSet, Remainder, Matched )
```

This function matches all characters up to, but not including, a single character from the `charSet` character set parameter. If the `InputStr` parameter does not contain a character in the specified character set, this function fails. If it succeeds, it copies all the matched characters (not including the character in the character set) to the `Matched` parameter and it copies all remaining characters (including the character in the character set) to the `Remainder` parameter.

```
@zeroOrOneCset( InputStr, charSet, Remainder, Matched )
```

If the first character of `InputStr` is a member of `charSet`, this function succeeds and returns that character in the `Matched` parameter. It also returns the remaining characters in the string in the `Remainder` parameter.

This function always succeeds (since it matches zero characters). If the first character of `InputStr` is not in `charSet`, then this function returns `InputStr` in `Remainder` and returns the empty string in `Matched`.

```
@exactlynCset( InputStr, charSet, n, Remainder, Matched )
```

This function returns true if the first `n` characters of `InputStr` are in the character set specified by `charSet`. The `n+1st` character must not be in the character set specified by `charSet`. If this function succeeds (i.e., returns true), then it copies the first `n` characters to the `Matched` string and it copies all remaining characters into the `Remainder` string. If this function fails and returns **false**, `Remainder` and `Matched` are undefined.

```
@firstnCset( InputStr, charSet, n, Remainder, Matched )
```

This function is very similar to `exactlynCset` except it doesn't require that the `n+1st` character not be a member of the `charSet` set. If the first `n` characters of `InputStr` are in `charSet`, this function succeeds (returning **true**) and copies those `n` characters into the `Matched` string; it also copies any following characters into the `Remainder` string.

```
@nOrLessCset( InputStr, charSet, n, Remainder, Matched )
```

This function always succeeds. It will match between zero and `n` characters in `InputStr` from the `charSet` set. The `n+1st` character may be in `charSet`, this function doesn't care and only matches up to the `nth` character. This function copies up to `n` matched characters to the `Matched` string (the empty string if it matches zero characters); the remaining characters in the string are copied to the `Remainder` parameter.

```
@nOrMoreCset( InputStr, charSet, n, Remainder, Matched )
```

This function succeeds if it matches at least `n` characters from `InputStr` against the `charSet` set. It returns **false** if there are fewer than `n` characters from `charSet` at the beginning of `InputStr`. If this function succeeds, it copies the characters it matches to the `Matched` string and all characters after that sequence to the `Remainder` string.

```
@ntomCset( InputStr, charSet, n, Remainder, Matched )
```

This function succeeds if `InputStr` begins with at least `n` characters from `charSet`. If additional characters in `InputStr` are in this set, `ntomcset` will match up to `m` characters ($n < m$). It will not match any additional characters beyond the `m`th character, although those characters may be in the `charSet` set without affecting the success/failure of this routine. If this routine succeeds, it copies all the characters it matches to the `Matched` parameter and any remaining characters to the `Remainder` parameter.

```
@exactlyntomCset( InputStr, charSet, n, Remainder, Matched )
```

Similar to the `ntomcset` function, except this function fails if more than `m` characters at the beginning of `InputStr` are in the specified character set.

```
@zeroOrMoreCset( InputStr, charSet, Remainder, Matched )
```

This function always succeeds. If the first character of `InputStr` is not in `charSet`, this function copies `InputStr` to `Remainder`, sets `Matched` to the empty string, and returns **true**. If some sequence of characters at the beginning of `InputStr` is in `charSet`, this function copies those characters to `Matched` and copies the following characters to `Remainder`.

```
@oneOrMoreCset( InputStr, charSet, Remainder, Matched )
```

This function succeeds if `InputStr` begins with at least one character from `charSet`. It will match all characters at the beginning of `InputStr` that are members of `charSet`. It copies the matched chars to the `Matched` string and any remaining characters to the `Remainder` string. It fails if the first character of `InputStr` is not a member of `charSet`.

```
@peekChar( InputStr, Character, Remainder, Matched )
```

This function succeeds if the first character of `InputStr` matches `Character`. If it succeeds, it copies the character to the `Matched` string and copies the entire string (including the first character) to `Remainder`.

```
@oneChar( InputStr, Character, Remainder, Matched )
```

This function succeeds if the first character of `InputStr` is equal to `Character`. If it succeeds, it copies the matched character to `Matched` and any remaining characters to `Remainder`. If it fails, then `Remainder` and `Matched` are undefined.

```
@uptoChar( InputStr, Character, Remainder, Matched )
```

This function matches all characters up to, but not including, the specified character. If fails if the specified character is not in the `InputStr` string. If this function succeeds and returns **true**, it copies the matched character to the `Matched` string and copies all remaining characters to the `Remainder` string (the `Remainder` string will begin with the value found in `Character`). If this function fails, it leaves `Remainder` and `Matched` undefined.

```
@zeroOrOneChar( InputStr, Character, Remainder, Matched )
```

This function always succeeds since it can match zero characters. If the first character of `InputStr` is not equal to `Character` this function returns **true** and sets `Remainder` equal to `InputStr` and sets `Matched` to the empty string. If the first character of `InputStr` is equal to `Character`, then this function returns that character in `Matched` and returns any remaining characters from `InputStr` in `Remainder`.

```
@zeroOrMoreChar( InputStr, Character, Remainder, Matched )
```

This function always succeeds since it can match zero characters. If the first character of `InputStr` is not equal to `Character`, this function returns **true** and sets `Remainder` equal to `InputStr` and sets `Matched` to the empty string. If `InputStr` begins with a sequence of characters that are all equal to `Character`, then this function returns those characters in `Matched` and returns any remaining characters from `InputStr` in `Remainder`.

```
@oneOrMoreChar( InputStr, Character, Remainder, Matched )
```

This function always succeeds since it can match zero characters. If the first character of *InputStr* is not equal to *Character* this function returns **true** and sets *Remainder* equal to *InputStr* and sets *Matched* to the empty string. If *InputStr* begins with a sequence of characters that are all equal to *Character*, then this function returns those characters in *Matched* and returns any remaining characters from *InputStr* in *Remainder*.

```
@exactlynChar( InputStr, Character, n, Remainder, Matched )
```

This function returns true if the first *n* characters of *InputStr* are equal to *Character*. The *n*+1st character cannot be equal to *Character*. If this function succeeds, it returns a string consisting of *n* copies of *Character* in *Matched* and returns any remaining characters in *Remainder*. *Matched* and *Remainder* are undefined if this function returns **false**.

```
@firstnChar( InputStr, Character, n, Remainder, Matched )
```

This function returns true if the first *n* characters of *InputStr* are equal to *Character*. The *n*+1st character may or may not be equal to *Character*. If this function succeeds, it returns a string consisting of *n* copies of *Character* in *Matched* and returns any remaining characters in *Remainder*.

```
@nOrLessChar( InputStr, Character, n, Remainder, Matched )
```

This function always returns **true**. It matches up to *n* copies of *Character* at the beginning of *InputStr*. More than *n* characters can be equal to *Character* and this routine will still succeed. However, this routine only matches the first *n* copies of *Character* in *InputStr*. It copies the matched characters to the *Matched* string and copies any remaining characters to the *Remainder* string.

```
@nOrMoreChar( InputStr, Character, n, Remainder, Matched )
```

The *normorechar* function matches any string that begins with at least *n* copies of *Character*. If it succeeds, it copies the sequence of *Character* chars to the *Matched* string and copies any remaining characters (that must begin with something other than *Character*) to the *Remainder* string. This function fails and returns **false** if the string doesn't begin with at least *n* copies of *Character*. Note that the *Remainder* and *Matched* variables are undefined if this function fails.

```
@ntomChar( InputStr, Character, n, m, Remainder, Matched )
```

This function returns true if the first *n* characters of *InputStr* are equal to *Character*. It will match up to *m* characters ($m \geq n$). The *m*+1st character does not have to be different than *Character*, although this function will match, at most, *m* characters. If this function succeeds, it copies the matched characters to the *Matched* string and any following characters to the *Remainder* string. If this function fails and returns **false**, the values of *Matched* and *Remainder* are undefined.

```
@exactlyntomChar( InputStr, Character, n, m, Remainder, Matched )
```

This function succeeds and returns **true** if there are at least *n* copies of *Character* at the beginning of *InputStr* and no more than *m* copies of *Character* at the beginning of *InputStr*. If this function succeeds, it copies the matched characters at the beginning of *InputStr* to the *Matched* parameter and any following characters to the *Remainder* parameter. If this function fails, the values of *Remainder* and *Matched* are undefined upon return.

```
@peekiChar
```

```
@oneiChar
```

```
@uptoiChar
```

```
@zeroOrOneiChar
```

```
@zeroOrMoreiChar
```

```

@oneOrMoreiChar
@exactlyniChar
@firstniChar
@nOrLessiChar
@nOrMoreiChar
@ntomiChar
@exactlyntomiChar

```

These functions use the same syntax as the standard `xxxxxxChar` functions. The difference is that these functions do a case insensitive comparison of the `Character` parameter with the `InputStr` parameter.

```
@matchStr( InputStr, String, Remainder, Matched )
```

This function checks to see if the string specified by `String` appears as the first set of characters at the beginning of `InputStr`. This function returns **true** if `InputStr` begins with `String`. If this function succeeds, it copies `String` to `Matched` and any following characters to `Remainder`.

```
@matchiStr( InputStr, String, Remainder, Matched )
```

Just like `@matchStr` except this function does a case insensitive comparison.

```
@uptoStr( InputStr, String, Remainder, Matched )
```

The `uptoStr` function matches all characters in `InputStr` up to, but not including, the string specified by `String`. If it succeeds, it copies all the matched characters (not including the string specified by `String`) into the `Matched` parameter and any following characters to `Remainder`. If this function returns **false**, the values of `Remainder` and `Matched` are undefined.

```
@uptoiStr( InputStr, String, Remainder, Matched )
```

Same as `@uptoStr` function except that this function does a case insensitive comparison.

```
@matchToStr( InputStr, String, Remainder, Matched )
```

This function is similar to `@uptoStr` except this function matches all characters up to and including the characters in the `String` parameter.

```
@matchToiStr( InputStr, String, Remainder, Matched )
```

Same as `@matchToStr` except this function does a case insensitive comparison.

```
@matchID( InputStr, Remainder, Matched )
```

This is a special matching function that matches characters in `InputStr` that correspond to an HLA identifier. That is, `InputStr` must begin with an alphabetic character or an underscore and `@matchID` will match all following alphanumeric or underscore characters. If this function succeeds by matching a prefix of `InputStr` that looks like an identifier, it copies the matched characters to `Matched` and all following characters to `Remainder`. This function returns **false** if the first character of `InputStr` is not an underscore or an alphabetic character. Note that the first character beyond a matched identifier can be anything other than an alphanumeric or underscore character and this function will still succeed.

```
@matchIntConst( InputStr, Remainder, Matched )
```

This function matches a string of one or more decimal digit characters (i.e., an unsigned integer constant). The `Matched` parameter, if present, must be an **int32 val** object. If `@matchIntConst` succeeds, it will convert the string to an integer and copy this integer to the `Matched` parameter; it will also copy any characters following the integer string to the `Remainder` parameter.

```
@matchRealConst( InputStr, Remainder, Matched )
```

This function matches a sequence of characters at the beginning of `InputStr` that correspond to a real constant (note that a simple sequence of digits, i.e., an integer, satisfies this). The number may have a leading plus or minus sign followed by at least one decimal digit, an optional fractional part and an optional exponent part (see the definition of an HLA real literal constant for more details). If this function succeeds, it converts the string to a **real80** value and stores this value into `Matched` (which must be a **real80 val** object). The characters after the matched string are copied into the `Remainder` parameter. If this function fails, the values of `Matched` and `Remainder` are undefined.

```
@matchNumericConst( InputStr, Remainder, Matched )
```

This is a combination of `@matchRealConst` and `@matchIntConst`. It checks the prefix of `InputStr`. If it corresponds to an integer constant it will behave like `@matchIntConst`. If the prefix string corresponds to a real constant, this function behaves like `@matchRealConst`. If the prefix matches neither, this function returns **false**.

```
@matchStrConst( InputStr, Remainder, Matched )
```

This function matches a sequence of characters that correspond to an HLA literal string constant. Note that such constants generally contain quotes surrounding the string. If this function returns true, it copies the matched string, minus the quote delimiters, to the `Matched` parameter and it copies the following characters to the `Remainder` parameter. If this function fails, those two parameter values are undefined.

This function automatically handles several idiosyncrasies of HLA literal string constants. For example, if two adjacent quotes appear within a string, `@matchStrConst` copies only a single quote to the `Matched` parameter. If two quoted strings appear at the beginning of `InputStr` separated only by whitespace (a space or any control character other than NUL), then this function concatenates the two strings together. Likewise, any character objects (surrounded by apostrophes or taking the form `#ddd`, `#$hh`, or `##%bbbbbbbb` where `ddd` is a decimal constant, `hh` is a hexadecimal constant, and `bbbbbbbb` is a binary constant) are automatically concatenated into the result string. See the definition of HLA literal constants for more details.

```
@zeroOrMoreWS( InputStr, Remainder )
```

This function always succeeds. It matches zero or more whitespace characters (white space is defined here as a space or any control character other than NUL [ASCII code zero]). This function copies any characters following the white space characters to the `Remainder` parameter (this could be the empty string).

```
@oneOrMoreWS( InputStr, Remainder )
```

This function matches one or more whitespace characters (white space is defined here as a space or any control character other than NUL [ASCII code zero]). If this function succeeds, it copies any characters following the white space characters to the `Remainder` parameter. If this function fails, the `Remainder` string's value is undefined.

```
@WSorEOS( InputStr, Remainder )
```

This function always succeeds. It matches zero or more whitespace characters (white space is defined here as a space or any control character) or the end of string token (a zero terminating byte). This function copies any characters following the white space characters to the `Remainder` parameter (this could be the empty string if it matches EOS or there is only white space at the end of the string).

```
@WSthenEOS( InputStr )
```

This function matches zero or more whitespace characters (white space is defined here as a space or any control character) immediately followed by the EOS token (a zero terminating byte). Technically, it allows a `Remainder` parameter, but such a parameter will always be set to the empty string if this function succeeds, so it's hardly useful to supply the parameter.

```
@peekWS( InputStr, Remainder )
```

This function returns true if the first character of `InputStr` is a white space character. If it succeeds and the `Remainder` parameter is present, this function copies `InputStr` to `Remainder`.

```
@EOS( InputStr )
```

This function returns **true** if `InputStr` is the empty string.

```
@reg( InputStr )
```

This function returns **true** if `InputStr` matches a valid register name.

```
@reg8( InputStr )
```

This function returns **true** if `InputStr` matches a valid eight-bit register name.

```
@reg16( InputStr )
```

This function returns **true** if `InputStr` matches a valid 16-bit register name.

```
@reg32( InputStr )
```

This function returns **true** if `InputStr` matches a valid 32-bit register name.

13.8.5 Symbol and constant related functions and assembler control functions

```
@name( identifier )
```

This function returns a string of characters that corresponds to the name of the identifier (note: after text/macro expansion). This is useful inside macros when attempting to determine the name of a macro parameter variable (e.g., for error messages, etc). This function returns the empty string if the parameter is not an identifier.

```
@type( identifier_or_expression )
```

This function returns a unique integer value that specifies the type of the specified symbol. Unfortunately, this unique integer may be different across assemblies. Do not use this function when comparing types of objects in different source code modules. This is a deprecated function. Future versions of the assembler will return the same value as **@typename**. Do not use this function in new code, and change any existing uses to use **@typename** instead.

```
@typename( identifier_or_expression )
```

This function returns the string name of the type of the identifier or constant expression. Examples include "int32", "boolean", and "real80".


```
@basetype( identifier_or_expression )
```

Similar to **@typename**, except this function returns the underlying primitive type for array and pointer objects. For other types, it behaves just like **@typename**.

```
@ptype( identifier_or_expression )
```

This function returns a small integer constant denoting the primitive type of the specified identifier or expression. Primitive types would include things like **int32**, **boolean**, and **real80**. See the "hla.hhf" header file for the latest set of constant definitions for **@pType**. At the time this was written, the definitions were (though don't count on these particular values):

```
// pType constants.

hla.ptIllegal:= 0;
hla.ptBoolean:= 1;
hla.ptEnum:= 2;

hla.ptUns8:= 3;
hla.ptUns16:= 4;
hla.ptUns32:= 5;
hla.ptUns64:= 6;
hla.ptUns128:= 7;

hla.ptByte:= 8;
hla.ptWord:= 9;
hla.ptDWord:= 10;
hla.ptQWord:= 11;
hla.ptTByte:= 12;
hla.ptLWord:= 13;

hla.ptInt8:= 14;
hla.ptInt16:= 15;
hla.ptInt32:= 16;
hla.ptInt64:= 17;
hla.ptInt128:= 18;

hla.ptChar:= 19;
hla.ptWChar:= 20;

hla.ptReal32:= 21;
hla.ptReal64:= 22;
hla.ptReal80:= 23;
hla.ptReal128:= 24;

hla.ptString:= 25;
hla.ptZString:= 26;
hla.ptWString:= 27;
hla.ptCset:= 28;

hla.ptArray:= 29;
hla.ptRecord:= 30;
hla.ptUnion:= 31;
hla.ptRegex:= 32;
hla.ptClass:= 33;
hla.ptProcptr:= 34;
hla.ptThunk:= 35;
```

```

hla.ptPointer:= 36;

hla.ptLabel:= 37;
hla.ptProc:= 38;
hla.ptMethod:= 39;
hla.ptClassProc:= 40;
hla.ptClassIter := 41;
hla.ptIterator:= 42;
hla.ptProgram:= 43;
hla.ptMacro:= 44;
hla.ptText:= 45;
hla.ptRegExMac:= 46;

hla.ptNamespace:= 47;
hla.ptSegment:= 48;
hla.ptAnonRec:= 49;
hla.ptAnonUnion := 50;
hla.ptVariant:= 51;
hla.ptError:= 52;

// Total Number of ptypes we support:

hla.sizePTypes:= 53;

```

```
@baseptype( identifier_or_expression )
```

This function returns a small integer constant denoting the underlying primitive type of the specified identifier or expression. See the discussion for **@ptype** for details. The difference between **@ptype** and **@baseptype** is that **@baseptype** returns the element type for arrays and the base type for *ptPointer* types.

```
@class( identifier_or_expression )
```

This returns a symbol's class type. The class type is constant, value, variable, static, etc., this has little to do with the class abstract data type. See the "hla.hhf" header file for the current symbol class definitions. At the time this was written, the definitions were:

```

hla.cIllegal:= 0;
hla.cConstant:= 1;
hla.cValue:= 2;
hla.cType := 3;
hla.cVar := 4;
hla.cParm := 5;
hla.cStatic:= 6;
hla.cLabel:= 7;
hla.cProc := 8;
hla.cIterator:= 9;
hla.cClassProc:= 10;
hla.cClassIter := 11;
hla.cMethod:= 12;
hla.cMacro:= 13;
hla.cKeyword:= 14;
hla.cTerminator:= 15;
hla.cRegEx:= 16;
hla.cProgram:= 17;
hla.cNamespace:= 18;
hla.cSegment := 19;

```

```
hla.cRegister := 20;
hla.cNone := 21;
```

`@size(identifier_or_expression)`

This function returns the size, in bytes, of the specified object.

`@elementsize(identifier_or_expression)`

This function returns the size, in bytes, of an element of the specified array. If the parameter is not an array identifier, this function generates an assembly-time error.

`@offset(identifier)`

For **var**, **parm**, **method**, and class **iterator** objects only, this function returns the integer offset into the activation record (or object record) of the specified symbol.

`@staticname(identifier)`

For **static/readonly/storage** objects, procedures, methods, iterators, and external objects, this function returns a string specifying the "static" name of that string. HLA emits this name to the assembly output file for certain objects (when producing an assembly language output file).

`@lex(identifier)`

This function returns an integer constant specifying the static lexical nesting for the specified symbol. Variables declared in the main program have a lex level of zero. Variables declared in procedures (etc.) that are in the main program have a lex level of one. This function is useful as an index into the `_display_array` when accessing non-local variables.

`@IsExternal(identifier)`

This function returns true if the specified identifier is an external symbol.

`@arity(identifier_or_expression)`

This function returns zero if the specified identifier is not an array. Otherwise, it returns the number of dimensions of that array.

`@dim(array_identifier_or_expression)`

This function returns a single array of integers with one element for each dimension of the array passed as a parameter. Each element of the array returned by this function gives the number of elements in the specified dimension. For example, given the following code:

```
val threeD: int32[ 2, 4, 6];
tdDims := @dim( threeD );
```

The `tdDims` constant would be an array with the three elements [2, 4, 6];

`@elements(array_identifier_or_expression)`

This function returns the total number of elements in the specified array. For multi-dimensional array constants, this function returns the number of all elements, not just a particular row or column.

`@defined(identifier)`

This function returns **true** if the specified identifier has been previously defined in the program and is currently in scope.

`@pclass(identifier)`

If the specified identifier is a parameter, this function returns a small integer indicating how the parameter was passed to the function. These constants are defined in the `hla.hhf` header file. At this time this document was written, these constants had the following values.

```
hla.illegal_pc:= 0;
hla.valp_pc:= 1;
hla.refp_pc:= 2;
hla.vrp_pc:= 3;
hla.result_pc:= 4;
hla.name_pc:= 5;
hla.lazy_pc:= 6;
```

`valp_pc` means pass by value. `refp_pc` means pass by reference. `vrp_pc` means pass by value/result (value/returned). `result_pc` means pass by result. `name_pc` means pass by name. `lazy_pc` means pass by lazy evaluation.

```
@localsyms( record_union_procedure_method_or_iterator_identifier )
```

This function returns an array of string listing the local names associated with the argument. If the argument is a record or union object, the elements of the string array contain the field names for the specified record or union. Note that the field names appear in their declaration order (that is, element zero contains the name of the first field, element one contains the name of the second field, etc.).

If the argument is a procedure, method, or iterator, the string array this function returns is a list of all the local identifiers in that program unit. Note that the local object names appear in the reverse order of their declarations (that is, element zero contains the name of the last local name in the program unit, element one contains the second identifier, etc.). Note that parameters are considered local identifiers and will appear in this array. Also note that HLA automatically predefines several symbols when you declare a program unit; those HLA declared symbols also appear in the array of strings `@localsyms` creates.

Currently, `@localsyms` does not allow namespace, program, or class identifiers. This restriction may be lifted in the future if there is sufficient need.

```
@isconst( expr )
```

This function returns true if the specified parameter is a constant identifier or expression.

```
@isreg( expr )
```

This function returns **true** if the specified parameter is one of the 80x86 general purpose registers. It returns **false** otherwise.

```
@isreg8( expr )
```

This function returns **true** if the specified parameter is one of the 80x86 eight-bit general purpose registers. It returns **false** otherwise.

```
@isreg16( expr )
```

This function returns **true** if the specified parameter is one of the 80x86 16-bit general purpose registers. It returns **false** otherwise.

```
@isreg32( expr )
```

This function returns **true** if the specified parameter is one of the 80x86 32-bit general purpose registers. It returns **false** otherwise.

```
@isfreg( expr )
```

This function returns **true** if the specified parameter is one of the 80x86 FPU registers. It returns **false** otherwise.

`@ismem(expr)`

This function returns **true** if the specified expression is a memory address.

`@isclass(expr)`

This function returns **true** if the specified parameter is a class or a class object.

`@istype(identifier)`

This function returns **true** if the specified identifier is a type id.

`@linenumber`

This function returns the current line number in the source file.

`@linenumberstk(expr)`

The expression must be a small unsigned integer value. `@linenumberstk(0)` returns the current line number in the source file (exactly like `@linenumber`). If the expression evaluates to some value larger than zero, the `@linenumberstack` crawls up the macro/include/text expansion/regular expression include stack and prints the line number of the invocation at the level specified by the argument. Note that `@linenumberstk(1)` prints the line number of the invocation of the current macro (or include, etc.). If the expression is larger than the number of entries on the line number stack, this function returns the line number of the first invocation.

`@filename`

This function returns the name of the current source file.

`@filenamestk(expr)`

The expression must be a small unsigned integer value. `@filenamestk(0)` returns the current filename for the source file (exactly like `@filename`). If the expression evaluates to some value larger than zero, the `@filenamestack` crawls up the macro/include/text expansion/regular expression include stack and prints the filename of the invocation at the level specified by the argument. If the expression is larger than the number of entries on the filename stack, this function returns the filename of the main file.

`@curllex`

This function returns the current static lex level (e.g., zero for the main program).

`@curoffset`

This function returns the current **var** offset within the activation record.

`@curdir`

This function returns +1 if processing parameters, it returns -1 otherwise. This corresponds to whether variable offsets are increasing or decreasing in an activation record during compilation. This function also returns +1 when processing fields in a record or class. This function returns zero when processing fields in a union.

`@addofs1st`

This function returns **true** when processing local variables, it returns **false** when processing parameters and record/class/union fields.

`@lastobject`

This function returns a string containing the name of the last macro object processed.

`@curobject`

This function returns a string containing the name of the last class object processed.

@curvar

This function returns a string containing the name of the last memory object processed.

13.8.6 Pseudo-Variables

HLA provides several special identifiers that act as functions in expressions and as variables in **val** assignments. These "pseudo-variables" let you control the code emission during compilation. Typically, you would use these pseudo-variables in a statement like "?@bound:=true;" in order to set their values.

@errorprefix

This variable contains a string (default the empty string). Whenever the assembler reports an error message, it first checks this string to see if it is not the empty string. If the string is not the empty string, then HLA will print this string before printing the error message. This pseudo-variable is useful when processing macros and an error message might not appear until deep into the macro expansion. The programmer can set up a helpful string (perhaps using the @lineNumberStack and/or @fileNameStack functions) to print when an error occurs.

@parmoffset

This variable contains the starting offset for parameters. This is generally eight for most procedures since the parameters start at offset eight. You can change this value during assembly by assigning a value to this variable (e.g., ?@parmoffset = 10;). However, this activity is not recommended except by advanced programmers.

@localoffset

This variable returns the starting offset for local variables in an activation record. This is typically zero. You can change this value during assembly by assigning a value to this variable (e.g., ?@localoffset = -10;). However, this activity is not recommended except by advanced programmers.

@basereg

This variable returns a string containing either "ebp" or "esp". You assign either ebp or esp (the registers, not a string) to this variable. This sets the base register that HLA uses for automatic (**var**) variables. The default is ebp. Examples:

```
?SaveBase :string := @basereg;
?@basereg := esp;
<< code that uses esp to access locals and parameters>>
?@basereg := @text( SaveBase ); // Restore to original register.
```

Note the use of @text to convert the string to an actual register name. This must be done because HLA only allows the assignment of the actual ebp/esp registers to @basereg, not a string.

@enumsiz

This assembly time variable specifies the size (in bytes) of enumerated objects. This has a default value of one.

@minparmsiz

This assembly time variable has the initial value four. You should not change the value of this object when running under Win32, *NIX, or other 32-bit OS.

@bound

This assembly time variable is a boolean value that indicates whether HLA compiles the **bound** instruction into actual machine code (or ignores the **bound** instruction).

@into

This assembly time variable is a boolean value that indicates whether HLA compiles the **into** instruction into actual machine code.

`@label`

This assembly time variable is an integer value that must be assigned a value greater than zero. This value controls how HLA generates internal unique symbols. HLA normally translates non-external/non-public symbols to some form such as "*originalSymbol_XXX_nnnn*" where *originalSymbol* is the identifier appearing in the HLA source file, *XXX* is some special string (currently "HLA" as this is being written, but this is subject to change in future versions of HLA), and *nnnn* is a decimal integer string. HLA increments the value of *nnnn* for each symbol it generates, thus ensuring that all internal symbols are unique within a given source file.

A problem can occur with HLA's unique symbol generation algorithm if you're generating an assembly language source file for use with an assembler such as MASM that has an option to make all symbols public. Usually, symbols of the form "*originalSymbol_XXX_nnnn*" are private to a given source file, such symbols are almost never public. However, if you compile HLA code to a MASM source file and then compile the MASM code with the "all symbols public" option, it's quite possible, when linking multiple files together, that you wind up with duplicate symbol errors from the linker. In such (rare) cases, you can use the `@label` pseudo-variable to work around this problem by changing the value HLA uses for its internal label counter. For example, if the linker complains that the symbol "false_HLA_1023" is multiply defined, you can use the `@label` pseudo-variable to change the symbol number suffixes in one of the source files using a statement like following near the beginning of your source file:

```
?@label := 5000;
```

Be careful about using this pseudo-variable; you should only change the value once and you should only change it near the beginning of a source file. If you reset the `@label` value to a smaller value somewhere beyond the start of the source file, you can create internal symbol conflicts in your source file. Use this option with care!

`@exceptions`

This assembly time variable controls whether HLA emits full exception handling code or an abbreviated set of routines. If this variable contains **true**, then HLA emits the full exception handling code. If **false**, the HLA emits the minimal amount of code to pass exceptions on to Windows or *NIX. Note that this variable only affects code generation in the main **program**, it does not affect the code generation in a **unit**. This variable must be set to true before the **begin** clause associated with the main program if it is to have any effect. Note that including the EXCEPTS.HHF file automatically sets this to true; so you will have to explicitly set it to **false** if you include this file (or some other file that includes EXCEPTS.HHF, like STDLIB.HHF).

`@optstring`

By default, HLA *folds* string constants to generate better code. This means that whenever you ask the compiler to emit code for a string constant like "Hello World" the compiler will first check to see if it has already emitted such a string. If so, the compiler uses the reference to the original string constant rather than emitting a second copy of the string; this shortens the size of your program if there are multiple occurrences of the same string in the program. Since string constants generally go into a read-only section of memory, the program cannot accidentally change this unique occurrence. The `@optstrings` pseudo-variable lets you control this optimization. If `@optstrings` is **true** (the default condition), then HLA folds all duplicate string constants; if `@optstrings` is **false**, then HLA emits duplicate strings to the code.

`@trace`

This boolean variable controls the emission of "trace" statements by the HLA compiler. This feature is offered in lieu of a decent debugger for tracing through HLA programs. When this variable is false (the default), HLA emits the code you specify. However, if you set this compile-time variable to true, HLA emits the following code before most statements in the program:

```
_traceLine_( filename, lineNumber );
```

The filename parameter is a string that specifies the current filename HLA is processing. The *linenumber* parameter is an **uns32** value that specifies the current line number in the file. You are

responsible for supplying the `"_traceLine_"` procedure somewhere in your program. Here's a typical implementation:

```
procedure trace( filename:string; linenum:uns32 ); @external(
  "_traceLine_" );
procedure trace( filename:string; linenum:uns32 ); @nodisplay;
begin trace;

  pushfd(); // This function must preserve all registers and flags!
  stdout.put( filename, ": #", linenum, nl );
  popfd();

end trace;
```

As the comments above note, it is your responsibility to preserve all registers and flags in the `_traceLine_` procedure. If you fail to do this, it will corrupt those values in the code that calls `_traceLine_` .

A common operation inside the `_traceLine_` procedure is to display register values. Don't forget that EBP's and ESP's values are modified by this call. Furthermore, if you do any processing whatsoever at all, the flag values will change. To obtain EBP's value prior to the call, fetch the double-word at address `[EBP+0]`. To obtain ESP's value, take the value of EBP inside `_traceLine_` and subtract 16 from it (EBP, return address, and eight bytes of parameters are on the stack). Obviously if you build `_traceLine_` 's activation record yourself, these values can change. To display the flag values, access the copy of the FLAGS register you pushed on the stack (at offset `[EBP-4]` in the code above).

In addition to simply displaying values, you can write some very sophisticated debugging routines that let you set breakpoints, watch values, and so on. Someday the HLA Standard Library will include some trace support functions, until then have fun doing whatever you want.

13.8.7 Text emission functions

```
@text( str_expr )
```

This function replaces itself with the text of the specified parameter. The result is then processed by HLA. E.g.,

```
@text( "mov( 0, eax );" );
```

The above is equivalent to the single move instruction.

```
@string( identifier )
```

The identifier must be a constant of type text. HLA replaces this item with the string data assigned to the text object.

```
@string:identifier
```

The identifier must be a constant of type text. HLA replaces this item with the string data assigned to the text object. Note that this operation is deprecated. HLA now allows `@string(textVal)` to convert a text object to a string value.

```
@tostring:identifier
```

Like `@string:identifier` , the identifier must be a constant of type text. Also like `@string:identifier` , HLA replaces this item with the string data assigned to the text object. However, this function also converts `identifier` from a text to a string object.

13.8.8 Miscellaneous Functions

@section

This function returns a 32-bit bitmap that identifies the current point in the source. Identification is as follows:

```

Bit 0:    Currently processing the CONST section.
Bit 1:    Currently processing the VAL section.
Bit 2:    Currently processing the TYPE section.
Bit 3:    Currently processing the VAR section.
Bit 4:    Currently processing the STATIC section.
Bit 5:    Currently processing the READONLY section.
Bit 6:    Currently processing the STORAGE section.

Bit 12:   Currently processing statements in the "main" program.
Bit 13:   Currently processing statements in a procedure.
Bit 14:   Currently processing statements in a method.
Bit 15:   Currently processing statements in an iterator.
Bit 16:   Currently processing statements in a #macro.
Bit 17:   Currently processing statements in a #keyword macro.
Bit 18:   Currently processing statements in a #terminator macro.
Bit 19:   Currently processing statements in a thunk.

Bit 23:   Currently processing statements in a Unit.
Bit 24:   Currently processing statements in a Program.

Bit 25:   Currently processing statements in a record.
Bit 26:   Currently processing statements in a union.
Bit 27:   Currently processing statements in a class.
Bit 28:   Currently processing statements in a namespace.

```

This function is useful in macros to determine if a macro expansion is legal at a given point in a program.

13.9 #Text and #endtext Text Collection Directives

The #TEXT and #ENDTEXT directives surround a block of text in an HLA program from which HLA will create an array of string constants. The syntax for these directives is:

```

#text( identifier )

    << arbitrary lines of text >>

#endtext

```

The *identifier* must either be an undefined symbol or an object declared in the VAL section.

This directive converts each line of text between the #text and #endtext directives into a string and then builds an array of strings from all this text. After building the array of strings, HLA assigns this array to the identifier symbol. This is a **val** constant array of strings. The #text..#endtext directives may appear anywhere in the program where white space is allowed.

Although these directives provide an easy way to initialize a constant array of strings, the real purpose for these directives is to allow the inclusion of Domain Specific Embedded Language (DSEL) text within an HLA program. Presumably, a parser (written with macros, regular

expression macros, and the HLA compile-time language) would process the statements between the **#text** and **#endtext** directives.

13.10#String and #endstring Text Collection Directives

The **#string** and **#endstring** directives surround a block of text in an HLA program from which HLA will create a single string constant. The syntax for these directives is:

```
#string( identifier )

    << arbitrary lines of text >>

#endstring
```

Either the *identifier* must be an undefined symbol or an object declared in the **val** section.

These directives are similar in principle to the **#text..#endtext** directives except that they produce a single string (including new line characters) holding the entire block of text rather than an array of strings.

Although these directives provide an easy way to initialize a string, the real purpose for these directives is to allow the inclusion of Domain Specific Embedded Language (DSEL) text within an HLA program. Presumably, a parser (written with macros, regular expression macros, and the HLA compile-time language) would process the statements between the **#string** and **#endstring** directives.

13.11 Regular Expression Macros and the @match/@match2 Functions

Regular expression macros contain sequences of pattern-matching statements that you can use to determine if some string takes a particular form. With HLA's regular expression macros and the attendant **@match** and **@match2** functions, you can develop sophisticated language processors inside HLA and specify whatever syntax you like (well, within certain bounds) for those languages.

Technical Note: although these features are called "regular expression macros", the purists out there will note that "regular expression" is actually a misnomer here. HLA's regular expression macros actually handle a subset of the context-free languages. This language facility is called "regular expression macros" because most programmers, even those not intimately familiar with automata theory, recognize the term and associate "pattern matching" with the term. Hence the use of the term "regular expression" when "context-free grammar" would probably be a better choice. For those of you who aren't intimately familiar with automata theory design, fear not: the context-free languages are a proper superset of the regular languages and you're not being short-changed here. HLA's "regular expression" macros will actually handle all the stuff you can do with a regular expression, and more.

Before describing the syntax for a regular expression macro, it's probably best to begin by discussing how you use them in a program. This will better motivate you when this document actually discusses the regular expression syntax.

Regular expressions are used for pattern matching.¹ Generally, a regular expression is applied to some string of text and a boolean "success (matched) / failure (no match)" result comes back from the operation. The HLA compile-time function **@match** (and **@match2**) is how you achieve this task. The basic syntax for the **@match**² function is the following:

-
1. Actually, the purists will argue that regular expressions are used for pattern *generation*, not recognition. Because these two facilities are technically equivalent in theoretical computer science, this documentation will ignore this issue and claim that regular expressions are pattern matching devices.
 2. For brevity, this document will use **@match** to imply the use of **@match** or **@match2**. The two functions are almost identical in usage other than how they handle whitespace.

```
@match( stringToMatch, RegexMacroName, ReturnsResult, Remainder,
MatchedString )
```

This function returns the boolean result `true` if the regular expression specified by *RegexMacroName* matches some prefix of the string *stringToMatch*. The remaining three arguments are optional, though if one argument is present then any preceding arguments must also be present.

The optional *ReturnsResult* argument must be an HLA **val** identifier. The **@match** function will store a special **#return** string into this **val** object. We'll look at what a **#return** string is a little later in this documentation. For now, suffice to say that this is the "text" that the regex macro expands into (regex macros do not expand in-place as standard HLA macros do). If this argument is not present and the regex macro produces a **#return** string, then HLA simply throws away the associated string data.

The optional *Remainder* argument must be an HLA **val** identifier. If this argument is present, then the *ReturnsResult* argument must also be present. This argument is identical to the "remainder" arguments of the string matching functions given earlier. When matching *stringToMatch* with *RegexMacroName*, the regex macro might not match the entire string, only a prefix of the string (this is still a successful match). Any remaining characters that are not matched once **@match** exhausts the regular expression are collected and stored into the *Remainder* argument, if it is present. **@match** will not generate this string if you do not pass the *Remainder* argument (and the string information is simply thrown away at that point).

The optional *MatchedString* argument must be an HLA **val** identifier. If this argument is present, then the *Remainder* and *ReturnsResult* arguments must also be present. This argument is identical to the "matched" arguments of the string matching functions given earlier. If the regular expression macro successfully matches *stringToMatch*, then **@match** will store a copy of the sequence that has been matched into this **val** argument.

Note that if the **@match** function returns false, because *RegexMacroName* failed to match the characters in *stringToMatch*, then **@match** will not disturb the existing values of the *ReturnsResult*, *Remainder*, and *MatchedString* parameters. Therefore, you should only expect those arguments to contain reasonable values if **@match** returns true.

13.11.1 #regex.#endregex

The syntax for a regular expression macro is very similar to a standard macro declaration. Here is the basic form:

```
#regex macroName ( optional_parameter_list ) : optional_locals_list;

    << regex body >>

#endregex
```

The *optional_parameter_list* and *optional_locals_list* items are identical (in syntax) to a macro declaration. The following **#regex** statements demonstrate some of the legal permutations:

```
#regex noParmsOrLocals;
#regex onParmNoLocals( oneParm );
#regex oneLocalNoParms:oneLocal;
#regex variableParms( a, b, c[] );
#regex stringParms( string parms );
```

It's actually a somewhat rare occurrence for a regular expression macro to have parameters. The semantics for parameters (and locals) are different for compiled and precompiled regular expression macros. Therefore, it's a good idea to avoid using parameters unless they are necessary.

The body of a **#regex** macro consists of zero or more regular expression items following by an optional **#return** clause. If the regular expression body is empty, then the regular expression will match the empty string, which means it will match any string appearing in an **@match** function call. The section *Regular Expression Elements* describes the exact syntax for the body of a regular expression macro. The next section describes the optional **#return** clause.

13.11.2 The #return Clause

A **#regex** macro declaration may optionally contain a **#return** clause immediately after the regular expression body (and immediately before the **#endregex** clause). The **#return** clause specifies a string expression to return (via the *ReturnsResult* argument in the **@match** function call). Here is a typical example:

```
#regex newMov;
  <<body for newMov>>
#returns "mov( eax, ebx )"
#endregex
```

Note that an arbitrary HLA string expression is legal after the **#returns** clause, not just a simple literal constant. So you can use the concatenation operation (+) or any other HLA compile-time string functions to build up the **#return** string. Note that there is no semicolon at the end of the string expression. The **#endregex** properly terminates the string expression.

If no **#return** clause is present in a **#regex** macro, then that **#regex** macro returns the empty string as the **#return** string result.

The main purpose for the **#return** clause is to return some text to expand in the invoking code should the **@match** function succeed. Unlike standard macros, you cannot expect to be able to arbitrarily expand text found in a **#regex** macro because you only "invoke" **#regex** macros in an **@match** function call, and those generally appear in a compile-time boolean expression. For example, if the **#regex** macro above directly emitted the **mov** instruction during the invocation of this macro, you'd get syntax errors whenever you made calls like:

```
#if( @match( "Hello World", newMov ) )
  .
  .
#endif
```

because HLA would emit the **mov** instruction right into the boolean expression associated with the **#if** statement (which is syntactically incorrect). By putting the **#return** value into a string and returning that string result, the system can defer the expansion of the text until the caller gets to an appropriate context, e.g., (from earlier)

```
#if( @match( "Hello World", newMov, returnResult ) )

  @text( returnResult );

#endif
```

This example expands the "mov(eax, ebx)" instruction if and only if the pattern matches "Hello World".

If you would like the default situation to be "expand text if match" then it's easy enough to write a macro to do this job for you:

```
#macro expand( theStr, theRegex ):returnResult;

  #if( @match( theStr, theRegex, returnResult ) )

    @text( returnResult );

  #endif

#endmacro

.
.
expand( "Hello World", newMov );
```

The return string is automatically processed by the `#match(regex)..#endmatch` block. See the description of `#match..#endmatch` for more details.

13.11.3 Regular Expression Elements

The "meat" of a regular expression macro is the sequence of regular expression elements that appear in a `#regex` macro body. Each element in a regular expression body can match a part of the source string. The following subsections describe each regular expression element in detail.

With only a couple exceptions (that will be noted as they arrive), each time a regular expression element matches a character in the source string (the first parameter provided to `@match`), the match operation *consumes* that character. For example, if the source string is "Hello World" and the first regular expression element matches the single character 'H', then 'H' is consumed from the source string (yielding "ello World") and further regular expression elements operate on that remainder of the string.

13.11.4 Kleene Star, Plus, and Numeric Range Specifications

Most regular expression elements we're about to explore match a single instance of themselves. For example, a literal character constant in the body of a regular expression macro will match a single character in the source string (see the next section). You can modify this match operation by supplying one of the following suffixes to the literal character constant.

Suffix	Meaning
*	(Kleene star) Matches zero or more occurrences of the preceding operand.
+	(Kleene plus) Matches one or more occurrences of the preceding operand.
: <i>[n]</i>	Matches exactly <i>n</i> occurrences of the preceding operand. <i>n</i> must be a reasonably-valued unsigned integer constant expression.
: <i>[n,m]</i>	Matches between <i>n</i> and <i>m</i> occurrences of the preceding operand. <i>n</i> and <i>m</i> must be reasonable unsigned integer constants with <i>n</i> < <i>m</i> .
: <i>[n,*]</i>	Matches <i>n</i> or more occurrences of the preceding operand. <i>n</i> must be a reasonably-valued unsigned integer constant expression.

Examples:

```
'c'*  Matches zero or more 'c' characters.
'c'+  Matches one or more 'c' characters.
'c':[4] Matches exactly four 'c' characters.
'c':[4,6] matches between four and six 'c' characters.
'c':[4,*] Matches four or more 'c' characters.
```

Exceptions to this syntax will be noted whenever they occur.

13.11.5 Matching Characters in a Regular Expression

A character literal constant within a `#regex` body matches the corresponding character in the source string. For example, the following regular expression macro matches a string beginning with the single character 'c':

```
#regex matchesC;
```

```
    'c'
#endregex
```

Note that this form only allows a single character constant. In particular, you cannot specify an arbitrary HLA character expression. However, you can also use the HLA **@matchChar** (synonym: **@oneChar**) function in a regular expression body to specify a character expression. **@matchChar** requires a single parameter that must evaluate to a single character. For example,

```
#regex matchesC;
    @matchChar( char( uns8('b') + 1) ) // Matches 'c'
#endregex
```

The single character match operation consumes a single character from the beginning of the source string if it successfully matches the first character of the source string.

Examples of character matching repetition:

```
'c'*   Matches zero or more 'c' characters.
'c'+   Matches one or more 'c' characters.
'c':[4] Matches exactly four 'c' characters.
'c':[4,6] matches between four and six 'c' characters.
'c':[4,*] Matches four or more 'c' characters.
@matchChar( char( uns8('b') + 1) ) * Matches zero or more 'c' characters
```

13.11.6 Case-insensitive Character Matching in a Regular Expression

You can perform a case-insensitive character match by prefixing a literal character constant with the "!" operation. For example, `!'c'` matches either 'c' or 'C'. Here is an explicit example:

```
#regex matchesCorc;
    !'c'
#endregex
```

If you want to specify a character expression rather than a single literal character constant, you can use the **@matchiChar** function in a manner similar to **@matchChar** given earlier. This operation also consumes a single character from the source string if a match occurs.

Examples of character matching repetition:

```
!'c'*   Matches zero or more 'c' or 'C' characters.
!'c'+   Matches one or more 'c' or 'C' characters.
!'c':[4] Matches exactly four 'c' or 'C' characters.
!'c':[4,6] matches between four and six 'c' or 'C' characters.
!'c':[4,*] Matches four or more 'c' or 'C' characters.
@matchiChar( char( uns8('b') + 1) ) * Matches zero or more 'c' or 'C'
characters
```

Note that repetitive matches allow any combination of upper and lower case characters. For example, `!'c'+` will match the sequence "ccCcCCc".

13.11.7 Negated Character Matching

Sometimes you'll want to match "anything but a given character." The HLA **#regex** macro body provides a shortcut for matching anything but a single character. By placing a minus sign in front of a single literal character constant, you can tell HLA to match anything but that character. E.g., `-'c'` matches anything but the 'c' character. You can combine this with the "!" operator to match anything but the upper or lower case version of a character. For example, `!'c'` matches anything but 'c' or 'C'.

There is no generic function you can call like `@matchChar` or `@matchiChar` if you want to specify a character expression rather than a character literal constant. However, you can easily achieve the same effect by using negated character sets. See the discussion of matching character sets a little later in this documentation.

If the first character of the source string is not the specified literal constant, then this operation consumes the first character of the source string.

Examples of character matching repetition:

```
-`c'*  Matches zero or more characters that are not `c'.
-`c'+  Matches one or more characters that are not `c'.
-`c':[4] Matches exactly four characters that are not `c'.
-`c':[4,6] matches between four and six characters that are not `c'.
-`c':[4,*] Matches four or more characters that are not `c'.
```

13.11.8 String Matching in Regular Expressions

A string literal constant within a `#regex` body matches the corresponding sequence of characters in the source string. For example, the following regular expression macro matches a string beginning with the sequence "str":

```
#regex matchesC;
    "str"
#endregex
```

Note that this form only allows a single literal string constant. In particular, you cannot specify an arbitrary HLA string expression. However, you can also use the HLA `@matchStr` function in a regular expression body to specify a string expression. `@matchStr` requires a single parameter that must evaluate to a single string. For example,

```
#regex matchesHelloWorld;
    @matchStr( "Hello " + "World" ) // Matches "Hello World"
#endregex
```

The string match operation consumes one character from the source string for each character in the regular expression element, but only if the match is completely successful. This is, if the first few characters of the source string match the regular expression element but not all the characters match, then the operation consumes no characters.

Although it is not commonly done, the repetition operations apply to string objects as well as characters. Examples of string matching repetition:

```
"str"*  Matches zero or more "str" sequences.
"str"+  Matches one or more "str" sequences.
"str":[4] Matches exactly four "str" sequences.
"str":[4,6] matches between four and six "str" sequences.
"str":[4,*] Matches four or more "str" sequences.
@matchStr( "Hello" + " world" ) * Matches zero or more "Hello world"
sequences.
```

13.11.9 Case-insensitive String Matching in Regular Expressions

Like character matching, you can do a case-insensitive string match by prefixing a string literal constant with "!" or by using the `@matchiStr` function. E.g.,

```
#regex caseInsensitive;
    @matchiStr( "Hello world" )
#endregex
```

Another example:

```
#regex caseInsensitive;
    !"Hello world"
#endregex
```

Although it is not commonly done, the repetition operations apply to string objects as well as characters. Examples of case-insensitive string matching repetition:

```
!"str"* Matches zero or more "str" sequences (case insensitive).
!"str"+ Matches one or more "str" sequences (case insensitive).
!"str":[4] Matches exactly four "str" sequences (case insensitive).
!"str":[4,6] matches between four and six "str" sequences (case
insensitive).
!"str":[4,]* Matches four or more "str" sequences (case insensitive).
@matchiStr( "Hello" + " world" )*
    Matches zero or more "Hello world" sequences (case insensitive).
```

13.11.10 Negated String Matching

You can put the "-" operator in front of a string literal expression to specify that the match should fail if the following characters match a given string. For example,

```
#regex caseInsensitive;
    -"Hello world"
#endregex
```

will succeed as long as the next 11 characters are not "Hello world". You can also apply the case-insensitive operator to this sequence,, e.g., -!"Hello worrld".

Note: negated string matching never consumes any characters from the source string. That is, once this pattern succeeds, the source string contains the same data it did before the match operation. Character consumption doesn't make sense for this operation because the source string could actually be shorter than the negated match string (in which case we still want the pattern to succeed because the source string doesn't begin with the negated string).

The repetition operators do not apply to negated string-matching operations.

13.11.11 String List Matching

The following regular expression syntax tells HLA to successfully match if any one of a list of strings matches the front of the source string:

```
[ "string1", "string2", ..., "stringn" ]
```

The match operation fails only if all the strings in the list fail to match the front of the source string. If multiple strings match the start of the source string, then the first string in the list is the one that will match. So if you want a maximal match, put the longest strings at the beginning of the list, e.g.,

```
[ "these", "the", "th" ]
```

Similarly, if you want a minimal match, put the shortest strings first in the list.

If this operation succeeds, then it consumes the matching characters from the source string.

The repetition operators do not apply to string list matching operations. If you really need this capability, use the alternation operator (discussed later).

13.11.12 Character Set Matching in a Regular Expression

A character set literal constant within a **#regex** body matches a character from the set in the source string. For example, the following regular expression macro matches a string beginning with any of the character 'c', 's', 'e', or 't':

```
#regex matchesC;
    { 'c', 's', 'e', 't' }
#endregex
```

Note that this form only allows a single character set constant. In particular, you cannot specify an arbitrary HLA character set expression. However, you can also use the HLA **@matchCset** (synonym: **@oneCset**) function in a regular expression body to specify a character set expression. **@matchCset** requires a single parameter that must evaluate to a single character. For example,

```
#regex matchesC;
    @matchCset( -{ 'c', 'C' } + numericCset ) // Matches anything but 'c',
    'C', or a digit
#endregex
```

The single character set match operation consumes a single character from the beginning of the source string if it successfully matches the first character of the source string.

Examples of character matching repetition:

```
{ '0'..'9' }*   Matches zero or more digit characters.
{ '0'..'9' }+   Matches one or more digit characters.
{ '0'..'9' }:[4] Matches exactly four decimal digit characters.
{ '0'..'9' }:[4,6] matches between four and six decimal digit characters.
{ '0'..'9' }:[4,*] Matches four or more digit characters.
@matchCset( {"0123456789"})* Matches zero or more digit characters
```

13.11.13 Negated Character Set Matching

Although you can use the **@matchCset** function to specify a negated character set (e.g., **@matchCset(-someSet)**), for simple literal character set constants HLA allows a shortcut operation. Just put a minus sign in front of the literal character set constant. E.g., **-{ 'c', 'C', 'd', 'D' }** matches anything except upper/lower case C and D.

13.11.14 Matching Arbitrary Characters

You can match a single character (regardless of its value) using the negated empty character set (i.e., **-{ }**). However, HLA provides a shortcut for this - the period operator. A period appearing in regular expression body will match any single character and consume that character from the source string. It only fails if there are no more characters in the source string.

```
. * Matches zero or more characters.
.+ Matches one or more characters.
.: [4] Matches exactly four characters.
.: [4,6] matches between four and six characters.
.: [4,*] Matches four or more characters.
```

The **.*** pattern is useful at the beginning of a pattern if you want to match some subsequent pattern anywhere in the source string. The **.*** pattern will skip over any characters up to the desired pattern.

Note that there are some performance issues (at compile time) concerning the use of the repeated **"."** operator in complex regular expressions. Please see the section on regular expression performance later in this document.

13.11.15 Sequences (Concatenation) - The ',' Operator

Most regular expressions will consist of more than a single regular expression item. The "," operator lets you create a sequence of regular expression items in a regular expression macro. The resulting regular expression is effectively a concatenation of the match semantics. For example, consider the following regular expression macro:

```
#regex identifier;
  { 'a'..'z', 'A'..'Z', '_' }, { 'a'..'z', 'A'..'Z', '_' }*
#endregex
```

This regular expression matches a sequence of characters that begin with at least one alphabetic or underscore character followed by zero or more alphanumeric or underscore characters (i.e., the definition of an HLA identifier). Here is another example that matches signed integer literal constants:

```
#regex intConst;
  '-' : [0,1], { '0'..'9' }+
#endregex
```

The repetition operators do not apply to sequences (they apply, instead, to the last element of the regular expression sequence). See the discussion of parentheses ("()") for a way to apply a repetition to a sequence.

13.11.16 Alternation - The "|" Operator

The alternation operator ("|") lets HLA select from amongst several different alternative regular expression elements. The basic syntax is:

```
RX1 | RX2
```

where RX1 and RX2 are two regular expressions (e.g., the regular expression elements we've discussed thus far). The **@match** function will try to match the first regular expression against the source string. If this succeeds, then the whole expression succeeds and the **@match** function ignores the second alternative. If matching the first regular expression fails, then the **@match** function tries to match against the second regular expression. The success or failure of the match is then based on the result of this second match.

Because $R | S$ is itself a regular expression, recursively we can come up with an arbitrary list of alternatives, e.g.,

```
RX1 | RX2 | RX3 | RX4 | ... | RXn
```

The **@match** function will try to match the first expression. If that fails, it will try the second; if that fails, it will try the third, etc. If any of the n regular expressions succeeds, then the alternation succeeds and **@match** ignores any remaining regular expressions in the alternation expression. The alternation sequence fails only if all the subpatterns fail. Note that the string list operator, ["str1", "str2", str3", ..., "strn"] is just a shorthand for:

```
"str1" | "str2" | ... | "strn"
```

The repetition operators do not apply to alternative sequences (they apply, instead, to the last element of the alternation sequence). See the discussion of parentheses ("()") for a way to apply a repetition to an alternation sequence.

13.11.17 Subexpressions - The "()" operator

Like arithmetic operators, regular expression operators exhibit operator precedence. The precedence order is repetitive operators (e.g., "*" and "[2]"), sequences ("("), and last, alternation ("|"). This precedence is natural and eliminates some ambiguity that would otherwise be present in a regular expression. For example, consider the following regular expression sequence:

```
'c', 'd' | 'e'
```

Does this mean match the string "cd" or "e" (that is, match 'c', 'd' or match 'e'), or does this mean match either of the strings "cd" or "ce" (that is, match 'c' followed by 'd' or 'e')? An argument could be made for either resolution of the ambiguity. However, the '|' operator has higher precedence than the '|' operator in HLA, so the first possibility is the one that HLA uses (that is, it matches "cd" or "e").

No matter which choice is made with respect to precedence, there will be situations where you need to override the precedence. As for arithmetic expressions, you can use the parentheses to override precedence. For example, if you really want to match "cd" or "ce" in the previous example, you could rewrite the expression as follows:

```
'c', ( 'd' | 'e' )
```

You may apply the repetition operators to a parenthetical regular expression. For example, the regular expression

```
'c', ( 'd' | 'e' )*
```

matches the character 'c' followed by a string of zero or more 'd' and 'e' characters.

Some regular expression items don't directly support the repetition operators. For example, sequences don't support the repetition operators (because of precedence issues). You can use parentheses to overcome this problem, e.g.,

```
('a', 'b', {'c', 'd'}):+
```

matches a sequence of characters containing "abc" or "abd" (or both) repeated one or more times.

Note: some operators don't support repetition because it just doesn't make sense to do so. Be careful when you force repetition on to an operation that doesn't otherwise support it. It's very easy to create a regular expression that never succeeds, or always succeeds, by misapplying the repetition operators.

13.11.18 Extracting Substrings - The Extraction Operator "<>:"

On occasion, you'll want to save some part of the source string you've matched. Granted, the **@match** function has a *MatchedString* argument that returns the entire matched string, but sometimes you'll want to extract only a portion of the entire matched string. The regular expression extraction operator lets you achieve this. The extraction operator uses the following syntax:

```
< Regular_Expression_sequence >:identifier
```

For the purposes of pattern matching, the extraction operator behaves exactly like the subexpression (parentheses) operator. Everything between the two angle brackets ("**<**" and "**>**") is used as a unit. If this sequence matches the source string, then the **@match** function will extract the substring matched by this subexpression and store that string into the compile-time variable specified by *identifier*. This identifier must be a regular expression macro parameter, a regular expression local symbol, or a global **val** object.

One very common use of the **#return** statement is to return some string composed of items processed by the extraction operator. For example, if you want to create a LISP-like assembly language, you could use a regular expression macro like the following (for the **mov**, **add**, and **sub** instructions):

```
#regex stmt:mnemonic, op1, op2;

\(',
<["mov", "add", "sub"]>:mnemonic, // Match the mnemonic
\,',
<.*>:op1, // Everything up to the 2nd comma is the 1st operand
\,',
<.*>:op2, // Everything up to the ')' is the 2nd operand
```

```

    \)'
#return mnemonic + "(" + op1 + "," + op2 + ")" //Construct HLA statement
#endmacro

```

13.11.19 Invoking Other #regex Macros in a Regular Expression

HLA's **#regex** macros allow you to call other **#regex** macros as though they were pattern matching functions. This one feature alone is what gives HLA's "regular expressions" the power to handle many context-free grammars (rather than being limited to just the regular language subset). If you include the name of some **#regex** macro within a regular expression, the **@match** function will match the current source string using that other regular expression and its success or failure will determine if the match proceeds upon return from that other **#regex** macro. Consider the following example:

```

#regex ID;
    { 'a'..'z', 'A'..'Z', '_' }, { 'a'..'z', 'A'..'Z', '0'..'9', '_' }*
#endregex

#regex arrayAccess;
    ID, '[', { '0'..'9' }+, ','
#endregex

```

The `arrayAccess` regular expression matches an identifier followed by a numeric constant surrounded by braces, e.g., `myArray[4]`.

Regular expression invocations can even be recursive. However, you must be careful not to create an infinitely recursive loop (that is, creating a "left recursive" expression, using compiler terminology). Advanced HLA users (and hopefully you are an advanced HLA user if you're reading this stuff) might think that they can use HLA's conditional assembly directives (e.g., **#if**) to halt the recursion. Though the compile-time language elements may appear in a **#regex** macro, they don't work the way you probably think that they do; in particular, they cannot be used to terminate left recursion. There primary ways to make decisions in regular expressions is via success/failure and via alternation. Specifically, if you have two regular expressions *R* and *S*, then the expression "*R*, *S*" will not execute *S* if *R* fails. Similarly, the sequence "*R* | *S*" will not execute *S* if *R* succeeds. If these two sequences are inside *S*, then you can stop infinite recursion via the success or failure of *R*.

Eliminating left recursion (and left factoring, another important operation for creating grammars that a predictive parser like **@match** can use) is a subject well beyond the scope of this manual. Pick up any decent compiler design text for details.

There are some important compile-time performance issues associated with invoking regular expression macros from within another regular expression.

13.11.20 Lookahead (peeking)

Sometimes when matching a string, you'll need to look ahead one or more characters to determine whether you can satisfy the current regular expression. A classic example is the "less than" operator in many programming languages ("*<*"). A simple regular expression of the form '*<*' is insufficient because the next character might be "*=*" or "*>*" (for languages that use "*<*" to denote 'not equals', such as HLA). Of course, with HLA's regular expressions you could use the string list ["*<=*", "*<*", "*>*"] to handle this specific match, but in general you might want the ability to look ahead a character or two before deciding if you're going to succeed. This is accomplished using the peek operator and functions.

For literal constants, prefacing the constant with "/" tells the **@match** function that the following literal constant must appear in the source string, but **@match** will not consume any of those characters. For example, `'a'/'b'` requires that the source string begin with "ab" but it only consumes the 'a' from the source string. Similarly, `!"ax"/-{'a'..'z', 'A'..'Z', '0'..'9', '_'}` matches "ax" (case-insensitive) as long as whatever follows is not an alphanumeric or underscore character (btw, this expression isn't quite good enough, you'll also want to allow end of string after the "ax", but we haven't discussed how to match end of string yet, so that will have to wait).

You can also use the `@peekChar`, `@peekiChar`, `@peekStr`, `@peekiStr`, and `@peekCset` functions to look ahead without consuming any characters in the source string. E.g, this last example is equivalent to:

```
!"ax" @peekCset (-{'a'..'z', 'A'..'Z', '0'..'9', '_' } )
```

13.11.21 Utility Matching Functions

HLA's regular expression macros support several utility functions that match common strings, thus sparing you from having to write regular expressions for these common items. The following table lists the built-in functions.

Name	Parameters	Supports Repetition	Description
<code>@eos</code>		No	Matches the end of the string.
<code>@ws</code>		Yes	Matches a whitespace character.
<code>@reg</code>		No	Matches an x86 general-purpose 8, 16, or 32-bit register.
<code>@reg8</code>		No	Matches an x86 8-bit register name.
<code>@reg16</code>		No	Matches an x86 16-bit register name.
<code>@reg32</code>		No	Matches an x86 32-bit register name.
<code>@regfpu</code>		No	Matches an x86 FPU register name (HLA syntax: <code>st0, st1, ..., st7</code>).
<code>@regmmx</code>		No	Matches an x86 MMX register name (HLA syntax: <code>mm0, mm1, ..., mm7</code>).
<code>@regxmm</code>		No	Matches an x86 SSE register name (HLA syntax: <code>xmm0, xmm1, ..., xmm7</code>).
<code>@matchid</code>		No	Matches a sequence that looks like an HLA identifier (begins with alphabetic or underscore, followed by zero or more alphanumeric or underscore characters).
<code>@matchIntConst</code>		No	Matches a sequence of one or more decimal digits.
<code>@matchRealConst</code>		No	Matches a sequence that is a syntactically (HLA) valid floating-point literal constant.
<code>@matchStrConst</code>		No	Matches an HLA string literal (including quotes around the object).

@matchWord	("string")	No	Similar to @matchStr (or "literal String") except that the next character after the string it matches must not be alphanumeric or underscore.
@matchiWord	("string")	No	Case-insensitive variant of @matchWord .
@arb		Yes	Matches an arbitrary character. Similar to '.' but uses a lazy algorithm rather than a greedy algorithm (that is, it matches as few characters as possible rather than as many characters as possible when the repetition operator allows an arbitrary number of characters).
@pos	(n)	No	<i>n</i> is a small unsigned integer. This pattern succeeds if the current character being matched is the <i>n</i> th character in the <i>original</i> source string (the one passed to @match). Note that the first character in the string is at @pos(0) .
@tab	(n)	No	<i>n</i> is a small unsigned integer. This pattern succeeds if <i>n</i> is greater than or equal to the current character position in the original source string. If the current character position is less than <i>n</i> , then @tab matches all characters up to the <i>n</i> th position. Note that the first character in the string is at @tab(0) .
@at:identifier		No	This function stores the current zero-based index into the source string into the val object <i>identifier</i> (identifier can also be a #regex parameter or local symbol). The type of this value is uns32 .

13.11.22 Backtracking

#regex regular expressions fully support *backtracking* during pattern matching. This means that if a regular expression ambiguously specifies the text to match (and most non-trivial regular expressions are ambiguous), then the **@match** function will back up and try possible alternatives if one possibility fails. The most obvious example is the alternation operator. If you have a regular expression of the form $R | S$ and *R* fails to match, then the **@match** function will "back track" in the source string to where *R* began its match ('unconsuming' any characters consumed by *R*) and retry the match using *S*.

Alternation certainly isn't the only case where backtracking occurs. Consider the following regular expression:

```
.*, "hello"
```

This regular expression matches the string "hello" anywhere in the source string. The `.*` prefix skips over an arbitrary number of characters and then "hello" must match some substring of the source string. Note that the `.*` regular expression is *greedy*. That is, it will match as many characters as possible. Indeed, when `@match` first encounters `.*`, it will match the remainder of the string. Such a match, of course, will cause the next pattern ("hello") to fail as there are no characters left in the string. When this happens, `@match` will back up some characters (up to the first character that `.*` matched) and then see if the following regular expression matches. If so, then `@match` succeeds. If `@match` backs up all the way in the source string to where `.*` began matching in the source string. The `@match` function fails only if it back tracks all the way to the start of what `.*` matches and then the subsequent pattern still fails.

One thing to note here: because `.*` is greedy, a regular expression like `.*, "hello"` will match everything up to the *last* occurrence of "hello" in the source string, not up to the first occurrence. If you would prefer to match up to the first "hello" in the source string, you cannot use a greedy algorithm when skipping arbitrary characters. The `@arb` function matches arbitrary characters, like `.`, except it uses a lazy (or deferred) matching algorithm, matching as few characters as possible. An expression like `@arb*` begins by matching zero characters. If the subsequent pattern fails, it matches one character. If the subsequent pattern fails, it tries matching two characters, and so on. Therefore, the regular expression `@arb*, "hello"` will match up to the first occurrence of "hello" in the source string.

Backtracking can be a very expensive operation if you're not careful when designing your regular expressions. Consider the following regular expression:

```
'a'+, 'a'+, 'a'+
```

This regular expression (ambiguously) matches three or more 'a' characters. Consider what happens, however, when it is fed a source string such as "aaa". The first `'a'+` term above matches the entire string. This causes the second `'a'+` term to fail, so backtracking occurs. The first `'a'+` term backs off one character and now the second `'a'+` term can succeed. At this point, the third `'a'+` term fails. So the second `'a'+` expression attempts to backtrack, but it fails to match, so the first `'a'+` term backs up one more character. Now, the second `'a'+` term greedily grabs the two available characters. The third `'a'+` term fails at this point, so backtracking occurs yet again. The second `'a'+` term backs up one character and, finally, the third `'a'+` term succeeds. As you can see, this is a lot of work to match a three-character string. In general, backtracking is exponential time complexity (that is, the number of backtracking operations that can take place is proportional to 2^{*n} , where n is the number of regular expression elements). Fortunately, with a little care, you can usually avoid the degenerate cases that exhibit such poor performance. For example, the previous expression could be efficiently written as `'a':[3,*]`.

Matching an arbitrary number of characters is best done at the end of a regular expression rather than at the beginning or in the middle of a regular expression. Doing so reduces the amount of backtracking that will take place. If you cannot avoid matching an arbitrary sequence of characters, then the next best thing to avoid is having two or more subexpressions in a regular expression that match arbitrary expressions. When you have two or more subexpressions that can match an arbitrary number of characters, backtracking can get ugly. Fortunately, you can usually avoid such degenerate cases by carefully choosing your regular expressions.

13.11.23 Lazy Versus Greedy Evaluation

By default, the algorithms that `@match` uses are greedy. That is, if a given subexpression can match an arbitrary number of characters it will attempt to match as many as possible. If matching too many would cause the match operation to fail, then backtracking will come to the rescue and allow the pattern match to succeed (if at all possible). If all you care about is whether the pattern matches, then it really doesn't matter whether the match algorithm is greedy or non-greedy. There are two cases, however, where you might want to use a non-greedy ("lazy") algorithm: compile-time performance and minimal string matching.

As you saw in the previous section on backtracking, using a greedy algorithm can produce very slow performance in certain degenerate situations. A lazy algorithm (which matches as few

characters as possible rather than as many characters as possible) will generally produce much better performance as it can reduce the amount of backtracking that takes place. For example, if you could run the 'a'+, 'a'+, 'a'+ algorithm from the previous section using lazy evaluation, then it would match the first three 'a' characters it finds and stop. No backtracking would take place.

Another issue with greedy evaluation is that it always matches the maximum length string. Perhaps this is not what you want. Perhaps you want to match the minimal length string and then process the remainder of the string (after the match) separately. For example, you might expect the following pattern to match everything up to "hello" in the source string and leave the rest of the source string in the remainder operand:

```
.*, "hello"
```

In fact, this regular expression matches everything up to the last occurrence of "hello" in the source string. Therefore, if the source string is something like "hello world, hello people, hello creation" then the remainder string winds up being " creation". Sometimes you want minimal string matching so greedy evaluation is inappropriate.

You can specify lazy evaluation in a pattern using the following repetition forms (assume *R* is some regular expression that supports repetition):

```
R::[n,m]  Matches between n and m copies of R
R::[n,*]  Matches n or more copies of R
```

Although you cannot directly specify lazy evaluation for the unadorned * and + operators, you can easily synthesize lazy evaluation for these operators as follows:

```
R::[0,*]  Matches zero or more copies of R
R::[1,*]  Matches one or more copies of R
```

13.11.24 The @match and @match2 Functions

Consider a simple regular expression that matches a string of the form "id+id" (that is, a simple arithmetic expression). The `#regex` macro might take the following form:

```
#regex simpleExpr;
    @matchID, '+', @matchID
#endregex
```

and you could use this regular expression with an `@match` invocation like this:

```
?boolResult := @match( "value1+value2", simpleExpr );
```

This will work great right up to the point you try something like the following, at which point the pattern matching operation will fail:

```
?falseResult := @match( "value1 + value2", simpleExpr );
```

(notice that there are spaces around the '+' operator in the source string.)

You can solve this problem, and allow arbitrary whitespace in an expression, by inserting `@ws*` regular expressions at appropriate points in your regular expression. For example, you could rewrite *simpleExpr* thusly:

```
#regex simpleExpr;
    @ws*, @matchID, @ws*, '+', @ws*, @matchID
#endregex
```

This new regular expression will ignore whitespace at all the appropriate points in the source string.

There are three problems with sticking `@ws*` terms throughout your regular expression. First, it clutters up the regular expression and makes it difficult to read. Second, it's easy to misplace (or

leave out) one of the `@ws*` terms. Finally, a bunch of terms like `@ws*` can have a serious impact on the processing time needed by `@match` when backtracking occurs.

The `@match2` function solves these three problems. `@match2` automatically skips any white space present before each term it finds in a regular expression that it processes. This spares you having to clutter your code with `@ws*` items, it guarantees that it skips whitespace before each term, and the whitespace it skips is not subject to backtracking issues. Therefore, unless you want absolute control over matching whitespace in your source strings, you should really use the `@match2` function rather than `@match`.

In some very rare cases, you may need the ability to switch between `@match` and `@match2` semantics within the same regular expression. For example, if you want to be able to parse HLA-style character constants, you might be tempted to use a regular expression like the following:

```
"'''' " | \\'\'', ., \\'\''
```

(That is, match `''''` or a single character surrounded by apostrophes.)

Unfortunately, if you use `@match2` to process this regular expression it will fail when you attempt to match the character constant `' '`. This is because `@match2` will skip the space between the two apostrophes. To avoid this problem, the solution is to make a recursive call to `@match` within the regular expression, as follows:

```
"'''' " | @match( \\'\'', ., \\'\'\' )
```

This guarantees `@match` semantics (no whitespace skipping) for the specified subexpression. Note that there are no returns, remainder, or matched parameters allowed here, and the source string is always the current string being processed.

You can also call `@match2` in a similar manner if you want to guarantee `@match2` semantics in a subexpression.

13.11.25 Compiling and Precompiling Regular Expressions

To improve pattern matching performance, particularly when backtracking occurs, HLA does not interpret the text of a `#regex` macro directly. Instead, HLA compiles a `#regex` macro into an internal format and operates on that internal format rather than on the `#regex` text directly. This effects the operation and usage of `#regex` macros in several subtle ways. To avoid complications when using `#regex` macros, it's important to understand how compiling `#regex` macros affects their operation.

Prior to the introduction of `#regex` macros, there were two distinct times a programmer had to be concerned with: assembly (compile) time and run time. For example, the `#if` statement operates at compile time whereas the `if` statement operates at run time. In order to fully utilize the HLA compile-time language, a programmer has to become comfortable with the difference between compile-time operations and run-time code. `#regex` regular expressions also exhibit two distinct phases - compile time and run time - though the confusing part is that both of these phases take place during the HLA compilation phase. Unfortunately, and this is the confusing part, the complete facilities of the HLA compile-time language are only available during regular expression compilation, not while HLA is executing those regular expressions.

Consider, for a moment, the following `#regex` macro definition:

```
#regex sample( count );
  #for( i:= 1 to count )
    'a',
  #endfor
  'b'
#endregex
```

At first glance, this code seems rather straightforward. You would think that it would match the number of 'a' characters passed as the parameter, followed by a single 'b' character. In fact, the behavior is subtly different. As for machine instructions, the `#for` loop simply replicates the body while compiling the regular expression. Once compiled, the number of matching 'a' characters is

immutable. For example, if you compile a regular expression using the value 5 as the actual argument value, the above regular expression macro is equivalent to:

```
#regex sample( count );
    'a', 'a', 'a', 'a', 'a',
    'b'
#endregex
```

Unless you recompile this regular expression with a different argument value, the value will never be anything other than five.

Of course, one question that naturally rises is "how does one compile a **#regex** macro?" None of the examples to date have require the use of a special "regular expression compiler" to process a **#regex** macro before using it. Well, as it turns out, HLA will automatically compile a **#regex** macro to its internal form if you use such a macro within an **@match/@match2** function call or if a **#regex** macro name appears within some other regular expression. Because the regular expression is compiled on the spot, the distinction between compile time and run time for the regular expression almost becomes a moot point.

The only problem with compiling a regular expression every time you encounter it is that compilation can be an expensive operation if you recompile a regular expression on each use. Consider the following **#regex** macros:

```
#regex matchHello;
    "hello"
#endregex

#regex hasHello;
    .*, matchHello
#endregex
```

The **.*** operand in *hasHello* guarantees that backtracking will occur within this regular expression. Unfortunately, on each backtracking instance (and there will be five of them in this case), HLA is forced to recompile the regular expression. This is extremely inefficient. For this reason, you should try to avoid placing uncompiled regular expression macro invocations inside a **#regex** definition. Instead, you should precompile the regular expression to the internal form and specify that compiled version. This saves the expense of recompiling the regular expression on each invocation of the internal **#regex** macro.

The obvious question is "how does one precompile a **#regex** macro?" This is accomplished by creating a **val** object of type **regex** and assigning a **#regex** macro to that **val** identifier. For example:

```
#regex matchHello;
    "hello"
#endregex

val
    compiledMatchHello :regex := matchHello;
```

When HLA sees a statement like this, it compiles the **#regex** macro (*matchHello* in this example) to the internal form and stores this internal data structure into the **regex val** object (*compiledMatchHello* in this example). Now you can use the compiled variant of the **#regex** macro just like the macro itself with one very important difference - compiled regexes do not allow any actual arguments. The processing of the **#regex** parameters (and any HLA compile-time language statements appearing in the macro) takes place when the **#regex** macro is compiled, the statements that would make use of those compile-time language statements is gone when HLA actually executes the regular expression.

If you're only going to use a regular expression macro once in a source file, precompiling the macro won't achieve anything. However, if you use a regular expression macro several times, and especially if you use the **regex** macro within some other regular expression, you should get in the habit of precompiling the **#regex** macro and using the compiled version. Here's a good convention

to use: prefix your **#regex** macro names with an underscore and then immediately follow the **#regex** macro with a **val** statement that compiles the macro to the unadorned name, e.g.,

```
regex _matchHello;
    "hello"
#endregex

val
    matchHello :regex := matchHello;
```

13.11.26 The #match..#endmatch Block

Although you can use **@match** and regular expression macros as generic pattern-matching functions in your HLA compile-time program, the true intended purpose of these pattern-matching facilities is to allow you to write your own "mini-languages" (i.e., domain-specific languages) directly in your HLA source files. The **#match..#endmatch** directives provide a convenient way to compile such domain-specific languages (DSELS). A **#match..#endmatch** block takes the following form:

```
#match( regexID )

    <<body>>

#endmatch
```

The **#match** directive converts the block of text after the closing parenthesis and up to the **#endmatch** directive into a single string, runs **@match** on this string along with the regular expression specified by *regexID*, and then expands the return string to text if the **@match** function returns true. This is roughly equivalent to:

```
?returnStr:string;
#if( @match( <<body text as a string>>, regexID, returnStr ))

    @text( returnStr );

#endif
```

Here is a hypothetical example of **#match..#endmatch** in action:

```
#match( smallBASIClanguage )

    for i = 1 to 10
        print i
    next i

#endmatch
```

Presumably, the *smallBASIClanguage* regular expression would contain the statements to compile the body of the **#match..#endmatch** statement into the corresponding machine instructions.

13.11.27 Using Regular Expressions in Your Assembly Programs

Unless you've had a firm grounding in compiler theory and pattern-matching theory, you're probably wondering what the heck these **#regex** macros are all about. What do they have to do with assembly language? Although this documentation cannot begin to go into details about automata theory and whatnot, it is useful to describe exactly why you might want to create and use **#regex** macros in your assembly programs.

HLA's standard macro facilities let you extend the HLA language, but you don't have a whole lot of say in the design of the syntax for those macro invocations. Though HLA's *context-free macro facilities* provide many options you just don't see in other assemblers, the truth is that you're stuck using the standard HLA syntax when using macros. Regular expressions give you the ability to design a syntax of your own choosing. You can even create full programming languages inside HLA using **#regex** pattern matching macros. All you need to is place your "program" inside some HLA compile-time string object (e.g., using the **#text..#endtext** directive) and then call **@match** to compile your program.

Examples of **#regex** macros appear in the HLA examples download module. Please grab a copy of these examples to see some working examples of HLA **#regex** macros.

13.12 The **#asm..#endasm** and **#emit** Directives

These directives are deprecated and should not appear in new HLA programs. Much of the need for these statements has gone away over the years as HLA's instruction set was expanded to incorporate most x86 instructions. These statements emit text to an output assembly language source file; obviously, these statements have no effect when HLA produces object code directly.

Probably the biggest use of the **#asm..#endasm** directive today is to emit comments into the assembly language source file that HLA produces. This is useful if you want to mark a section of the assembly language code to determine statement boundaries in the output code. If you use this scheme to inject comments into the output code, you should always encode your comments as follows:

```
;/* comment text */
```

The ";" character begins comments in all output assembly languages except HLA and Gas; the ";" is a statement separator in HLA and Gas (which is an innocuous output character). The "/*" and "*/" sequences are the comment delimiters in HLA and Gas. Of course, the (pseudo-) HLA output from an HLA compilation is not compilable, so it doesn't really matter if you emit correct comment syntax for pseudo-HLA output, but Gas uses the same comment syntax as HLA so that's the best approach to use if you want your output to be portable across all assemblers.

Note that the HLA back engine will also ignore any text after a ';' up to the end of the line. Therefore, you can emit this text when directly producing object files with the HLABE and it will not impact the output code. Here is an example:

```
program seeCode;
begin seeCode;

    #asm
    ; /* Beginning of main program body */
    #endasm

    mov( 0, eax );
    mov( 1, ebx );
    add( eax, ebx );

    #asm
    ; /* End of main program body */
    #endasm

end seeCode;
```

Here is the code that HLA emits with the "-masm -source" command-line parameters for the main program:

```
_HLAMain proc near32

start    proc near32
```

```

start    endp

        call    BuildExcept$__hla_
        pushd   0
        push    ebp
        push    ebp
        lea    ebp, [esp-4]

        ; /* Beginning of main program body */
        mov     eax, 0
        mov     ebx, 1
        add     ebx, eax

        ; /* End of main program body */
QuitMain__hla_::
        pushd   0
        call    dword ptr __imp__ExitProcess@4
_HLMain endp

```

13.13 The #system Directive

The **#system** directive requires a single string parameter. It executes this string as an operating system (shell/command interpreter) operation via the C "system" function call. This call is useful, for example, to run a program during compilation that dynamically creates a text file that an HLA program may include immediately after the **#system** invocation.

Example:

```
#system( "dir" )
```

Note that the **#system** directive is legal anywhere white space is allowable and doesn't require a semicolon at the end of the statement.

13.14 The #print and #error Directives

The **#print** directive displays its parameter values during compilation. The basic syntax is the following:

```
#print( comma, separated, list, of, constant, expressions, ... )
```

The **#print** statement is very useful for displaying messages during assembly (e.g., when debugging complex macros or compile-time programs). The items in the **#print** list must evaluate to constant (**const** or **val**) values at compile time.

A common use for **#print** is to display "TODO" messages during compilation, alerting the programmer to features that have yet to be implemented in the application. This helps remind the programmer that code still needs to be written so they don't forget to incorporate that feature. For example,

```
#print( "TODO: Still need to add expression parser here" )
```

The **#error** directive behaves like **#print** insofar as it prints its parameter to the console device during compilation. However, this instruction also generates an HLA error message and does not allow the creation of an object file after compilation. This statement only allows a single string expression as a parameter. If you need to print multiple values of different types, use string concatenation and the **@string** function to achieve this. Example:

```
#error( "Error, unexpected value. Value = " + #string( theValue ) )
```

Notice that neither the **#print** nor the **#error** statements end with a semicolon.

13.15 Compile-Time File Output (**#openwrite**, **#append**, **#write**, **#closewrite**)

These compile-time statements let you do simple file output during compilation. The **#openwrite** statement opens a single file for output, **#write** writes data to that output file, and **#closewrite** closes the file when output is complete. These statements are useful for automatically generating include files that the source file will include later on during the compilation. These statements are also useful for storing bulk data for later retrieval or generating a log during assembly.

The **#openwrite** statement uses the following syntax:

```
#openwrite( string_expression )
```

This call opens a single output file using the filename specified by the string expression. If the system cannot open the file, HLA emits a compilation error. Note that **#openwrite** only allows one output file to be active at a time. HLA will report an error if you execute **#openwrite** and there is already an output file open. If the file already exists, HLA deletes it prior to opening it (so be careful!). If the file does not already exist, HLA creates a new one with the specified name.

The **#append** statement has the same syntax as **#openwrite**. The difference is that using **#append** will not first delete the file you are opening. Instead, all data written to the file will be appended to the end of the existing file (if any).

The **#write** statement uses the same syntax as the **#print** directive. Note, however, that **#write** doesn't automatically emit a newline after writing all its operands to the file; if you want a newline output you must explicitly supply it as the last parameter to **#write**.

The **#closewrite** statement closes the file opened via **#openwrite** or **#append**. HLA automatically closes this file at the end of assembly if you leave it open. However, you must explicitly close this file before attempting to use the data (via **include** or **#openread**) in your program. Also, since HLA allows only one open output file at a time, you must use **#closewrite** to close the file before you can open another with **#openwrite**.

Warning: Internally, the **#write** statement simply redirects the standard output stream to send output to the write file and then invokes **#print**, restoring the standard output file handle upon return. This creates a minor problem if there is a syntax error in the **#write** operand list -- the error message is written to the output file! If you're having problems with the **#write** output, temporarily change it to **#print** to see if there's an error in the statement. This defect will probably get fixed in some future version.

13.16 Compile-time File Input (**#openread**, **@read**, **#closeread**)

These compile-time statements and function let you do simple file input during compilation. The **#openread** statement opens a single file for input, **@read** is a compile-time function that reads a line of text from the file, and **#closeread** closes the file when input is complete. These statements are useful for reading files produced by **#openwrite**/**#write**/**#closewrite** or any other text file during compilation.

The **#openread** statement uses the following syntax:

```
#openread( filename )
```

The *filename* parameter must be a string expression or HLA reports an error. HLA attempts to open the specified file for reading; HLA prints an error message if it cannot open the file.

The **@read** function uses the following call syntax:

```
@read( val_object )
```

The *val_object* parameter must either be a symbol you've defined in a **val** section (or via "?") or it must be an undefined symbol (in which case **@read** defines it as a **val** object). **@read** is an HLA compile-time function (hence the "@" prefix rather than "#"; HLA uses "#" for compile-time statements). It returns either **true** or **false**, **true** if the read was successful, **false** if the read operation encountered the end of file. Note that if any other read error occurs, HLA will print an error message and return **false** as the function result. If the read operation is successful, then HLA

stores the string it read (up to 4095 characters) into the `val` object specified by the parameter. Unlike `#openread` and `#closeread`, the `@read` function may not appear arbitrarily in your source file. It must appear within a constant expression since it returns a boolean result (and it is your responsibility to check for EOF).

The `#closeread` statement closes the input file. Since you may only have one open input file at a time, you must close an open input file with `#closeread` prior to opening a second file. Syntax:

```
#closeread
```

Example of using compile-time file I/O:

```
#openwrite( "hw.txt" )
#write( "Hello World", nl )
#closewrite
#openread( "hw.txt" )
?goodread := @read( s );
#closeread
#print( "data read from file = ", s )
```

13.17 The Conditional Compilation Statements (`#if`)

The conditional compilation statements in HLA use the following syntax:

```
#if( constant_boolean_expression )

    << Statements to compile if the >>
    << expression above is true.    >>

#elseif( constant_boolean_expression )

    << Statements to compile if the >>
    << expression immediately above >>
    << is true and the first expres->>
    << sion above is false.        >>

#else

    << Statements to compile if both >>
    << the expressions above are false. >>

#endif
```

The `#elseif` and `#else` clauses are optional. As you would expect, there may be more than one `#elseif` clause in the same conditional if sequence.

Unlike some other assemblers and high-level languages, HLA's conditional compilation directives are legal anywhere whitespace is legal. You could even embed them in the middle of an instruction! While directly embedding these directives in an instruction isn't recommended (because it would make your code very hard to read), it's nice to know that you can place these directives in a macro and then replace an instruction operand with a macro invocation.

An important thing to note about this directive is that the constant expression in the `#IF` and `#ELSEIF` clauses must be of type boolean or HLA will emit an error. Any legal constant expression that produces a boolean result is legal here. In particular, you are limited to expressions like those allowed by the HLA HLL IF statement.

Keep in mind that conditional compilation directives are executed at compile-time, not at run-time. You would not use these directives to (attempt to) make decisions while your program is actually running.

13.18 The Compile-Time Loop Statements (#while and #for)

The HLA compile time language also provides a couple of looping structures -- the **#while** loop and the **#for** loop.

The **#while..#endwhile** compile-time loop takes the following form:

```
#while( constant_boolean_expression )

    << Statements to execute as long >>
    << as the expression is true.    >>

#endwhile
```

While processing the **#while..#endwhile** loop, HLA evaluates the constant boolean expression. If it is false, HLA immediately skips to the first statement beyond the **#endwhile** directive.

If the expression is **true**, then HLA proceeds to compile the body of the **#while** loop. Upon encountering the **#endwhile** directive, HLA jumps back up to the **#while** clause in the source code and repeats this process until the expression evaluates false.

Warning: since HLA allows you to create loops in your source code that evaluation during the compilation process, HLA also allows you to create *infinite* loops that will lock up the system during compilation. If HLA seems to have gone off into la-la land during compilation and you're using **#while** loops in your code, it might not be a bad idea to put some **#print** directives into your loop(s) to see if you've created an infinite loop.

Note: because of the limitations of HLA's implementation language (FLEX and BISON), it is not possible to begin a **#while** loop and have the matching **#endwhile** appear in a (different) macro or TEXT constant. When the HLA compiler encounters a **#while** statement it scans the source code looking for the matching **#endwhile** collecting up the statements that make up the body of the loop. During this scan it does not expand TEXT constants or macros. Hence, if you bury the **#endwhile** in a macro or TEXT constant HLA will not be able to find it. For performance and functional reasons, HLA cannot expand macro and TEXT variables during this scan. This is a limitation we will all have to live with until v3.0 of HLA (which will be rewritten in a different language).

The **#for..#endifor** loop can take one of the following forms:

```
#for( loop_control_var := Start_expr to end_expr )

    << Statements to execute as long as the loop control variable's >>
    << value is less than or equal to the ending expression.    >>

#endifor

#for( loop_control_var := Start_expr downto end_expr )

    << Statements to execute as long as the loop control variable's >>
    << value is greater than or equal to the ending expression.    >>

#endifor
```

The HLA compile-time **#for..#endifor** statement is very similar to the for loops found in languages like Pascal and BASIC. This is a definite loop that executes some number of times determine when HLA first encounters the **#for** directive (this can be zero or more times, but the number is computed only once when HLA encounters the **#for**). The loop control variable must be a **val** object or an undefined identifier (in which case, HLA will create a new **val** object with the specified name). In addition, the number control variable must be an eight, sixteen, or thirty-two bit integer value (**uns8**, **uns16**, **uns32**, **int8**, **int16**, or **int32**). In addition, the starting and ending expressions must be values that an **int32 val** object can hold.

The `#for` loop with the `to` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is less than or equal to the ending expression's value. The `#for..to..#endifor` loop increments the loop control variable on each iteration of the loop.

The `#for` loop with the `downto` clause initializes the loop control variable with the starting value and repeats the loop as long as the loop control variable's value is greater than or equal to the ending expression's value. The `#for..downto..#endifor` loop decrements the loop control variable on each iteration of the loop.

Note that the `#for..to/downto..#endifor` loop only computes the value of the ending expression once, when HLA first encounters the `#for` statement. If the components of this expression would change as a result of the execution of the `#for` loop's body, this will not affect the number of loop iterations.

The `#for..#endifor` loop can also take the following form:

```
#for( loop_control_var in composite_expr )

    << Statements to execute for each element present in the expression >>

#endifor
```

The *composite_expr* in this syntactical form may be a string, a character set, an array, or a record constant.

This particular form of the `#for` loop repeats once for each item that is a member of the composite expression. For strings, the loop repeats once for each character in the string and the loop control variable is set to each successive character in the string. For character sets, the loop repeats for each character that is a member of the set; the loop control variable is assigned the value of each character found in the set (you should assume that the extraction of characters from the set is arbitrary, even though the current implementation extracts them in order of their ASCII codes). For arrays, this `#for` loop variant repeats for each element of the array and assigns each successive array element to the loop control variable. For record constants, the `#for` loop extracts each field and assigns the fields, in turn, to the loop control variable.

Examples:

```
#for( c in "Hello" )
    #print( c ) // Prints the five characters 'H', 'e', ..., 'o'
#endifor

// The following prints a..z and 0..9 (not necessarily in that order):

#for( c in { 'a'..'z', '0'..'9' } )
    #print( c )
#endifor

// The following prints 1, 10, 100, 1000

#for( i in [1, 10, 100, 1000] )
    #print( i )
#endifor

// The following prints all the fields of the record type r
// (presumably, r is a record type you've defined elsewhere):

#for( rv in r:[0, 'a', "Hello", 3.14159] )
    #print( rv )
#endifor
```

13.19 Compile-Time Functions (macros)

Keep in mind that HLA macros are text expansion devices that may appear anywhere whitespace is allowed. Therefore, you can use them for so much more than 80x86 instruction synthesis. In particular, along with the "?" operator, you can create compile-time functions. For example, consider the following macro that converts the first character of a string to upper case and forces the remaining characters to lower case:

```
program macroFuncDemo;
#include( "stdio.hhf" );

    #macro Capitalize( s );
        @uppercase( @substr( s,0,1), 0 ) +
        @lowercase( @substr( s, 1, 1000 ), 0)
    #endmacro

static
    Hello: string := Capitalize( "hELLO" );
    World: string := Capitalize( "world" );

begin macroFuncDemo;

    stdout.put( Hello, " ", World, nl );

end macroFuncDemo;
```

13.20 Sample Macro: A Modified IF..ELSE..ENDIF Statement

In this section we'll create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF..ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement some as follows:

```
if( expr1 ) then

    << some 'true' statements >>

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

endif;
```

If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If *expr2* evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we won't actually define a new macro named "if" because if we did (e.g., by using the `#id` statement) we would no longer be able to use standard IF statements in an HLA program (at least, not without the '~' prefix). Further, doing so would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high-level statements.

```

/*****
/*
/* if.hla
/*
/* This program demonstrates a modification of
/* the IF..ELSE..ENDIF statement using HLA's
/* multi-part macros.
/*
/*****

program newIF;
#include( "stdlib.hhf" )

// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression. Syntax:
//
// _if( expression ) _then
//
//     <<statements including nested _if clauses>>
//
// _else // this is optional
//
//     <<statements, but _if clauses are not allowed here>>
//
// _endif
//
//
// Note that nested _if clauses do not have a corresponding
// _endif clause. This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses
// including the first one. Of course, once the code
// encounters an _endif another _if statement may begin.

```

```
// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;

// Just ignore the _then keyword.

keyword _then;

// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro). Identical to the
// above _if implementation except this one does not
// require a matching _endif clause. The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

// If this appears within the _else section, report
// an error (we don't allow _if clauses nested in
// the else section, that would create a loop).

#if( hasElse )

    #error( "All _if clauses must appear before the _else clause" )

#endif

// Handle the _else clause here. All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:

    #endif
```

```

// _endif has two tasks. First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section. Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)

// in the _if clause(s) have a legal target label.

terminator _endif;

    ifDone:
    #if( !hasElse )

        elseLbl:

    #endif

endmacro;

static
    tr:boolean := true;
    f:boolean := false;

begin newIF;

    // Real quick demo of the _if statement:

    _if( tr ) _then

        _if( tr ) _then
        _if( f ) _then

            stdout.put( "error" nl );

        _else

            stdout.put( "Success" );

        _endif

end newIF;

```

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader¹.

1. I.e., I don't even want to have to think about this problem!

13.21 Text Processing, Lexical Analysis and the #text..#endtext Block

Although HLA's multi-part macros are very powerful and flexible, they do have some important limitations if you're trying to create your own statements. In particular, if the statements you want to create require some operands, the multi-part macro invocation forces you to specify those operands within parentheses immediately after the macro's name. While you can probably live with this most of the time, there are some situations where you might want to specify the new language feature using a different syntax. Well, with a bit of work it is certainly possible to do this. HLA's compile-time language actually provides all the tools you need to write a full-fledged compiler. While extending HLA in this fashion is well beyond the scope of this text, it is worthwhile to point you in the right direction, just in case you're dying to do really fancy things with HLA.

The key to creating your own personal structures in HLA lies with the HLA compile-time string and pattern matching functions. These functions let you process strings of data in very complex ways, translating that string data into whatever you please. Combined with HLA's #text..#endtext blocks, which let you copy a portion of your source file into string variables, it is possible to write an HLA compile-time program that processes those portions of your source files. Of course, once you process your source file as string data, you can use any syntax you choose (and support) within that string data. You can design very sophisticated DSELS using this technique.

The #text..#endtext block uses the following syntax:

```
#text( identifier )

    << arbitrary lines of text >>

#endtext
```

The *identifier* symbol must be undefined or a **val** object within the current scope. HLA creates a **val** object named *identifier* that will be an array of strings. Each string in the array will contain one line of text between the #text and #endtext reserved words. The array of strings will contain the text immediately following "#text(identifier)" up to the character just before the #endtext directive.

```
// textDemo.hla
//
// This program demonstrates how the #text and #endtext
// directives operate.

program textDemo;

// A quick demonstration of the #text..#endtext directives:

#text( lines ) Hello
World
how are
you #endtext

// Print out the strings gathered into "lines" above
// so you can see the effect of the #text..#endtext directive:

?i := 0;
#while( i < @elements( lines )

    #print( i, ": ", lines[i], " " )
```

```

        ?i := i + 1;

#endwhile
#print( "-----" )

// A cleaner example (typical of what you would find in DSELS):
#text( MyDSELSource )

    if( x=y && a<b || c<>d ) then
        print "This is my own special language, a=", a;
    endif;

#endtext

// Print the above text (to attempt to actually compile
// those statements in this example!)

?i := 0;
#while( i < @elements( MyDSELSource ) )

    #print( i, ": '", MyDSELSource[i], "'" )
    ?i := i + 1;

#endwhile

begin textDemo;
end textDemo;

```

Demonstration of the #TEXT..#ENDTEXT Directives

The program above prints the following when you compile this program with HLA:

```

0: ' Hello'
1: 'World'
2: 'how are'
3: 'you '
-----
0: ''
1: ''
2: '    if( x=y && a<b || c<>d ) then'
3: '    '
4: '        print "This is my own special language, a=", a;'
5: '    '
6: '    endif;'
7: '    '

```

8: ''

As this example suggests, if you want to create a DSEL (Domain Specific Embedded Language) that supports an arbitrary syntax, you would insert your DSEL statements between the **#text** and **#endtext** directives and then use the HLA compile-time language to process this text in the associated array of strings.

In order to process these statements, one of the first activities will be to break up the text into its constituent parts. In the second example above, this would correspond to breaking up those nine strings into:

```
if
(
x
=
Y
&&
a
<
b
||
c
<>
d
)
then
print
"This is my own special language, a="
,
a
;
endif
;
```

Each of these pieces is called a *lexeme*. Compiler writers call the process of breaking a stream of text up into lexemes *lexical analysis* or *scanning*. A *lexical analyzer* or *scanner* is the code responsible for actually breaking up the text. While a full treatment of lexical analysis is, again, beyond the scope of this document¹, some simple techniques you can use to write a *scanner* are easy to understand and well within the scope of this chapter.

Some languages ignore white space and new lines in the source code; others treat these characters as part of the syntax. For example, a language such as HLA ignores new lines, you can cram your whole program onto a single physical source code line if you so desire². Traditional assemblers, on the other hand, only allow one statement per line and use the new line sequence to separate these statements. In our current example (MyDSELsource), we'll assume that the language ignores white space and new line characters.

Actually, HLA's **#text..#endtext** block automatically eliminates all new lines appearing in the text. Instead of new lines, HLA copies each line of text (sans new line) to a separate string in the string array. For our example this is unfortunate because it would be more convenient to treat the entire block of text as a single string of characters. (Note: you could also use HLA's **#string..#endstring** block to capture all the text into a single string, complete with newline characters; if you do that, then you get to ignore the **#while** loop below; this example uses **#text..#endtext** to demonstrate the process of processing one line at a time.) Therefore, one of the first jobs of the scanner we are going to write is to combine these separate lines of text back together. One simple solution is to execute some (compile-time) code like the following before attempting to process the text:

-
1. That subject belongs in a text on compiler design and implementation.
 2. That would be really bad programming style, but it is legal syntactically.


```
?i := 0;
?source := "";
#while( i < @elements( MyDSELSource ) )

    ?source := source + " " + MyDSELSource[i];
    ?i := i + 1;

#endwhile
```

(Inserting a space between lines is necessary since HLA has removed the original separating new line character sequence. This prevents the end of one line from running directly into the beginning of the next line.)

There are two problems with the code above; first, and least important, is that this code wastes a lot of memory. Once you are done there will be two copies of the source file hanging around in memory. This is especially problematic if there is a lot of text between the **#text** and **#endtext** directives. The second problem with this sequence is that it is slow, especially if it has to process a lot of text.

A better solution is to grab a new line of text only after the scanner has finished processing all the previous text. This is easily handled by including the following compile-time statements at the beginning of the scanner code:

```
// Before executing the following code, you must initialize
// CurrentInput and lineNumber as follows:
//
//     ?lineNumber := 0;
//     ?CurrentInput := MyDSELSource[ 0 ];

?CurrentInput := @trim( CurrentInput, 0 ); // Remove leading spaces from
input.
#while( @length( CurrentInput ) = 0 )

    ?lineNumber := lineNumber + 1;
    #if( lineNumber < @elements( MyDSELSource ) )

        ?CurrentInput := @trim( MyDSELSource[ lineNumber ], 0 );

    #endif

#endwhile
```

Notice that this code only returns an empty string when it exhausts all the lines of text in the **#text.#endtext** block. You may test for "end of file" (or, at least, end of this sequence) by explicitly testing for an empty string after the code above executes. Also, note that this code automatically removes any leading and trailing spaces from the text it processes (the call to **@TRIM** handles this). Therefore, when the above code executes, the first item to process appears in the first character of the *CurrentInput* string (assuming, of course, that *CurrentInput* is not empty).

Extracting single character lexemes from the input string is easy. You can use the **@OneChar** function to see if the first character of a string matches a particular character. For example, if the plus and minus signs are special lexemes in your language, then you can use code like the following to see if *CurrentInput* (from above) begins with one of these characters:

```
#if( @OneChar( CurrentInput, '+', CurrentInput ) )

    << CurrentInput began with a '+'. Note that we've extracted
       the '+' from the beginning of CurrentInput in the call above >>

#elseif( @OneChar( CurrentInput, '-', CurrentInput ) )
```

```

    << CurrentInput began with a '-'. Otherwise this is the same
        as the above. >>

#else ...

```

The compile-time pattern matching functions (e.g., **@OneChar**) only store the remainder characters into the remainder operand (the third parameter above, which is *CurrentInput*) if they return true. Therefore, if **@OneChar** in the first **#if** above does not match a plus sign at the beginning of the *CurrentInput* string, it will not change the value of *CurrentInput*; instead, the **#elseif** clause will test the original string. On the other hand, if the first call to **@OneChar** above discovers that *CurrentInput* does begin with a plus sign, then it stores the characters in *CurrentInput* following the plus sign into the remainder operand (which is *CurrentInput*). This deletes the plus sign from the beginning of the string.

To match specific multi-character lexemes, you would use the compile-time **@MatchStr** function. For example, to match the "&&" lexeme, you would use **@MatchStr** as follows:

```

#if( @MatchStr( CurrentInput, "&&", CurrentInput )

    << Drop down here if CurrentInput begins with "&&" >>
    << (this also extracts "&&" from the string. >>

#else ...

```

Like the **@OneChar** function, the **@MatchStr** call above only deletes the "&&" characters from *CurrentInput* if the string begins with these two characters; otherwise **@MatchStr** does not affect the string.

Extracting single character lexemes is generally quite easy, but you must be careful if some multi-character lexemes begin with the same character as a single character lexeme. For example, "<" is a common single-character lexeme that generally means "less than." Matching "<" as a single character lexeme may create problems if you also need to match the two character lexeme "<=" in your language. If you use the **@OneChar** function as we did above for plus and minus then your code may treat the less than or equal operator as two one-character lexemes rather than as a single two-character lexeme. The solution is to check for the longer lexemes first:

```

#if( @MatchStr( CurrentInput, "<=", CurrentInput )

    << Come here on "<=" >>

#elseif( @MatchStr( CurrentInput, "<>", CurrentInput ) )

    << Drop down here if the string begins with "<>" >>

#elseif( @OneChar( CurrentInput, '<', CurrentInput ) )

    << Do this if string begins with '<' but not "<=" or "<>" >>

#else ...

```

Simple lexemes like operators are very easy to process using HLA functions like **@OneChar** and **@MatchStr**. However, there are many string patterns you will want to recognize that do not consist of simple strings. Two common examples are numeric values and identifiers. To recognize these lexemes we must use a general pattern that matches more than a single string. Fortunately, HLA's compile-time pattern matching functions are up to the task.

Let's consider the example of an unsigned decimal integer constant first. Such lexemes begin with a single numeric digit and may contain zero or more additional numeric digits. So the string is always at least one character long and may be longer, as necessary. Recognizing a character from a set of characters is easy; all you need do is call the **@OneOrMoreCset** function to match the value. The following sample code demonstrates how easy this is:

```

#if( @OneOrMoreCset( CurrentInput, {'0'..'9'}, CurrentInput, theNumber ) )

```

```

    << At this point, we've matched a string of digits,
        "theNumber" contains the string we've matched and
        "CurrentInput" contains the remainder of the string >>

#else ... // It wasn't a numeric lexeme.

```

Note that the call to **@OneOrMoreCset** in the example above supplies the fourth, optional, parameter. If **@OneOrMoreCset** successfully matches a string of digits, it will copy the string it matched into this fourth parameter (which must be a VAL object). This fourth parameter was not necessary in the previous examples because the code knew what string it matched since there was only one possible string it could match. However, the call to **@OneOrMoreCset** above can match a nearly infinite variety of different strings. Since you might actually want to use that value while processing the statements in your language, it's a good idea to save that value for future use, hence the last parameter above. As usual, if **@OneOrMoreCset** fails to match the pattern you specify, it does not affect the values of *CurrentInput* or *theNumber*.

Another common pattern you will often need to recognize is a string that represents an identifier. Different languages may specify identifiers differently, but a common definition is that an identifier must begin with an underscore or an alphabetic character and may contain additional alphanumeric or underscore characters (this matches HLA's definition of an identifier). HLA actually has a special pattern matching function, **@MatchID**, that matches HLA-style identifiers; we will not employ that function here so you can see how to write more complex patterns.

Recognizing an HLA identifier requires two steps: first, we must ensure that the identifier begins with an alphabetic character or an underscore. This is easily accomplished with the following **@PeekCset** function call:

```
@PeekCset ( CurrentInput, { 'a'..'z', 'A'..'Z', '_' } )
```

This call to the **@PeekCset** function returns true if *CurrentInput* begins with an underscore or an alphabetic character, it returns false otherwise. It does not affect *CurrentInput*'s value. Therefore, we can use this function to determine if our identifier begins with an appropriate character. Once we know that it begins with an underscore or alphabetic character, we can easily match the entire identifier by calling **@OneOrMoreCset** as follows:

```
@OneOrMoreCset
(
    CurrentInput,
    { 'a'..'z', 'A'..'Z', '0'..'9', '_' },
    CurrentInput,
    theID
)
```

This call will match all the characters in an identifier and leave those characters in the *theID* string; as usual, it removes the identifier from the beginning of the *CurrentInput* string. You would typically match an identifier using code like the following:

```
#if ( @PeekCset ( CurrentInput, { 'a'..'z', 'A'..'Z', '_' } ) )

#if
(
    @OneOrMoreCset
    (
        CurrentInput,
        { 'a'..'z', 'A'..'Z', '0'..'9', '_' },
        CurrentInput,
        theID
    )
)

    << Okay, we've got an identifier and it's in "theID" >>

```

```

#endif

#else ...

```

If you carefully study the above logic, you might think that you can shorten this to the following code:

```

#if
(
    @PeekCset( CurrentInput, {'a'..'z', 'A'..'Z', '_' } )
    && @OneOrMoreCset
    (
        CurrentInput,
        {'a'..'z', 'A'..'Z', '0'..'9', '_' },
        CurrentInput,
        theID
    )
)

    << Okay, we've got an identifier and it's in "theID" >>

#else ...

```

However, there is a subtle flaw in this logic. The HLA compile-time language uses complete boolean evaluation. Therefore, if the call to **@PeekCset** returns false the code above will go ahead and call **@OneOrMoreCset**. Most of the time, this will not adversely affect anything. However, if the next set of characters in the input stream happen to be a set of numeric digits, the call to **@OneOrMoreCset** will return true. Of course, **false AND true** is still **false**, but don't forget that **@OneOrMoreCset** has the side effect of modifying *CurrentInput*. This is probably not what you've intended to do. If you are intent on using the "&&" operator, you can use code like to following to eliminate the problem with the side effect that the pattern matching functions will produce:

```

#if
(
    @PeekCset( CurrentInput, {'a'..'z', 'A'..'Z', '_' } )
    && @OneOrMoreCset
    (
        CurrentInput,
        {'a'..'z', 'A'..'Z', '0'..'9', '_' },
        Remainder,
        theID
    )
)

    << Okay, we've got an identifier and it's in "theID" >>

    ?CurrentInput := Remainder;

#else ...

```

In this example, the remainder of the string is copied into a temporary variable. The code only overwrites *CurrentInput* (with the temporary value) if the full expression evaluates true.

Most languages will have a set of *reserved words*. Reserved words (or *keywords*) are generally nothing more than identifiers that have special meaning within the context of a language. In the MyDSEL example earlier, it is a good bet that the identifiers *if*, *then*, *print*, and *endif* are all reserved words in this DSEL. The easiest way to handle (a small number of) reserved words is to first recognize them as identifiers and then use a sequence of string comparisons to see if the

identifier you've matched is actually a reserved word. You could use code like the following to do this:

```
#if( @PeekCset( CurrentInput, { 'a'..'z', 'A'..'Z', '_' } ))

    #if
    (
        @OneOrMoreCset
        (
            CurrentInput,
            { 'a'..'z', 'A'..'Z', '0'..'9', '_' },
            CurrentInput,
            theID
        )
    )

    #if( theID = "if" )

        << It's the "IF" reserved word >>

    #elseif( theID = "then" )

        << It's the "THEN" reserved word >>

    #elseif( theID = "endif" )

        << It's the "ENDIF" reserved word >>

    #elseif( theID = "print" )

        << It's the "THEN" reserved word >>

    #else

        << Okay, we've got an identifier and it's in "theID" >>

    #endif

#endif

#else ...
```

HLA lets you design and implement your own complex patterns. However, HLA does contain some built-in pattern matching functions for some common patterns. These include functions that match identifiers (**@MatchID**), integer constants (**@MatchIntConst**), floating-point constants (**@MatchRealConst**), numeric (integer or floating point) constants (**@MatchNumericConst**), and string constants (**@MatchStrConst**). These functions are generally much more convenient to use and certainly more efficient than using patterns you've written to match these types of strings. As long as HLA's idea of an identifier, number, or string is suitable for your application, you should use these pattern-matching functions for these purposes.

In addition to the specialized pattern matching functions above, HLA also provides special pattern matching function that deal with whitespace and the end of a string. These functions include **@ZeroOrMoreWS**, **@OneOrMoreWS**, **@WSorEOS**, **@WSthenEOS**, **@PeekWS**, and **@EOS**. See the HLA Compile-time Language document for more details on these functions.

With the basic tools and techniques out of the way, now it's time to look at how we would actually write a scanner using the HLA macro (compile-time function/procedure) facilities. The following program provides a small lexer for the "MyDSELsource" example above.

```
// textDemo2.hla
//
// This program demonstrates how to write a lexical
// analyzer (scanner) with the HLA compile-time language.

program textDemo2;

// DSEL text to scan:
#text( MyDSELsource )

    if( x=y && a<b || c<>d ) then

        print "This is my own special language, a=", a;

    endif;

#endtext

// Compile-time function that scans the text above.
macro lexer( Input, index ):CurrentInput, Matched;

    ?CurrentInput:string := "";
    ?Matched:string := "";

    #if( @elements( Input ) = 0 )

        "Expected an array of strings as 'lexer' argument"

    #else

        ?CurrentInput := @trim( Input[ index ], 0 );

        // The following #while loop removes all blank lines.

        #while( @length( CurrentInput ) = 0 && index < @elements( Input ) )

            ?index := index + 1;
            #if( index < @elements( Input ) )

                ?CurrentInput := @trim( Input[ index ], 0 );

            #else

                ?CurrentInput := "#endtext";

            #endif

        #endwhile

    #endif
endmacro
```

```
// If we reached the end of the input, just return
// "#endtext" for this example. The demo code that
// calls this function automatically stops after this
// point.

#if( index >= @elements( Input ))

    "#endtext"

#else

    // Okay, we've got a non-empty string.
    // Do the lexical analysis on it.

    #if( @OneChar( CurrentInput, '=', CurrentInput ))

        "=" // Return this item as the lexeme.

    // Note: we must check for "<>" before checking for "<".
    #elseif( @MatchStr( CurrentInput, "<>", CurrentInput ))

        "<>"

    #elseif( @OneChar( CurrentInput, '<', CurrentInput ))

        "<"

    #elseif( @OneChar( CurrentInput, '(', CurrentInput ))

        "("

    #elseif( @OneChar( CurrentInput, ')', CurrentInput ))

        ")"

    #elseif( @OneChar( CurrentInput, ',', CurrentInput ))

        ","

    #elseif( @OneChar( CurrentInput, ';', CurrentInput ))

        ";"

    #elseif( @MatchStr( CurrentInput, "&&", CurrentInput ))

        "&&"

    #elseif( @MatchStr( CurrentInput, "||", CurrentInput ))

        "||"

    #elseif( @MatchStrConst( CurrentInput, CurrentInput, Matched ))

        // For the purposes of this program, put the quotes
```

```

        // back around the string constant (@MatchStrConst
        // removes the delimiting quotes).

        (""" + Matched + "")

#elseif( @MatchID( CurrentInput, CurrentInput, Matched ))

    // We've matched an ID, see if it is actually one
    // of the reserved words:

    #if( Matched = "if" )

        ("rw: if")

    #elseif( Matched = "then" )

        ("rw: then")

    #elseif( Matched = "endif" )

        ("rw: endif")

    #else

        // If it's not one of our reserved words, then
        // just return the ID:

        ("id: " + Matched)

    #endif

#else

    #error( "Unexpected lexeme: " + CurrentInput )
    ?CurrentInput := "";
    ""

    #endif
    ?Input[ index ] := CurrentInput;

#endif

#endif
?CurrentInput:string := "";
?Matched:string := "";

endmacro;

val
    lineNumber := 0;

#while( lineNumber < @elements( MyDSELsource))

    #print( lexer( MyDSELsource, lineNumber ))

```



```
#endwhile
```

```
begin textDemo2;  
end textDemo2;
```

Sample Lexical Analyzer

14 HLA Language Reference and User Manual

14.1 High Level Language Statements

HLA provides several control structures that provide a high level language flavor to assembly language programming. The statements HLA provides are

```
try..unprotect..exception..anyexception..endtry
try..always..endtry
raise
if..then..elseif..else..endif
switch..case..default..endswitch
while..endwhile
repeat..until
for..endfor
foreach..endfor
forever..endfor
break, breakif
continue, continueif
begin..end, exit, exitif
```

JT

JF

These HLL statements provide two basic improvements to assembly language programs: (1) they make many algorithms much easier to read; (2) they eliminate the need to create tons of labels in a program (which also helps make the program easier to read).

Generally, these instructions are "macros" that emit one or two machine instructions. Therefore, these instructions are not always as flexible as their HLL counterparts. Nevertheless, they are suitable for about 85% of the uses people typically have for these instructions.

Do keep in mind, that even though these statements compile to efficient machine code, writing assembly language using a HLL mindset produces intrinsically inefficient programs. If speed or size is your number one priority in a program, you should be sure you understand exactly which instructions each of these statements emits before using them in your code.

The JT and JF statements are actually "medium level language" statements. They are intended for use in macros when constructing other HLL control statements; they are not intended for use as standard statements in your program (not that they don't work, they're just not true HLL statements).

Note: The FOREACH..ENDFOR loop is mentioned above only for completeness. The full discussion of the FOREACH..ENDFOR statement appears a little later in the section on iterators.

14.2 Exception Handling in HLA:try..exception..endtry

HLA uses the TRY..EXCEPTION..ENDTRY and RAISE statements to implement exception handling. The syntax for these statements is as follows:

```
try
  << HLA Statements to execute >>

  << unprotected // Optional unprotected section.
  << HLA Statements to execute >>
>>
```

```
exception( const1 )  
  
    << Statements to execute if exception const1 is raised >>  
  
<< optional exception statements for other exceptions >>  
  
<< anyexception //Optional anyexception section.  
    << HLA Statements to execute >>  
>>  
  
endtry;  
  
raise( const2 );
```

Const1 and *const2* must be unsigned integer constants. Usually, these are values defined in the `excepts.hhf` header file. Some examples of predefined values include the following:

```
ex.StringOverflow  
ex.StringIndexError  
  
ex.ValueOutOfRange  
ex.IllegalChar  
ex.ConversionError  
  
ex.BadFileHandle  
ex.FileOpenFailure  
ex.FileCloseError  
ex.FileWriteError  
ex.FileReadError  
ex.DiskFullError  
ex.EndOfFile  
  
ex.MemoryAllocationFailure  
  
ex.AttemptToDerefNULL  
  
ex.WidthTooBig  
ex.TooManyCmdLnParms  
  
ex.ArrayShapeViolation  
ex.ArrayBounds  
  
ex.InvalidDate  
ex.InvalidDateFormat  
ex.TimeOverflow  
ex.AssertionFailed  
ex.ExecutedAbstract
```

Hardware related exception values:

```
ex.AccessViolation  
ex.Breakpoint  
ex.SingleStep  
  
ex.PrivInstr
```

```

ex.IllegalInstr

ex.BoundInstr
ex.IntoInstr

ex.DivideError

ex.fDenormal
ex.fDivByZero
ex.fInexactResult
ex.fInvalidOperation
ex.fOverflow
ex.fStackCheck
ex.fUnderflow

ex.InvalidHandle
ex.StackOverflow

ex.ControlC

```

This list is constantly changing as the HLA Standard Library grows, so it is impossible to provide a complete list of standard exceptions at this time. Please see the `excepts.hhf` header file for a complete list of standard exceptions. As this was being written, the *NIX-specific exceptions (signals) had not been added to the list. See the `excepts.hhf` file on your *NIX system to see if these have been added. Note that not all OSes support every hardware-related exception value.

The HLA Standard Library currently reserves exception numbers zero through 1023 for its own internal use. User-defined exceptions should use an integer value greater than or equal to 1024 and less than or equal to 65535 (\$FFFF). Exception value \$10000 and above are reserved for use by Windows Structured Exception Handler and *NIX signals.

The `TRY..ENDTRY` statement contains two or more blocks of statements. The statements to *protect* immediately follow the `TRY` reserved word. During the execution of the protected statements, if the program encounters the first exception block, control immediately transfers to the first statement following the `endtry` reserved word. The program will skip all the statements in the exception blocks.

If an exception occurs during the execution of the protected block, control is immediately transferred to an exception handling block that begins with the exception reserved word and the constant that specifies the type of exception.

Example:

```

repeat

    mov( false, GoodInput );
    try
        stdout.put( "Enter an integer value:" );
        stdin.get( i );
        mov( true, GoodInput );

    exception( ex.ValueOutOfRange )

        stdout.put( "Numeric overflow, please reenter ", nl );

    exception( ex.ConversionError )

        stdout.put( "Conversion error, please reenter", nl );

```

```

    endtry;

until( GoodInput = true );

```

In this code, the program will repeatedly request the input of an integer value as long as the user enters a value that is out of range (+/- 2 billion) or as long as the user enters a value containing illegal characters.

TRY..ENDTRY statements can be *nested*. If an exception occurs within a nested TRY protected block, the EXCEPTION blocks in the innermost try block containing the offending statement get first shot at the exceptions. If none of the EXCEPTION blocks in the enclosing TRY..ENDTRY statement handle the specified exception, then the next innermost TRY..ENDTRY block gets a crack at the exception. This process continues until some exception block handles the exception or there are no more TRY..ENDTRY statements.

If an exception goes unhandled, the HLA run-time system will handle it by printing an appropriate error message and aborting the program. Generally, this consists of printing "Unhandled Exception" (or a similar message) and stopping the program. If you include the `excepts.hhf` header file in your main program, then HLA will automatically link in a somewhat better default exception handler that will print the number (and name, if known) of the exception before stopping the program.

Note that TRY..ENDTRY blocks are dynamically nested, not statically nested. That is, a program must actually execute the TRY in order to activate the exception handler. You should never jump into the middle of a protected block, skipping over the TRY. Doing so may produce unpredictable results.

You should not use the TRY..ENDTRY statement as a general control structure. For example, it will probably occur to someone that one could easily create a switch/case selection statement using TRY..ENDTRY as follows:

```

try
    raise( SomeValue );

    exception( case1_const)
        <code for case 1>

    exception( case2_const)
        <code for case 2>

    etc.
endtry

```

While this might work in some situations, there are two problems with this code.

First, if an exception occurs while using the TRY..ENDTRY statement as a switch statement, the results may be unpredictable. Second, HLA's run-time system assumes that exceptions are rare events. Therefore, the code generated for the exception handlers doesn't have to be efficient. You will get much better results implementing a switch/case statement using a table lookup and indirect jump (see the Art of Assembly) rather than a TRY..ENDTRY block.

Warning: The TRY statement pushes data onto the stack upon initial entry and pops data off the stack upon leaving the TRY..ENDTRY block. Therefore, jumping into or out of a TRY..ENDTRY block is an absolute no-no. As explained so far, then, there are only two reasonable ways to exit a TRY statement, by falling off the end of the protected block or by an exception (handled by the TRY statement or a surrounding TRY statement).

The UNPROTECTED clause in the TRY..ENDTRY statement provides a safe way to exit a TRY..ENDTRY block without raising an exception or executing all the statements in the protected portion of the TRY..ENDTRY statement. An unprotected section is a sequence of statements, between the protected block and the first exception handler, that begins with the keyword UNPROTECTED. E.g.,

```

try

```

```

    << Protected HLA Statements >>

unprotected

    << Unprotected HLA Statements >>

exception( SomeExceptionID )

    << etc. >>

endtry;

```

Control flows from the protected block directly into the unprotected block as though the UNPROTECTED keyword were not present. However, between the two blocks HLA compiler-generated code removes the data pushed on the stack. Therefore, it is safe to transfer control to some spot outside the TRY..ENDTRY statement from within the unprotected section.

If an exception occurs in an unprotected section, the TRY..ENDTRY statement containing that section does not handle the exception. Instead, control transfers to the (dynamically) nesting TRY..ENDTRY statement (or to the HLA run-time system if there is no enclosing TRY..ENDTRY).

If you're wondering why the UNPROTECTED section is necessary (after all, why not simply put the statements in the UNPROTECTED section after the ENDTRY?), just keep in mind that both the protected sequence and the handled exceptions continue execution after the ENDTRY. There may be some operations you want to perform after exceptions are released, but only if the protected block finished successfully. The UNPROTECTED section provides this capability. Perhaps the most common use of the UNPROTECTED section is to break out of a loop that repeats a TRY..ENDTRY block until it executes without an exception occurring. The following code demonstrates this use:

```

forever

    try

        stdout.put( "Enter an integer: " );
        stdin.geti8(); // May raise an exception.

    unprotected

        break;

    exception( ex.ValueOutOfRange )

        stdout.put( "Value was out of range, reenter" nl );

    exception( ex.ConversionError )

        stdout.put( "Value contained illegal chars" nl );

    endtry;

endfor;

```

This simple example repeatedly asks the user to input an `int8` integer until the value is legal and within the range of valid integers.

Another clause in the TRY..EXCEPT statement is the *ANYEXCEPTION* clause. If this clause is present, it must be the last clause in the TRY..EXCEPT statement, e.g.,

```

try
  << protected statements >>

  <<
    unprotected

        Optional unprotected statements
  >>

  << exception( constant ) // Note: may be zero or more of
                            of these.

        Optional exception handler statements
  >>

    anyexception
        << Exception handler if none of the others execute >>

endtry;

```

Without the ANYEXCEPTION clause present, if the program raises an exception that is not specifically handled by one of the exception clauses, control transfers to the enclosing TRY..ENDTRY statement. The ANYEXCEPTION clause gives a TRY..ENDTRY statement the opportunity to handle any exception, even those that are not explicitly listed. Upon entry into the ANYEXCEPTION block, the EAX register contains the actual exception number.

14.3 Exception Handling in HLA:try..always..endtry

The HLA TRY..ALWAYS..ENDTRY statement is a variant of the try..endtry statement that has a single ALWAYS block (no EXCEPTION or ANYEXCEPTION clauses). This statement takes the following form:

```

try
  << protected statements >>

    always

        Statements that always execute

endtry;

```

The ALWAYS block in this statement always executes, whether an exception occurs or no exception occurs. The ALWAYS block is useful for executing code that must happen regardless of the successful execution of the protected statements. Examples including closing files that were opened prior to the TRY statement, freeing memory allocated on the heap, leaving critical sections, and so on.

If the ALWAYS block executes because an exception occurred, then the code will re-raise the exception immediately after the ALWAYS block finishes execution. An outer TRY..ENDTRY statement can handle the exception at that point.

If no exception occurs, then the ALWAYS block executes immediately after the last protected statement and once the ALWAYS block finishes, control resumes with the first statement after the ENDTRY.

Note that there is no way inside the ALWAYS block to determine if execution occurs because of an exception or because the protected statements completed execution without raising an exception. If you absolutely, positively, need to do something special if an exception occurs, then

insert a TRY..ANYEXCEPTION..ENDTRY statement around the protected statements or enclose the TRY..ALWAYS..ENDTRY statement inside a TRY .. EXCEPTION .. ANYEXCEPTION .. ENDTRY statement:

The following code executes the ANYEXCEPTION block prior to executing the code in the ALWAYS section:

```
try
  try
    << protected statements >>

    anyexception

        // Handle the statement before executing the ALWAYS clause
        raise( eax );

    endtry;

    always

        // Statements that always execute

endtry;
```

The following version executes the ALWAYS block first and then an ANYEXCEPTION block if there was an exception

```
try
  try
    << protected statements >>

    always

        // Statements that always execute

    endtry;

    anyexception

        // Statements that execute after ALWAYS bloc
        // if there was an exception

endtry;
```

14.4 Exception Handling in HLA:raise

The HLA RAISE statement generates an exception. The single parameter is an 8, 16, or 32-bit ordinal constant. Control is (ultimately) transferred to the first (most deeply nested) TRY..ENDTRY statement that has a corresponding exception handler (including ANYEXCEPTION).

If the program executes the RAISE statement within the protected block of a TRY..ENDTRY statement, then the enclosing TRY..ENDTRY gets first shot at handling the exception. If the RAISE statement occurs in an UNPROTECTED block, or in an exception handler (including ANYEXCEPTION), then the next higher level (nesting) TRY..ENDTRY statement will handle the exception. This allows *cascading* exceptions; that is, exceptions that the system handles in two or more exception handlers. Consider the following example:


```
try
  << Protected statements >>

  exception( someException )
  << Code to process this exception >>

  // The following re-raises this exception, allowing
  // an enclosing try..endtry statement to handle
  // this exception as well as this handler.

  raise( someException );

  << Additional, optional, exception handlers >>

endtry;
```

14.5 IF..THEN..ELSEIF..ELSE..ENDIF Statement in HLA

HLA provides a limited IF..THEN..ELSEIF..ELSE..ENDIF statement that can help make your programs easier to read. For the most part, HLA's if statement provides a convenient substitute for a CMP and a conditional branch instruction pair (or chain of such instructions when employing ELSEIF's).

The generic syntax for the HLA if statement is the following:

```
if( conditional_expression ) then

  << Statements to execute if expression is true >>

endif;

if( conditional_expression ) then

  << Statements to execute if expression is true >>

else

  << Statements to execute if expression is false >>

endif;

if( expr1 ) then

  << Statements to execute if expr1 is true >>

elseif( expr2 ) then

  << Statements to execute if expr1 is false
    and expr2 is true >>

endif;

if( expr1 ) then
```

```

    << Statements to execute if expr1 is true >>

elseif( expr2 ) then

    << Statements to execute if expr1 is false
        and expr2 is true >>

else

    << Statements to execute if both expr1 and
        expr2 are false >>

endif;

```

Note: HLA's if statement allows multiple ELSEIF clauses. All ELSEIF clauses must appear between IF clause and the ELSE clause (if present) or the ENDIF (if an ELSE clause is not present).

See the next section for a discussion of valid boolean expressions within the IF statement (this section appears first because the section on boolean expressions uses IF statements in its examples).

14.6 Boolean Expressions for High-Level Language Statements

The primary limitation of HLA's IF and other HLL statements has to do with the conditional expressions allowed in these statements. These expressions must take one of the following forms:

```

operand1 relop operand2

register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable

register
!register

memory
!memory

Flag

( boolean_expression )
!( boolean_expression )

boolean_expression && boolean_expression

boolean_expression || boolean_expression

```

For the first form, "operand1 *relop* operand2", *relop* is one of:

```
= or ==      (either one, both are equivalent)
<> or !=    (either one)
<
<=
>
>=
```

Operand1 and operand2 must be operands that would be legal for a "cmp(operand1, operand2);" instruction.

For the IF statement, HLA emits a CMP instruction with the two operands specified and an appropriate conditional jump instruction that skips over the statements following the "THEN" reserved word if the condition is false. For example, consider the following code:

```
if( al = 'a' ) then
    stdout.put( "Option 'a' was selected", nl );
endif;
```

Like the CMP instruction, the two operands cannot both be memory operands.

Unlike the conditional branch instructions, the six relational operators cannot differentiate between signed and unsigned comparisons (for example, HLA uses "<" for both signed and unsigned less than comparisons). Since HLA must emit different instructions for signed and unsigned comparisons, and the relational operators do not differentiate between the two, HLA must rely upon the types of the operands to determine which conditional jump instruction to emit.

By default, HLA emits unsigned conditional jump instructions (i.e., JA, JAE, JB, JBE, etc.). If either (or both) operands are signed values, HLA will emit signed conditional jump instructions (i.e., JG, JGE, JL, JLE, etc.) instead.

HLA considers the 80x86 registers to be *unsigned*. This can create some problems when using the HLA if statement. Consider the following code:

```
if( eax < 0 ) then
    << do something if eax is negative >>
endif;
```

Since neither operand is a signed value, HLA will emit the following code:

```
cmp( eax, 0 );
jnb SkipThenPart;
<< do something if eax is negative >>
SkipThenPart:
```

Unfortunately, it is never the case that the value in EAX is below zero (since zero is the minimum unsigned value), so the body of this if statement never executes. Clearly, the programmer intended to use a signed comparison here. The solution is to ensure that at least one operand is signed. However, as this example demonstrates, what happens when both operands are intrinsically unsigned?

The solution is to use coercion to tell HLA that one of the operands is a signed value. In general, it is always possible to coerce a register so that HLA treats it as a signed, rather than unsigned, value. The IF statement above could be rewritten (correctly) as

```

if( (type int32 eax) < 0 ) then
    << do something if eax is negative >>
endif;

```

HLA will emit the JNL instruction (rather than JNB) in this example. Note that if either operand is signed, HLA will emit a signed condition jump instruction. Therefore, it is not necessary to coerce both unsigned operands in this example.

The second form of a conditional expression that the IF statement accepts is a register or memory operand followed by "in" and then two constants separated by the ".." operator, e.g.,

```
if( al in 0..10 ) then ...
```

This code checks to see if the first operand is in the range specified by the two constants. The constant value to the left of the ".." must be less than the constant to the right for this expression to make any sense. The result is true if the operand is within the specified range. For this instruction, HLA emits a pair of compare and conditional jump instructions to test the operand to see if it is in the specified range.

HLA also allows a exclusive range test specified by an expression of the form:

```
if( al not in 0..10 ) then ...
```

In this case, the expression is true if the value in AL is outside the range 0..10.

In addition to integer ranges, HLA also lets you use the IN operator with CSET constants and variables. The generic form is one of the following:

```

reg8 in CSetConst
reg8 not in CSetConst
reg8 in CSetVariable
reg8 not in CSetVariable

```

For example, a statement of the form "if(al in {'a'..'z'}) then ..." checks to see if the character in the AL register is a lower case alphabetic character. Similarly,

```
if( al not in {'a'..'z', 'A'..'Z'}) then...
```

checks to see if AL is not an alphabetic character.

The fifth form of a conditional expression that the IF statement accepts is a single register name (eight, sixteen, or thirty-two bits). The IF statement will test the specified register to see if it is zero (false) or non-zero (true) and branches accordingly. If you specify the not operator ("!") before the register, HLA reverses the sense of this test.

The sixth form of a conditional expression that the IF statement accepts is a single memory location. The type of the memory location must be boolean, byte, word, or dword. HLA will emit code that compares the specified memory location against zero (false) and generate an appropriate branch depending upon the value in the memory location. If you put the not operator ("!") before the variable, HLA reverses the sense of the test.

The seventh form of a conditional expression that the IF statement accepts is a Flags register bit or other condition code combination handled by the 80x86 conditional jump instructions. The following reserved words are acceptable as IF statement expressions:

```

@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne

```

These items emit an appropriate jump (of the opposite sense) around the THEN portion of the IF statement if the condition is false.

If you supply any legal boolean expression in parenthesis, HLA simply uses the value of the internal expression for the value of the whole expression. This allows you to override default precedence for the AND, OR, and ! operators.

The !(boolean_expression) evaluates the expression and does just the opposite. That is, if the interior expression is false, then !(boolean_expression) is true and vice versa. This is mainly useful with conjunction and disjunction since all of the other interesting terms already allow the not operator in front of them. Note that in general, the "!" operator must precede some parentheses. You cannot say "! AX < BX", for example.

Originally, HLA did not include support for the conjunction (&&) and disjunction (||) operators. This was explicitly left out of the design so that beginning students would be forced to rethink their logical operations in assembly language. Unfortunately, it was so inconvenient not to have these operators that they were eventually added. So a compromise was made: these operators were added to HLA but "The Art of Assembly Language Programming/Win32 Edition" doesn't bother to mention them until an advanced chapter on control structures.

The conjunction and disjunction operators are the operators && and ||. They expect two valid HLA boolean expressions around the operator, e.g.,

```
eax < 5 && ebx <> ecx
```

Since the above forms a valid boolean expression, it, too, may appear on either side of the && or | operator, e.g.,

```
eax < 5 && ebx <> ecx || !dl
```

HLA gives && higher precedence than ||. Both operators are left-associative so if multiple operators appear within the same expression, they are evaluated from left to right if the operators have the same precedence. Note that you can use parentheses to override HLA's default precedence.

One wrinkle with the addition of && and || is that you need to be careful when using the flags in a boolean expression. For example, "eax < ecx && @nz" hides the fact that HLA emits a compare instruction that affects the Z flag. Hence, the "@nz" adds nothing to this expression since EAX must not equal ECX if eax<ecx. So take care when using && and ||.

HLA uses short-circuit evaluation when evaluating expressions containing the conjunction and disjunction operators. For the && operator, this means that the resulting code will not compute the right-hand expression if the left-hand expression evaluates false. Similarly, the code will not compute the right-hand expression of the || operator if the left-hand expression evaluates true.

Note that the evaluation of complex boolean expressions involving the !(--), &&, and || operators does not change any register or memory values. HLA strictly uses flow control to implement these operations.

Note that the "&" and "|" operators are for compile-time only expression while the "&&" and "||" operators are for run-time boolean expressions. These two groups of operators are not synonyms and you cannot use them interchangeably.

If you would prefer to use a less abstract scheme to evaluate boolean expressions, one that lets you see the low-level machine instructions, HLA provides a solution that allows you to write code to evaluate complex boolean expressions within the HLL statements using low-level instructions. Consider the following syntax:

```
if
  (#{
    <<arbitrary HLA statements >>
  }#) then

    << "True" section >>

else //or elseif...

    << "False" section >>
```

```
endif;
```

The "#{" and "}" brackets tell HLA that an arbitrary set of HLA statements will appear between the braces. HLA will *not* emit any code for the IF expression. Instead, it is the programmer's responsibility to provide the appropriate test code within the "#{---}#" section. Within the sequence, HLA allows the use of the boolean constants "true" and "false" as targets of conditional jump instructions. Jumping to the "true" label transfers control to the true section (i.e., the code after the "THEN" reserved word). Jumping to the "false" label transfers control to the false section. Consider the following code that checks to see if the character in AL is in the range "a".."z":

```
if
  (#{
    cmp( al, 'a' );
    jb false;
    cmp( al, 'z' );
    ja false;
  }) then

    << code to execute if AL in {'a'..'z'} goes here >>

endif;
```

With the inclusion of the "#{---}#" operand, the IF statement becomes much more powerful, allowing you to test any condition possible in assembly language. Of course, the "#{---}#" expression is legal in the ELSEIF expression as well as the IF expression.

It would be a good idea for you to write some code using the HLA if statement and study the MASM code produced by HLA for these IF statements. By becoming familiar with the code that HLA generates for the IF statement, you will have a better idea about when it is appropriate to use the if statement versus standard assembly language statements.

14.7 WHILE..WELSE..ENDWHILE Statement in HLA

The while..endwhile statement allows the following syntax:

```
while( boolean_expression ) do

    << while loop body>>

endwhile;

while( boolean_expression ) do

    << while loop body>>
else

    << Code to execute when expression is false >>

endwhile;

while(#{ HLA_statements }#) do

    << while loop body>>
```

```

endwhile;

while(#{ HLA_statements }#) do

    << while loop body>>

welse

    << Code to execute when expression is false >>

endwhile;

```

The WHILE statement allows the same boolean expressions as the HLA IF statement. Like the HLA IF statement, HLA allows you to use the boolean constants "true" and "false" as labels in the #{...}# form of the WHILE statement above. Jumping to the true label executes the body of the while loop, jumping to the false label exits the while loop.

For the "while(expr) do" forms, HLA moves the test for loop termination to the bottom of the loop and emits a jump at the top of the loop to transfer control to the termination test. For the "while(#{stmts}#)" form, HLA compiles the termination test at the top of the emitted code for the loop. Therefore, the standard WHILE loop may be slightly more efficient (in the typical case) than the hybrid form.

The HLA while loop supports an optional "welse" (while-else) section. The while loop will execute the code in this section only when then the expression evaluates false. Note that if you exit the loop via a "break" or "breakif" statement the welse section does not execute. This provides logic that is sometimes useful when you want to do something different depending upon whether you exit the loop via the expression going false or by a break statement.

14.8 REPEAT..UNTIL Statement in HLA

HLA's REPEAT..UNTIL statement uses the following syntax:

```

repeat

    << statements to execute repeatedly >>

until( boolean_expression );

repeat

    << statements to execute repeatedly >>

until(#{ HLA_statements }#);

```

For those unfamiliar with REPEAT..UNTIL, the body of the loop always executes at least once with the test for loop termination occurring at the bottom of the loop. The REPEAT..UNTIL loop (unlike C/C++'s do..while statement) terminates loop execution when the expression is true (that is, REPEAT..UNTIL repeats while the expression is false).

As you can see, the syntax for this is very similar to the WHILE loop. About the only major difference is the fact that jump to the "true" label in the #{---}# sequence exits the loop while jumping to the "false" label in the #{---}# sequence transfers control back to the top of the loop.

14.9 The FOR..ENDFOR Statement in HLA

The HLA for..endfor statement is very similar to the C/C++ for loop. The FOR clause consists of three components:

```
for( initialize_stmt; if_boolean_expression; increment_statement ) do
```

The *initialize_statement* component is a single machine instruction. This instruction typically initializes a loop control variable. HLA emits this statement before the loop body so that it executes only once, before the test for loop termination.

The *if_boolean_expression* component is a simple boolean expression (same syntax as for the IF statement). This expression determines whether the loop body executes. Note that the FOR statement tests for loop termination before executing the body of the loop.

The *increment_statement* component is a single machine instruction that HLA emits at the bottom of the loop, just before jumping back to the top of the loop. This instruction is typically used to modify the loop control variable.

The syntax for the HLA for statement is the following:

```
for( initStmt; BoolExpr; incStmt ) do
    << loop body >>
endfor;
-or-
for( initStmt; BoolExpr; incStmt ) do
    << loop body >>
false
    << statements to execute when BoolExpr evaluates false >>
endfor;
```

Semantically, this statement is identical to the following while loop:

```
initStmt;
while( BoolExpr ) do
    << loop body >>
    incStmt;
endwhile;
-or-
initStmt;
while( BoolExpr ) do
    << loop body >>
    incStmt;
welse
    << statements to execute when BoolExpr evaluates false >>
endwhile;
```

Note that HLA does not include a form of the FOR loop that lets you bury a sequence of statements inside the boolean expression. Use the WHILE loop if you want to do that. If this is inconvenient, you can always create your own version of the FOR loop using HLA's macro facilities.

The `FELSE` section in the `FOR.FELSE.ENDFOR` loop executes when the boolean expression evaluates false. Note that the `FELSE` section does not execute if you break out of the `FOR` loop with a `BREAK` or `BREAKIF` statement. You can use this fact to do different logic depending on whether the code exits the loop via the boolean expression going false or via some sort of `BREAK`.

14.10 The `FOREVER..ENDFOR` Statement in HLA

The forever statement creates an infinite loop. Its syntax is

```
forever

    << Statements to execute repeatedly >>

endfor
```

This HLA statement simply emits a single `JMP` instruction that unconditionally transfers control from the `ENDFOR` clause back up to the beginning of the loop.

In addition to creating infinite loops, the `FOREVER..ENDFOR` loop is very useful for creating loops that test for loop termination somewhere in the middle of the loop's body. For more details, see the `BREAK` and `BREAKIF` statements, next.

14.11 The `BREAK` and `BREAKIF` Statements in HLA

The `BREAK` and `BREAKIF` statements allow you to exit a loop at some point other than the normal test for loop termination. These two statements allow the following syntax:

```
break;
breakif( boolean_expression );
breakif(#{ stmts }#);
```

There are two very important things to note about these statements. First, unlike many HLA machine instructions, you do not follow the `BREAK` statement with a pair of empty parentheses. The 80x86 machine instructions behave like compile-time functions, so it made sense to require empty parentheses after those instructions. The HLA HLL statements do not behave like compile-time functions; the lack of parentheses after `BREAK` (and other HLL statements, e.g., `ELSE`) makes sense here if you think about it for a moment.

The second thing to note is that the `BREAK` and `BREAKIF` statements are legal only inside `WHILE`, `FOREACH`, `FOREVER`, and `REPEAT` loops. HLA does not recognize loops you've coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a `BREAK` function for your own loops). Note that the `FOREACH` loop pushes data on the stack that the `BREAK` statement is unaware of. Therefore, if you break out of a `FOREACH` loop, garbage will be left on the stack. The HLA `BREAK` statement will issue a warning if this occurs. It is your responsibility to clean up the stack upon exiting a `FOREACH` loop if you break out of it.

14.12 The `CONTINUE` and `CONTINUEIF` Statements in HLA

The `continue` and `continueif` statements allow you to restart a loop. These two statements allow the following syntax:

```
continue;
continueif( boolean_expression );
continueif(#{ stmts }#);
```

There are two very important things to note about these statements. First, unlike many HLA machine instructions, you do not follow the `CONTINUE` statement with a pair of empty

parentheses. The 80x86 machine instructions behave like compile-time functions, so it made sense to require empty parentheses after those instructions. The HLA HLL statements do not behave like compile-time functions; the lack of parentheses after `continue` (and other HLL statements, e.g., `else`) makes sense here if you think about it for a moment.

The `CONTINUE` and `CONTINUEIF` statements are legal only inside `WHILE`, `FOREACH`, `FOREVER`, and `REPEAT` loops. HLA does not recognize loops you've coded yourself using discrete assembly language instructions (of course, you can probably write a macro to provide a `CONTINUE` function for your own loops).

For the `WHILE` and `REPEAT` statements, the `CONTINUE` and `CONTINUEIF` statements transfer control to the test for loop termination. For the `FOREVER` loop, the `CONTINUE` and `CONTINUEIF` statements transfer control to the first statement in the loop. For the `FOREACH` loop, `CONTINUE` and `CONTINUEIF` transfer control to the bottom of the loop (i.e., forces a return from the `yield()` call).

14.13 The `BEGIN..END`, `EXIT`, and `EXITIF` Statements in HLA

The `BEGIN..END` statement block provides a structured goto statement for HLA. The `BEGIN` and `END` clauses surround a group of statements; the `EXIT` and `EXITIF` statements allow you to exit such a block of statements in much the same way that the `BREAK` and `BREAKIF` statements allow you to exit a loop. Unlike `BREAK` and `BREAKIF`, which can only exit the loop that immediately contains the `BREAK` or `BREAKIF`, the exit statements allow you to specify a `BEGIN` label so you can exit several nested contexts at once. The syntax for the `BEGIN..END`, `EXIT`, and `EXITIF` statements is as follows:

```
begin contextLabel ;

    << statements within the specified context >>

end contextLabel;

exit contextLabel;
exitif( boolean_expression ) contextLabel;
exitif(#{ stmts }#) contextLabel;
```

The `BEGIN..END` clauses do not generate any machine code (although `END` does emit a label to the assembly output file). The `EXIT` statement simply emits a `JMP` to the first instruction following the `END` clause. The `EXITIF` statement emits a compare and a conditional jump to the statement following the specified end.

If you break out of a `FOREACH` loop using the `EXIT` or `EXITIF` statements, there will be garbage left on the stack. It is your responsibility to be aware of this situation (i.e., HLA doesn't warn you about it) and clean up the stack, if necessary.

You can nest `BEGIN..END` blocks and `EXIT` out of any enclosing `BEGIN..END` block at any time. The `BEGIN` label provides this capability. Consider the following example:

```
program ContextDemo;

#include( "stdio.hhf" );

static
    i:int32;

begin ContextDemo;

    stdout.put( "Enter an integer:" );
    stdin.get( i );

    begin c1;
```

```

begin c2;

    stdout.put( "Inside c2" nl );
    exitif( i < 0 ) c1;

end c2;
stdout.put( "Inside c1" nl );
exitif( i = 0 ) c1;
stdout.put( "Still inside c1" nl );

end c1;
stdout.put( "Outside of c1" nl );

end ContextDemo;

```

The EXIT and EXITIF statements let you exit any BEGIN..END block; including those associated with a program unit such as a procedure, iterator, method, or even the main program. Consider the following (unusable) program:

```

program mainPgm;

    procedure LexLevel1;

        procedure LexLevel2;
        begin LexLevel2;

            exit LexLevel2;    // Returns from this procedure.
            exit LexLevel1;    // Returns from this procedure and
                               // and the LexLevel1 procedure
                               // (including cleaning up the stack).
            exit mainPgm;      // Terminates the main program.

        end LexLevel2;

    begin LexLevel1;
        .
        .
        .
    end LexLevel1;

begin mainPgm;
    .
    .
    .
end mainPgm;

```

Note: You may only exit from procedures that have a display and all nested procedures from the procedure you wish to exit from through to the EXIT statement itself must have displays. In the example above, both LexLevel1 and LexLevel2 must have displays if you wish to exit from the LexLevel1 procedure from inside LexLevel2. By default, HLA emits code to build the display unless you use the "@nodisplay" procedure option.

Note that to exit from the current procedure, you must not have specified the "@noframe" procedure option. This applies only to the current procedure. You may exit from nesting (lower lex level) procedures as long as the display has been built.

14.14 The SWITCH/CASE/DEFAULT/ENDSWITCH Statement in HLA

As of HLA v1.102, a multi-way switch statement is available in the HLA language (prior to HLA v1.102, the switch statement was handled by a macro provided in the HLA Standard Library). This statement uses syntax similar to the following:

```
switch( reg32 )  
  
    case( constant_list )  
  
        <statements>  
  
    << any number of additional case clauses >>  
  
    default// This is optional  
  
        <statements>  
  
endswitch;
```

The case clause argument list is either a single ordinal constant, or a list of ordinal constants separated by commas. The following is an example of a legal switch statement with multiple case clauses:

```
switch( eax )  
  
    case( 0 )  
  
        mov( 1, eax );  
  
    case( 1, 2 )  
  
        mov( 2, eax );  
  
    case( 5 )  
  
        add( 4, eax );  
  
endswitch;
```

The *switch* statement, like it's HLL counterpart, transfers control to the statements following the *case* clause containing the value held in the 32-bit register passed into the *switch* statement.

The *case* constant values in a single *case* statement must all be unique. HLA will report an error if two *cases* contain the same constant value.

During the execution of the *switch* statement, if the value in the 32-bit register passed as an argument to the *switch* statement is not present any any of the *case* clauses, then control transfers to the statements associated with the *default* clause (if one is present) or to the first statement following the *endswitch* class if there is no *default* section present.

In general, HLA compiles the *switch* statement into a jump table and an indirect *jmp* instruction that transfers control to the code associated with the specified case. However, in a couple of special cases HLA will not compile a switch into an indirect jump instruction. To understand when this occurs, there are a couple of terms you'll need to understand.

Jump tables created for *switch* statements will have one entry for every ordinal value between the smallest *case* value and the largest *case* value in the table. The difference between the largest

and smallest case values (plus one) is called the *spread*. This means that a jump table's size in bytes will be four times the spread. Note that the spread value is independent of the number of cases. Consider the following *switch* statement fragments:

```
switch( eax )
    case( 1 )
        << code to execute if EAX = 1 >>
    case( 10 )
        << code to execute if EAX = 10 >>
endswitch;
```

The jump table associated with this switch entry will have ten entries, not two. This is because the spread is 10 for this switch statement. Consider the following example:

```
switch( eax )
    case( 1 )
        << code to execute if EAX = 1 >>
    case( 3 )
        << code to execute if EAX = 3 >>
    case( 6 )
        << code to execute if EAX = 6 >>
    case( 10 )
        << code to execute if EAX = 10 >>
endswitch;
```

In this examples the spread is still 10 and the jump table will have the same number of entries (10) as the previous example. This is true even though this latter example has twice as many cases as the earlier example.

The case clause lets you specify multiple values in a comma-separated list. Consider the following example:

```
switch( eax )
    case( 1 )
        << code to execute if EAX = 1 >>
    case( 3, 6, 12 )
        << code to execute if EAX = 3, 6, or 12 >>
    case( 10 )
```

```

    << code to execute if EAX = 10 >>

endswitch;

```

It is important to realize that this *switch* statement has five cases, not three. It just happens that three of the cases (3, 6, and 12) share the same set of instructions to execute. Also note that the spread is 12 in this example as the minimum *case* value is 1 and the largest is 12. Note that the default *case* does not count as a *case* for the purposes of counting the number of *case* values. The default case simply provides a sequence of instructions to execute for all the “holes” in the spread of case values (as well as all values below and greater than the minimum and maximum case values).

Because the jump table will have one entry for each integer value between the smallest and largest *case* values, you can easily generate a huge table with a very simple *switch* statement. Consider the following example:

```

switch( eax )

    case( 1 )

        << code to execute if EAX = 1 >>

    case( 1000 )

        << code to execute if EAX = 1000 >>

endswitch;

```

Even though this example has only two cases, the jump table will contain 1,000 entries (and be 4,000 bytes long). A set of widely spaced *case* values produces a sparse jump table (that is, only a few of the entries in the jump table contain pointers to sections of code associated with the cases, most entries contain a pointer to the *default* case (or the address of the first statement following the *endswitch* if there isn't a default section).

To improve efficiency and reduce the space consumed by large, sparse, jump tables, HLA specially handles a couple of situations. First of all, if the number of cases is three or less, HLA will not emit a jump table. Instead, it will emit a sequence of *CMP* and *JNE* instructions to test the three or fewer case values. Second, if the spread is 256 or greater but there are 32 or fewer cases, then HLA will emit a sequence of *CMP* and *JNE* instructions to implement the *switch* statement. In all other situations, HLA will emit a jump table implementation of the *switch*.

If the spread is 16384 or greater (this is an implementation-dependent constant and may change in the future), HLA will generate an error and refuse to compile the *switch* statement. If you really want to generate a *switch* statement whose jump table consumes 64K (or more) of data, you will have to implement the statement manually (or modify the *switch* macro in the “switch.hhf” header file).

If the spread is 4096 or greater but less than 16384, HLA will generate the code but issue a warning telling you that the jump table is going to be very large. If the spread is 16 times (or more) the number of cases, HLA will emit a warning telling you that the jump table is going to be very sparse.

All the case values in a particular *switch* statement must be unique. If there are any duplicate case values in a particular *switch* statement HLA will issue an error message.

14.15 The JT and JF Medium Level Instructions in HLA

The *JT* (jump if true) and *JF* (jump if false) instructions are a cross between the 80x86 conditional jump instruction and the HLA *IF* statement. These two instructions use the following syntax:

```
JT ( booleanExpression ) targetLabel;
JF ( booleanExpression ) targetLabel;
```

The *booleanExpression* component can be any legal HLA boolean expression that you'd use in an IF, WHILE, REPEAT..UNTIL, or other HLA HLL statement. The HLA compiler emits code that will transfer control to the specified target label in your program if the condition is true.

These instructions are primarily intended for use in macros when creating your own HLL control statements. For a discussion of macros and creating your own control structures, see the HLA documentation on the compile-time language.

14.16 Iterators and the HLA Foreach Loop

HLA provides a very powerful user-defined looping control structure, the FOREACH..ENDFOR loop. The FOREACH loop uses the following syntax:

```
foreach iteratorProc( parameters ) do
  << foreach loop body >>
endfor;
```

The *iteratorProc*(*parameters*) component is a call to a special kind of procedure known as an iterator¹. Iterators have the special property that they return one of two states, success or failure. If an iterator returns success, it generally also returns a function result. If an iterator returns success, the foreach loop will execute the loop body and reenter the iterator (more on that later) at the top of the loop. If an iterator returns failure, then the loop terminates.

If you've never used true iterators before, you may be thinking "big deal, an iterator is simply a function that returns a boolean value." This, however, isn't entirely true. An iterator behaves like a value returning function when it succeeds, it behaves like a procedure when it fails. The success or failure state of the iterator call is *not* the return value. To understand the difference, consider the syntax for an iterator:

```
iterator iteratorName <<( optional_parameters )>>;
<< procedure options >>
<< local declarations >>
begin iteratorName;

  << iterator statements >>

end iteratorName;
```

Other than the use of the "ITERATOR" keyword rather than "PROCEDURE," this declaration looks just like a procedure or method declaration. However, there are some crucial differences. First of all, HLA emits different code for building iterator activation records than it does for procedures and methods. Furthermore, whenever you declare an iterator, HLA automatically creates a special thunk variable named "yield". Also, HLA will not let you call an iterator directly by specifying the iterator's name as an HLA statement (although you can still use the CALL instruction to call an iterator procedure, though you'd better have set the stack up properly before doing so).

If an iterator returns via an EXIT(*iteratorname*) or RET() statement, or returns by "falling off the end of the function" (i.e., executing the "end" clause), then the iterator returns failure to the calling FOREACH loop (hence, the loop will terminate). To return success, and return a value to the body of the FOREACH loop, you must invoke the "yield" thunk. Yield doesn't actually return to the FOREACH loop, instead, it calls the body of the FOREACH loop and at the bottom of

1. HLA's iterators are based on the similar control structure from the CLU language. CLU's iterators are considerably more powerful than the misnamed "iterators" found in the C/C++ language/library (which, technically, should be called "cursors" not iterators).

the FOREACH loop HLA emits a return instruction that transfers control back into the iterator (to the first statement following the `yield`). This may seem counter-intuitive, but it has some important ramifications. First of all, an iterator maintains its context until it fails. This means that local variables maintain their values across the `yield` calls. Likewise, when a FOREACH loop reenters an iterator, it picks up immediately after the `yield`, it does not pass new parameters and begin execution at the top of the iterator code.

Consider the following typical iterator code:

```

program iteratorDemo;

#include( "stdio.hhf" );

iterator range( start:int32; stop:int32 ); @nodisplay;
begin range;

    forever

        mov( start, eax );
        breakif( eax > stop );
        yield();
        inc( start );

    endfor;

end range;

static
    i:int32;

begin iteratorDemo;

    foreach range( 1, 10 ) do

        stdout.put( "eax = ", eax, nl );

    endfor;

end iteratorDemo;

```

This example demonstrates how to create a standard "for" loop like those found in Pascal or C++². The `range` iterator is passed two parameters, a starting value and an ending value. It returns a sequence of values between the starting and ending values (respectively) and fails once the return value would exceed the ending value. The FOREACH loop in this example prints the values one through ten to the display.

Warning: because the iterator's activation is left on the stack while executing a FOREACH loop, you should take care when breaking out of a FOREACH loop using `BREAK`, `BREAKIF`, `EXIT`, `EXITIF`, or some sort of jump. Cavalierly jumping out of a loop in this fashion leaves the iterator's activation record on the stack. You will need to clean this up manually if you exit an iterator in this fashion. Since HLA cannot determine the myriad of ways one could jump out of a FOREACH loop body, it is up to you to make sure you don't do this (or that you handle the garbage on the stack in an appropriate way).

Keep in mind that the body of a FOREACH loop is actually a procedure your program calls when it encounters the `yield` statement³. Therefore, any registers whose values you change will be changed when control returns to the code following the `yield`. If you need to preserve any

2. Mind you, this is not a very efficient implementation of a standard for loop.

registers across a `yield`, either push and pop them at the beginning of the `FOREACH` loop body or place the `PUSH` and `POP` instructions around the `yield`.

3. Technically, `yield` is a variable of type `thunk`, not a statement. However, this discussion is somewhat clearer if we think of `yield` as a statement rather than a variable.

15 HLA Units and External Compilation

15.1 HLA Units and External Compilation

This section discusses how to create separately compilable modules in HLA and how you can link HLA code with code written in other languages.

15.2 External Declarations

HLA provides two features to support separate compilation: units and external objects. HLA uses a very general scheme, similar to C++ to communicate linkage information between object modules. This scheme lets HLA programmers link to their HLA programs code written in HLA, non-HLA assembly code (e.g., MASM), and even code written in other high level languages (HLLs). Conversely, the HLA program can also write modules to be linked with programs written in this other languages (as well as HLA).

Writing separate modules is quite similar to writing a single HLA program. The first thing to note is that an executable can have only one main program. When writing HLA programs, the **program** reserved word tells HLA that you are writing a module that contains a main program. When writing other modules, you must use a **unit** rather than a **program** so as not to generate an extra main procedure. If you wish to write a library module that contains only procedures and no main program, you would use an HLA unit. Units have a syntax that is nearly identical to programs, there just isn't a **begin** associated with the **unit**, e.g.,

```
unit UnitName;  
  
    << Declarations >>  
  
end UnitName;
```

Since a **unit** does not contain a main program, it cannot compile into a stand-alone program; therefore, you should always compile units with the "-c" command line option to avoid running the linker on the unit code (which will always produce a link error)¹.

HLA uses the **external** keyword to communicate names between modules in a compilation group. If a symbol is defined to be external, HLA assumes that the symbol is declared in a separate module and leaves it up to the linker to resolve the symbol's address.

Only two types of symbols may be external: subroutines (procedures, methods, and iterators) and static variables². Variables declared in the **var** section cannot be external because the linker cannot statically resolve their run-time address. Constants declared in the **const** or **val** sections cannot be external, however this is not a limitation because most programmers place public constants in header files and include them in the source files that require them.

Recall the syntax for an original-style procedure declaration presented in the chapter on procedure declarations:

```
procedure identifier ( optional_parameter_list ); procedure_options  
    declarations  
begin identifier;  
    statements
```

1. Actually, the HLA.EXE program allows you to specify several ".HLA" files on the command line. The command line option "-c" is only necessary if none of the files on the command line contain a main program.

2. For the purposes of this discussion, variables appearing in the READONLY, and STORAGE sections are treated as static variables along with variables declared in the STATIC section.

```
end identifier;
```

There are two additional forms to consider:

```
procedure identifier ( optional_parameter_list );
    options
    external;

procedure identifier ( optional_parameter_list );
    options
    external("extname");
```

These two forms tell the HLA compiler that it is okay to call the specified procedure, but the procedure itself may not otherwise appear in the current source file. It is the responsibility of the linker to ensure that the specified external procedures actually appear within the object modules the linker is combining.

The first form above is generally used when the external procedure is an HLA procedure that appears in a different source module. HLA assumes that the external name is the same name as the procedure identifier.

The second form above is generally used when calling code written in a language other than HLA¹. This form lets you explicitly state (via the string constant "extname") the name of the external procedure. This is especially important when calling procedures whose names contain characters that are not "HLA-Friendly." For example, many Windows API calls have at signs ("@") in their names; to call such routines you would use the second form of the external declaration above supplying the Windows API compatible name as the parameter to the **external** reserved word.

It is legal to declare an external procedure in the same source file that the procedure's actual code appears. However, the external declaration must appear *before* the actual declaration or HLA will generate an error. Whenever an external declaration appears in the same source file as the actual procedure code, HLA emits code to ensure that the procedure's name is *public*. Therefore, the external declaration *must* appear in the same file as the procedure's code if you wish the linker to be able to resolve the procedure's address at link time. This external declaration serves the same purpose as the "public" directive in other assemblers (e.g., MASM). Note that, unlike C/C++, procedure names are not automatically public. An external declaration must appear in the same file as the procedure code to make the symbol public.

Also, note above that the only options an external procedure declaration supports are the **@returns**, **@pascal**, **@cdecl**, and **@stdcall** options. You cannot use the **@align**, **@noalignstack**, **@noframe** or **@nodisplay** options in an external declaration. Conversely, if an **external** (or **forward**, for that matter) declaration appears in a source file, the corresponding procedure code may only contain the **@align**, **@noalignstack**, **@noframe**, and/or **@nodisplay** options. The **@returns**, **@pascal**, **@cdecl**, and **@stdcall** options are not legal in a procedure declaration if a corresponding **external** (or **forward**) declaration is present in the source code.

Note: External procedures are only legal at lex level one. You cannot declare an external procedure that is embedded inside another procedure.

In addition to procedures, HLA also lets you declare **external** variables. You may reference such variables in different source modules. The declaration of an external variable is very similar to the declaration of an external procedure: you follow the variable's name with the external clause. If an optional string parameter is not present, HLA uses the variable's name as it's external name. If you need to specify a specific name, to avoid conflicts with other languages or to contain characters illegal in an HLA identifier, then provide a string with the identifier you need.

Note that HLA does not allow the **external** keyword after every static declaration. Instead, only the following variable declarations allow the **external** keyword:

```
name: procedure optional_parameters; @external;
name: pointer to typename; @external;
name: typename; @external;
```

1. Or when the HLA procedure name is a MASM reserved word.

```
name: typename [ dimensions ]; @external;
```

In particular, note that static variable declarations with initializers cannot be external. Also note that **enum**, **record**, and **union** variables (those variables you directly create as **enum**, **record**, or **union**) may not be external. This is not a serious limitation, however, since you can declare a named type in the **type** section and use the third form above to create an external object of the desired type (this is also how you would declare **external** class variables).

Like the C/C++ language, you normally put all your external declarations in a header file and include that header file using the **#include** directive in each of the source files that reference the external symbols. This eases program maintenance by having to change only a single definition in an include file rather than multiple definitions across different source files (if not using include files). See the HLA Standard Library code for some good examples of using HLA header files.

By convention, HLA header files that contain external declarations always have an ".HHF" suffix (HLA Header File). To help make your programs easy to read by others, you should always use this same suffix for your HLA header files.

15.3 HLA Naming Conventions and Other Languages

If you wish to link together code written in a different language with code written in HLA, you must be aware of the differences in naming conventions between the two languages.

With respect to names, keep in mind that HLA is a case-neutral language. To the outside world, this means that HLA is case sensitive. Therefore, all public names that HLA exports are case sensitive. If you are using a case insensitive language like Pascal or Delphi, you should check with your compiler vendor to determine how the language emits public names (usually, case insensitive languages convert all public symbols to all upper case or all lower case). Some languages, e.g., MASM, let you choose whether public symbols are case sensitive or case insensitive; for such languages, you should select case sensitivity as the default and spell your names the same (with respect to case) between the HLA code and the other language.

In some cases, it might not be possible to match an HLA identifier with a public or external identifier in another language. One possible reason for this problem is that HLA only allows alphanumeric characters and underscores in identifiers; some other languages (e.g., MASM) allow other characters in their names while other languages (e.g., C++) often "mangle" their names by adding additional characters that are normally illegal within identifiers (e.g., the at sign, "@").

The HLA **external** directive provides an option that lets you use a standard HLA identifier within your program, but utilize a completely different identifier as the public symbol. The standard HLA identifier restrictions do not apply to the external name¹. This variant of the external directive takes the following forms:

External procedure declaration:

```
procedure ProcName; @external( "ExtProcName" );
```

External variable declaration:

```
varName: SomeType; @external( "ExtVarName" );
```

Within the confines of the HLA program, you would use the HLA identifiers *ProcName* and *varName*. To the outside world, however, you would use the names *ExtProcName* and *ExtVarName* to reference these objects.

Since the **external** parameter is a string constant rather than an HLA identifier, you can use characters that would otherwise be illegal in an HLA identifier. For example, Microsoft's Visual C++ language and Windows often insert the "@" symbol into identifiers. Normally, this character is illegal in (user-defined) HLA symbols. You may, however, give an identifier a legal HLA name and then specify the VC++ compatible name within the string constant. For example, here is a typical procedure declaration found in the HLA standard library "fileio.hla" source file:

```
procedure WriteFile
```

1. However, since HLA emits the identifier to the MASM assembly language output file, the external identifier must be MASM compatible.

```
(
    overlapped:    dword;
    var bytesWritten:  dword;
    len:           dword;
    var buffer:     byte;
    Handle:        dword
);
@external( "_WriteFile@20" );
```

(The "@20" suffix is a Win32 convention that indicates that there are 20 bytes of parameter data in this external function.)

As noted above, many languages "mangle" their external names for one reason or another. In addition to the "@20" suffix in the previous example, you will also note that VC++ added a leading underscore to the name (this procedure calls the Win32 API *WriteFile* function). Once again, this name mangling is a function of the particular compiler being used. Since Windows itself is written in VC++, Win32 API calls follow the VC++ standards for name mangling.

In addition to giving you the ability to conform external names as needed by external languages, the string parameter of the **external** directive will let you change the name for more mundane reasons. For example, if you really don't like the external name, perhaps it is not descriptive of the operation, you can use the string parameter feature of the external directive to allow the use of a different, perhaps more descriptive, name in your HLA code.

Some languages, for example C++, provide *function* overloading. This means that a program can use the same name to reference two completely different procedures in the code. Within the object file, however, all names must be unique. Once again, the compiler's name mangling facilities come into play to generate unique names. How a particular name is mangled is extremely compiler sensitive (e.g., Borland's C++ mangles names differently than Microsoft's Visual C++, even when compiling the same exact C++ program). When deciding on the name with which to reference an external procedure, you may need to consult your compiler documentation or be willing to experiment around a bit.

15.4 HLA Calling Conventions and Other Languages

Of course, HLA is an assembly language, so it is possible via the **push** and **call** instructions to mimic any calling sequence used by any language that allows the call of external assembly language code (which covers almost all languages). However, when using the HLA high level language features, in particular, HLA procedure declarations and calls, there are some details you must be aware of in order to successfully call code written in other languages or have those other languages call your code.

By default, HLA assumes that all parameters are pushed on the stack in a left-to-right order as the parameters appear in the formal parameter list. Some languages, like Pascal and Delphi, use this same calling mechanism. A few languages, most notably C/C++, push their parameters in the right-to-left order. If the language expects the parameters to be in the reverse order (right-to-left), a simple solution is to use the **@cdecl** or **@stdcall** procedure options to specify the calling convention.

Many languages, like HLA, Pascal, and Delphi, make it the procedure's responsibility to clear parameters from the stack when the procedure returns to the caller. Some languages, like C/C++ make it the caller's responsibility to clear parameters from the stack after the procedure returns to the caller. Procedures you declare with the **@pascal** and **@stdcall** procedure options automatically remove their parameter data from the stack when they return. Procedures you declare with the **@cdecl** option leave it up to the caller to remove the parameter data from the stack. Note that when using the HLA high-level procedure calling syntax, HLA automatically pushes the parameters on the stack in the correct order ("correct" as defined by the procedure's calling convention).

HLA procedures do not support a variable number of parameters in a parameter list. If you need this facility (e.g., to call a C/C++ function) then you will need to manually push the parameters on the stack yourself prior to calling the function. Procedures that have a variable number of parameters almost always using the **@cdecl** calling convention; since only the caller knows how much parameter data to remove from the stack, the procedure generally cannot remove the parameter data (as the **@pascal** and **@stdcall** conventions do).

15.5 Calling Procedures Written in a Different Language

When calling a subroutine written in a different language, your code must pass the parameters as the other language expects and clean up the parameters if the target language requires your code to do so upon return. Generally, calling code written in other languages is relatively easy. You have to ensure that you're passing the parameters in the proper places (e.g., in registers or pushing them on the stack in an appropriate order). Generally, such a call only requires that you provide a suitable external procedure declaration (e.g., swapping the order of the parameters in the parameter list if the language passes parameters in a right-to-left order). Some languages may require additional data structures (e.g., static links) to be passed. It is your responsibility to determine if such data is necessary and pass it to the subroutine you are calling.

15.6 Calling HLA Procedures From Another Language

Calling HLA procedures from another language is somewhat more complex than the converse operation. You still have the problem of parameter ordering; though this is usually fixed by reversing the parameters in the parameter list (e.g., using the `@cdecl` or `@stdcall` procedure options).

A bigger problem is the responsibility of cleaning up the parameters on the stack. By default, an HLA procedure automatically removes parameter data from the stack upon return. If the calling code thinks that it has the responsibility to do this cleanup, the parameter data will be removed twice, with disastrous results. Such code must use the `@cdecl` calling convention or you must use the `@noframe` option (and probably `@nodisplay` as well) to disable the automatic generation of procedure entry and exit code. Then you must manually write the code that sets up the activation record and returns from the procedure. Upon return, you must use the `ret()` instruction without a numeric parameter.

HLA external procedures must always be declared at lex level one. Since the condition of the stack is unknown upon entry into HLA code from some externally written code, your external HLA procedures should not depend upon the display to access non-local variables. HLA procedures that other languages call should always have the `@nodisplay` option associated with them. While it is okay to access non-local STATIC objects, you should never attempt to access non-local `var` objects from a procedure that code written in a different language will call.

HLA's `@pascal`, `@stdcall`, and `@cdecl` procedure options cover the calling conventions of most modern high level languages. However, other calling conventions do exist (for example, the METAWARE compilers give you an option of passing parameters in the left-to-right order and it is the caller's responsibility to clean up the stack afterwards). Some languages don't even pass their parameters on the stack. Some languages pass some or all of the parameters in registers. If you are linking your HLA code with a language that uses one of these non-standard calling conventions, it is your responsibility to write the explicit HLA code that passes these parameters and cleans up the parameter data upon return from the procedure.

15.7 Linking in Code Written in Other Languages

When linking in code written in a different language to an HLA main program, keep in mind that the foreign code may make calls to the standard library associated with the other language. You may need to link in that code as well. Also keep in mind that some compilers emit code that assumes that certain initialization has occurred when the program is loaded into memory. Unfortunately, if the main program is not written in this other language (i.e., main is written in HLA), this initialization might not have been done. This may very well cause the routine you're linking into an HLA program to fail.

Conversely, be very careful about calling HLA standard library routines in code you expect to link into programs written in other languages. The HLA standard library routines (and the exception handling code, in particular), rely upon initialization that the HLA main program performs. This could create a problem, for example, if you attempt to execute some procedure that raises an exception and the exception handling code has not been initialized.

15.8 Calling HLA Code From Other Languages

As explained earlier, calling HLA procedures and functions from other languages is generally easy. Just create an "external" procedure declaration (to make your procedure's name public),

compile the procedure as part of a unit, link it with your other code, and you're in business. There is one catch, and I quote from the chapter on Mixed Language Programming from the first edition of "The Art of Assembly Language":

A large percentage of the HLA Standard Library routines include exception handling statements or call other routines that use exception handling statements. Unless you've set up the HLA exception handling subsystem properly, you should not call any HLA Standard Library Routines from non-HLA programs.

Similarly, you should not use any exception handling statements in code that you call from non-HLA code unless you've properly set up the exception handling subsystem.

Until now, that advice has simply meant "Don't use exceptions and don't call any routines that use exceptions (e.g., HLA Standard Library routines) when calling HLA procedures from a non-HLA main program." What is the reason for this tough restriction? Simple, other than myself and perhaps a few hearty programmers who've probed the internals of HLA-generated code, very few people have known how to set up the HLA exception handling system properly.

Properly setting up the HLA exception handling system isn't that complex. In fact, once you know what you're doing, it's actually quite easy. However, until now that knowledge hasn't been publically available, so the best advice has always been "don't even try it." The purpose of this section is to rectify this situation by describing what you need to do to initialize HLA's exception handling system.

Before going too much farther, I should point out that the information in this document is specific to Windows. While the same concepts apply to Linux, Mac OS X, and FreeBSD there are a few differences. If there is demand for such a thing, I'll be more than happy to create a document such as this one for those users. The principle differences have to do with the way x86 CPU exceptions are handled. The general HLA exception handling mechanism is the same under all OSes, it's just a question of how the HLA exception handling subsystem taps into the OS' exception system. If you're interested in seeing a portable version of the following description, take a look at the source code for the HLABE (HLA back engine) code in the HLA compiler source files. HLABE is HLA code that a C/C++ program calls and it properly sets up the HLA run-time system when called for the HLA compiler.

When an HLA program first starts running, it executes a (compiler-generated) call to an HLA Standard Library procedure called *BuildExcepts*. *BuildExcepts* creates a Windows-compatible SEH (Structured Exception Handling) record in the main program's stack frame. This SEH record becomes the "catch-all" for any exceptions that the program doesn't specifically handle. Should an exception wind its way down to this particular exception handling record, then the code executes the program's default exception handler, that displays an error message and aborts the program.

The problem with calling HLA code from another language is that this default SEH record has never been built, because there is no HLA main program executing that built this record upon initial execution. When an unhandled exception comes along, the system generally crashes or hangs as there exists no default exception handler to deal with the exception. To avoid this problem (so you can use exceptions and call code that uses exceptions), what you have to do is manually build that SEH record yourself. Actually, you don't have to build the SEH record yourself - that's exactly what the HLA Standard Library *BuildExcepts* procedure does. What you have to do is call this procedure so it can build the SEH record for you.

In a normal HLA main program, an application calls *BuildExcepts* exactly once - immediately upon entry into the main program. This creates a single SEH exception handling record that sits around on the stack until the program exits. Unfortunately, when you call HLA code from some other language, you don't get the opportunity to build this SEH record at the beginning of the main program's execution (and even if you did, there is no guarantee that the exception handling system in place in that other language is compatible with HLA's). Therefore, we won't be able to build the SEH record once and forget about it; instead, we'll have to build the SEH record on each call to some HLA procedure from external code, and we'll have to tear down that SEH record before leaving. Yep, this is all overhead that you're going to execute on each call to an HLA function you make from some other language. The good news is that setting up (and tearing down) the SEH record takes less than a dozen instructions, so it's not that big of a deal.

Setting up and tearing down the SEH isn't the only work involved in supporting exceptions in HLA code. There are a couple of routines and a couple of data structures that the HLA compiler automatically generates whenever you write a main program. You'll have to manually supply these routines and data structures yourself.

The data structures exist to support HLA coroutines. Though it's unlikely you'll use coroutines in HLA code you call from C or some other language, you still have to create a coroutine data structure for the "main program" because the HLA exception handling code references this data structure. This is easily achieved with the following HLA code:

```
static
  MainPgmVMT:dword:= &QuitMain;

  // The following comprise the Main Program's coroutine data structure.

  MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
  MainPgmCoroutine: dword; @nostorage;
                    dword &MainPgmVMT, 0, 0;
  SaveSEHPointer:dword; @nostorage;
                    dword 0, 0;
```

The important field in this structure is the *SaveSEHPointer* field. The exception handling system expects a pointer to the previous SEH record in this field. The *BuildExcepts* stores the old SEH pointer in this field, when your code returns it should restore the SEH pointer from this field. You can ignore the remaining fields in these two data structures; they just exist to keep HLA happy.

The HLA Standard Library provides three routines we'll need to reference in the exception handler code we're setting up. However, the HLA Standard Library header files don't provide prototypes for all of these routines (because it would be unusual for user code to call them), therefore, you'll also have to manually supply prototypes for these routines. The prototypes are

```
procedure BuildExcepts; @external("BuildExcepts__hla_");
procedure HardwareException; @external( "HardwareException__hla_" );
procedure DefaultExceptionHandler; @external(
  "DefaultExceptionHandler__hla_" );
```

BuildExcepts we've already discussed. The *HardwareException* procedure is where the system would normally transfer control on a hardware exception. The *DefaultExceptionHandler* is the code that HLA jumps to whenever an exception occurs. The purpose behind these last two procedures is to allow the HLA compiler to link in a separate set of exception handling routines depending on whether you want a "compact" exception handler or the full exception handler (the difference has to do with the size of the string data that HLA would link in). Throughout this paper, we'll assume you want to link in the full exception-handling package. See the details in the HLA reference manual concerning exceptions (and look at the code HLA emits for short exceptions) if you're interested in linking in the shorter version of the exception handler (with a single generic message rather than exception-specific messages).

In addition to the Standard Library routines given above, the HLA compiler also writes a couple of procedures (and provides program termination code). These procedures take the following form:

```
procedure QuitMain;
begin QuitMain;

  ExitProcess( 1 );

end QuitMain;

procedure HWexcept;
begin HWexcept;

  jmp HardwareException;

end HWexcept;
```



```

procedure DfltExHndlr;
begin DfltExHndlr;

    jmp DefaultExceptionHandler;

end DfltExHndlr;

```

QuitMain, in the HLA generated code, is really just a label, not a full procedure. HLA transfers control to this label whenever it wants to terminate the program. As some exceptions will transfer control to this label, you must supply this label in your code. All this procedure's body need do is return control to the operating system. You can actually sneak in anything else you want, but when the procedure completes, it must return control to Windows (e.g., via the `ExitProcess` call).

The *HWexcept* label is where HLA's initialization code points the "hardware exception vector." Specifically, hardware exceptions like divide errors, segmentation faults, bounds violations, etc., first jump to this procedure. This short procedure simply passes control to the routine in the HLA Standard Library that actually handles the hardware exception.

DfltExHndlr is another procedure written by the HLA compiler. The purpose of this routine is to allow HLA code to link with the full exception handler (*DefaultExceptionHandler*) or the short exception handler (see the HLA standard library exception handling code for details). As noted earlier, in this paper we're going to use the full exception handling system.

To explain how to use all these functions and data types, an example is in order. Consider the following C program that will call an HLA procedure named *hlaFunc*:

```

/*
** A demonstration of how you can call HLA code
** that calls the HLA Standard Library from code
** that is not an HLA main program (in this case, it's
** a "C" program).
**
** Note: this program was compiled with Microsoft VC++
** using the following command lines:
**
** c:>vcvars32
** c:>hla -c hlafunc.hla
** c:>cl cdemo.c hlafunc.obj hlalib.lib kernel32.lib user32.lib
*/

#include <stdio.h>

extern void hlaFunc( int value );

int
main( void )
{
    printf( "Calling HLA code\n" );
    hlaFunc( 10 );
    printf( "Returned from HLA code\n" );

    return 0;
}

```

As usual, we'll place the code we want to call from our C function in an HLA unit and compile this to an .OBJ file. Here's the complete HLA procedure (discussion to follow):

```
unit hlaFuncUnit;

// We want to demonstrate how to call HLA Standard Library
// routines from code that is called from C, so let's include
// the standard library right here.

#include( "stdlib.hhf" )

// Here's the sample function we're going to call from external
// code ("C" in this example) that demonstrates HLA stdlib calls
// and exception handling.

procedure hlaFunc( i:int32 ); @cdecl; @external( "_hlaFunc" );

// These are declarations for procedures that exist in the HLA
// standard library, but are "shrouded" in the sense that there
// aren't corresponding declarations in the stdlib.hhf file (these
// routines generally get called by HLA generated code, and nothing
// else; however, as we have to simulate "HLA generated code" here,
// we have to manually provide these declarations):

procedure BuildExcept; @external("BuildExcept__hla_");
procedure HardwareException; @external( "HardwareException__hla_" );
procedure DefaultExceptionHandler; @external(
"DefaultExceptionHandler__hla_" );

// The following are forward/external declarations for procedures
// that are normally created by the HLA compiler when you write
// a "main program." As we are not using an HLA main program here,
// we have to manually create these procedures.

procedure HWexcept; @external( "HWexcept__hla_" );
procedure DfltExHndlr; @external( "DfltExHndlr__hla_" );
procedure QuitMain; @external( "QuitMain__hla_" );

// The following is a Win32 API function this code calls:

procedure ExitProcess( rtnCode:dword ); @external( "_ExitProcess@4" );

// The following are some global, public, variables that the
// HLA exception handling run-time system expect the compiler
// to create for the HLA main program. Once again, as we are not
// writing an HLA main program here, we have to manually supply
// these objects:

static
    MainPgmVMT:dword:= &QuitMain;

    MainPgmCoroutine: dword[ 5 ]; @external( "MainPgmCoroutine__hla_" );
    MainPgmCoroutine: dword; @nostorage;
        dword &MainPgmVMT, 0, 0;
    SaveSEHPointer:dword; @nostorage;
        dword 0, 0;
```

```
// HLA main programs provide a "QuitMain" external label that
// exception handling code can when the exception causes the
// program to abort. This label immediately terminates program
// execution. As we are not writing an HLA main program, the HLA
// compiler does not provide this code for us, we have to supply
// it manually. You can do anything you want here, as long as you
// cause the *whole* program to terminate execution. This particular
// example simply calls ExitProcess and returns a termination code
// of one (which you can change to anything you want; non-zero usually
// indicates successful completion of the application, but this label
// normally gets called when the application aborts because of some
// exception, so returning zero isn't typical in this particular case.

procedure QuitMain;
begin QuitMain;

    ExitProcess( 1 );

end QuitMain;

// HWexcept is where the OS would normally transfer control
// when an x86 exception occurs. This procedure is normally
// written by the HLA compiler and simply jumps to an
// appropriate handler in the HLA Standard Library.

procedure HWexcept;
begin HWexcept;

    jmp HardwareException;

end HWexcept;

// DfltExHndlr is where the exception handling code transfers
// control when an HLA exception occurs. This is normally
// written by the compiler (to allow the compiler to choose
// between the full and short forms of the default exception
// handler). NOTE: the following code invokes the *full*
// exception handler (lots of meaningful messages, at the
// expense of the space needed for all those messages).

procedure DfltExHndlr;
begin DfltExHndlr;

    jmp DefaultExceptionHandler;

end DfltExHndlr;

// Here's the HLA code we're going to call from C that
// demonstrates exception handling without an HLA main program.

procedure hlaFunc( i:int32 );
var
    s:string;
    saveSEH:dword;
```

```
begin hlaFunc;

    // Before doing anything else, save a copy of the SEH pointer:
    #asm
        mov eax, fs:[0]
    #endasm
    mov( eax, saveSEH );

    // Upon entry into any HLA code that needs exception support,
    // we have to set up the structured exception handling record
    // for HLA:

    call BuildExcepts;

    // Because exception handling code can mess up all the registers,
    // we need to preserve EBX, ESI, and EDI across this call:

    push( esi );
    push( edi );
    push( ebx );

    // Okay, here's the code we're going to execute that uses
    // exceptions, calls HLA stdlib routines, etc., even though
    // caller is not an HLA program:

    try

        stdout.put( "stdout.put called from HLA code, i = ", i, nl );
        raise( 5 );

        exception( 5 );
        stdout.put( "Exception handled by HLA code" nl );

    endtry;

    // One more demonstration, this time with an exception
    // occurring deep down inside an HLA Standard Library routine:

    try
        stralloc( 16 );
        mov( eax, s );
        str.cpy( "Hello World", s );
        stdout.put( "Successfully copied 'Hello World' to s: ", s, nl );
        str.cpy( "0123456789abcdefghijklmnop", s );
        stdout.put( "Shouldn't get here" nl );

        anyexception

            stdout.put( "Exception code: ", eax, nl );
            ex.printStackTrace();

    endtry;
    strfree( s );
    stdout.put( "Returning to C code" nl );

    // Restore the registers we saved earlier:
```

```
    pop( ebx );
    pop( edi );
    pop( esi );

    // Restore the saved SEH value:

    mov( saveSEH, eax );
    #asm
        mov fs:[0], eax
    #endasm

end hlaFunc;

end hlaFuncUnit;
```

The *hlaFunc* procedure appearing at the end of this source file is of primary interest to us here. The HLA function you call from C (or any other language) must begin by immediately saving the SEH pointer and then calling *BuildExcepts* upon entry into the procedure. This constructs the HLA SEH record and initializes the HLA exception handling system. Just as important, before the procedure returns it must clean up the SEH record; this is accomplished with the last two **mov** instructions in this code (including the one appearing in the **#asm..#endasm** sequence). Everything between those two points is the normal body of your procedure. This code can use the **try..endtry** statement, raise exceptions, and call external procedures that using **try..end** and/or raise exceptions. The code appearing in this sample both demonstrates directly raising an exception and calling an HLA Standard Library routine that raises an exception. Also note how this code is free to call HLA Standard Library routines without fear of crashing the system should an exception occur.

It is important to realize that you must call *BuildExcepts* and clean up the SEH record in each HLA procedure you call from some other language. Note, however, that you don't have to do this for HLA procedures that you only call from HLA code (that has already built the SEH record).

15.9 Exercising Complete Control with HLA

Note: This section was written with Windows and Unix (Linux/FreeBSD/Mac OS X) programmers in mind. Most of the examples are Windows examples; this document provides Unix-specific examples only when there is a major difference in the way the compiler operates under Windows versus Unix.

A common complaint I get about HLA is that it "hides the machine from the user and generates tons of code behind the programmer's back." This is usually followed by something along the lines of "true assemblers don't do that." The truth is that the HLA compiler generates very little extraneous code and there is actually only a little bit of overhead in an HLA program. Part of the confusion stems from the fact that many users think of calls to the HLA Standard Library as part of "the HLA language." For example, many programmers who see the ubiquitous "Hello World" program written in HLA automatically assume that the "stdout.put" library call is part of the language and demonstrates the "bloat" that exists in HLA. Obviously, such a belief is erroneous since anyone could write their own I/O routines and replace the call to stdout.put with their code and HLA would be none the wiser.

However, to say that there is no code overhead or to say that HLA doesn't emit code behind the programmers back isn't true either. HLA was designed to make learning assembly language programming easy for beginners. Therefore, HLA does automatically generate some code to help beginners. Fortunately, it's easy to turn this extra code generation off and have HLA only generate code that you've written. The purpose of this document is to describe how to turn off all extraneous code generation so you, the advanced assembly programmer, can exercise absolute control over the machine code that HLA generates.

By the way, it is understood that if you intend to exercise absolute control over your machine code, you won't achieve this if you're using HLA's high-level control statements and certain other

high level features that HLA provides. Fortunately, none of those high level features (that generate code behind your back) are necessary in an HLA program. You can easily avoid the extraneous code generation by simply not using those high-level control statements in your assembly programs. Since HLA allows you to write "pure assembly code" without any high level features, and there is nothing forcing you to use those statements, using high level control statements as an example of HLA's bloat is illogical. If you don't want such bloat in your assembly programs, don't use those statements!

15.9.1 Overhead Present in an HLA Program

Many people naturally assume that the HLA compiler introduces a lot of extra code into the assembly file it produces. They base their beliefs on several things including the sophistication of the HLA Standard Library (the HLA compiler must call some code to do some initialization required by the Standard Library, just like C), the sophistication of the data structures, and because of HLA's support for high-level control structures. This, however, is a misconception. Although the HLA compiler does emit some initialization code when it compiles an HLA program, this code is actually quite small; it's probably under a hundred bytes, not thousands of bytes or even hundreds of bytes. So let's get that misconception out of the way real fast; to prove this issue, we'll compile an empty HLA program and take a look at the MASM and Gas code it produces.

15.9.1.1 The "empty" Program

Conceptually, the simplest program we can write (and execute) is the empty program. The empty program compiles and runs, but just immediately returns to Windows without doing much of anything else. One would hope that the empty program would produce the smallest possible executable file size. Here's the empty program in HLA:

```
program t;  
begin t;  
end t;
```

The Canonical Empty Program

Here's the MASM code that (an early version of) HLA emitted (when using MASM as a back-end assembler) under Windows for the above program:

```
; Assembly code emitted by HLA compiler  
; Version 2.0 build 485 (prototype)  
; HLA compiler written by Randall Hyde  
; MASM compatible output  
  
if @Version lt 612  
  .586p  
else  
  .686p  
  .mmx  
  .xmm  
endif  
.model flat, syscall  
option noscoped  
option casemap:none  
  
offset32 equ <offset flat:>
```

```

    assume fs:nothing
    ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

.code

    public      QuitMain__hla_
    public      DfltExHndlr__hla_
    public      _HLAMain
    public      HWexcept__hla_
    public      start
    externdef   shorthwExcept__hla_:near32
    externdef   abstract__hla_:near32
    externdef   BuildExcepts__hla_:near32
    externdef   Raise__hla_:near32
    externdef   shortDfltExcept__hla_:near32

.data

    externdef   MainPgmCoroutine__hla_:byte
    externdef   __imp__MessageBoxA@16:dword
    externdef   __imp__ExitProcess@4:dword

.code

HWexcept__hla_ proc near32
    jmp         shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
    jmp         shortDfltExcept__hla_
DfltExHndlr__hla_ endp

_HLAMain proc near32

start proc near32
start endp

    call       BuildExcepts__hla_
    pushd     0
    mov       ebp, esp
    push     ebp

QuitMain__hla_::
    pushd     0
    call     dword ptr __imp__ExitProcess@4

```

```
_HLAMain endp
```

```
end
```

MASM Output Code for the Empty Program

Consider for a moment, the code appearing just before the main program (`_HLAMain`) in the assembly (MASM syntax) file:

```
HWexcept__hla_ proc near32
    jmp     shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
    jmp     shortDfltExcept__hla_
DfltExHndlr__hla_ endp
```

Obviously, these two jump instructions don't add much code to the executable, but they do jump to some external code, so it's fair to ask about the code associated with `shorthwExcept__hla_` and `shortDfltExcept__hla_` (these are two HLA Standard Library modules). These two procedures are actually quite small; their source code appears in the HLA Standard Library and is duplicated here:

```
unit shorthWexceptionUnit;
?@nodisplay := true;
?@noframe := true;

#macro fallsThrough( procName, externalName );

    procedure procName; @external( @string:externalName );
    procedure procName; begin procName; end procName;

#endmacro

fallsThrough( shw1, SHORTHWEXCEPT__HLA_ )
fallsThrough( shw2, shorthwExcept__hla_ )

procedure shorthWexcept;
begin shorthWexcept;
    mov( 1, eax );
    ret();
end shorthWexcept;

end shorthWexceptionUnit;
```

 The `shorthwExcept__hla_` Procedure

```

static
  messageBox:procedure
  (
    code:uns32;
    var title:var;
    var msg:var;
    n:uns32
  ); external( "__imp__MessageBoxA@16" );

  ExitProcess:procedure( code:uns32 );
    external( "__imp__ExitProcess@4" );

readonly
  defaultMessage:byte; @nostorage;
    byte "Unhandled exception error.", 0;

  HLAException: byte; @nostorage;
    byte "HLA Exception Handler", 0;

#macro fallsThrough( procName, externalName );

  procedure procName; @external( @string:externalName );
  procedure procName; begin procName; end procName;

#endmacro

procedure defaultException; @external( "SHORTDFLTEXCEPT__HLA_" );

fallsThrough( defaultException2, shortDfltExcept__hla_ );
fallsThrough( defaultException3, _shortDfltExcept__hla_ );

procedure defaultException;
begin defaultException;

  messageBox( $30, HLAException, DefaultMessage, 0 );
  ExitProcess( 0 );

end defaultException;

```

 The `shortDfltExcept__hla_` Procedure (Win32 version)

If you know anything about machine code, you'll probably realize quick that these procedures are very small. In fact, there are probably more bytes required for the two exception strings as the actual object code requires. Although I haven't actually counted the bytes, I'd guess that these two procedures and their data are well under 100 bytes, total.

Returning to the empty program, the main program (*_HLAMain*) for this file contains the following MASM code:

```
_HLAMain proc near32

start   proc near32
start   endp

        call     BuildExcept$__hla_
        pushd   0
        mov     ebp, esp
        push   ebp

QuitMain__hla_::
        pushd   0
        call   dword ptr __imp__ExitProcess@4
_HLAMain endp
```

The call to *BuildExcept\$__hla_* and the three instructions that follow are the "overhead" associated with a typical HLA program. The last two instructions return control to the operating system; It's hard to call these two instructions overhead as every Windows program is going to need something like these two instructions (these would only be overhead if the program returns to Windows somewhere else and these last two instructions never execute).

The three instructions following the call above set up the stack frame for the main program. This provides access to the **var** objects found in the main program (there are none, or there would actually be another **sub** instruction present above). In some respect, these instructions are pure overhead since there are no automatic (**var**) objects in this program (and HLA sets up the stack frame in order to access automatic variables from the main program). However, we are talking about *three instructions* here that normally execute only once. I'm claiming that this isn't an incredible amount of bloat.

That leaves us with the call to the *BuildExcept\$__hla_* procedure. This is another HLA Standard Library module that initializes HLA's exception handling system. Here's what the code to the *BuildExcept\$__hla_* procedure looks like (for the non-threaded version, threaded code has additional overhead and we won't consider that here):

```
pop( eax );

// Fill in the MainPgmCoroutine data structure:

lea( ebx, MainPgmCoroutineVMT );
mov( ebx, MainPgmCoroutine.theCoroutineVMT );
mov( 0, MainPgmCoroutine.currentESP );
mov( 0, MainPgmCoroutine.stackPointer );
mov( 0, MainPgmCoroutine.pointerToLastCaller );

// Build an structured exception handler frame on the stack:

pushd( &DfltExHandlr );
push( ebp );
pushd( &MainPgmCoroutine );
pushd( &HWexcept );
```

```

// push( fs:[0] );

xor( ebx, ebx );
fseg:push( (type dword [ebx]) );

// mov( esp, fs:[0] );

fseg:mov( esp, [ebx] );

// We need to initialize the main program's coroutine object
// with the pointer to the exception record we just created.
// Note that we must initialize the ExceptionContext field
// with this address:

mov( esp, MainPgmCoroutine.exceptionContext );

jmp( eax );

```

HLA Standard Library BuildExcepts_hla_Procedure

Again, as you can see, there's not a whole lot of code here. The vast majority of this code simply initializes HLA's exception handling subsystem. You've just seen all the "bloated" code that HLA emits for most programs. You'll see a little bit later than it's even possible to remove all this code from an HLA output file (assuming you can live without exception support or are willing to write the code to support exceptions yourself).

15.9.2 The empty Program, Part II

Although the empty program of the previous section is the smallest program we can write that will compile and run, it's not the smallest program we can create with HLA, assuming we don't care if it doesn't run. The smallest possible program you can write with HLA would consist of a **unit** with a single procedure that has no instructions associated with it. The following is such an empty program:

```

unit empty;

  procedure main; @external( "_HLAMain" );
  procedure main; @noframe;
  begin main;
  end main;

end empty;

```

The "empty2" Program

To properly link and produce an .EXE file without error, an HLA program must have a procedure named *_HLAMain*. the external declaration above and the corresponding procedure declaration for main achieves this. Note the presence of the **@noframe** procedure option. This tells HLA to skip any extra code emission for the procedure. Here's a typical MASM file that the above produces when you tell HLA to emit a MASM-compatible assembly language source file:

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

    if @Version lt 612
        .586p
    else
        .686p
        .mmx
        .xmm
    endif
    .model flat, syscall
    option noscoped
    option casemap:none

offset32 equ <offset flat:>

    assume fs:nothing
    ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

.code

    public     _HLMMain
    externdef  HWexcept__hla_:near32
    externdef  abstract__hla_:near32
    externdef  Raise__hla_:near32
    externdef  shortDfltExcept__hla_:near32

.data

    externdef  __imp__MessageBoxA@16:dword
    externdef  __imp__ExitProcess@4:dword

.code

    _HLMMain proc near32
    _HLMMain endp
```

end

MASM Output File for the "empty2" Program

For this program, all of the include files are empty, so there's no need to list them here. If you compile this program to an executable, the resulting file is only 400-500 bytes long. This is because there is no code, so we don't need a 4K block associated with the code; there is no data, so we don't need a 4K block associated with the data segment; there is no Win32 API pointer data because we don't make any Win32 API calls. The PE/COFF header information still requires some memory, which is why the file is 400-500 bytes long.

15.9.3 Overhead Associated With Exceptions

As you saw earlier in the "empty" example, there is a bit of overhead associated with HLA's exception support. The empty program requires somewhere around 100 bytes of data and code to support exceptions. In fact, if you're sloppy or unaware, HLA's exception handling facilities can require quite a bit more overhead. Consider the following program:

```
program empty3;
#include( "stdlib.hhf" )
begin empty3;
end empty3;
```

The "empty3" Program

Here's the MASM code that the HLA compiler produces when you compile empty3 (specifying MASM output):

```
; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

if @Version lt 612
.586p
else
.686p
.mmx
.xmm
endif
.model flat, syscall
option noscoped
option casemap:none

offset32 equ <offset flat:>

assume fs:nothing
```

```
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

.code

    public      QuitMain__hla_
    public      DfltExHndlr__hla_
    public      _HLAMain
    public      HWexcept__hla_
    public      start
    externdef   DefaultExceptionHandler__hla_:near32
    externdef   abstract__hla_:near32
    externdef   HardwareException__hla_:near32
    externdef   BuildExcepts__hla_:near32
    externdef   Raise__hla_:near32
    externdef   shortDfltExcept__hla_:near32

.data

    externdef   MainPgmCoroutine__hla_:byte
    externdef   __imp_MessageBoxA@16:dword
    externdef   __imp_ExitProcess@4:dword
    align      (4)

.code

    ;/* HWexcept__hla_ gets called when Windows raises the exception. */

    HWexcept__hla_ proc near32
        jmp     HardwareException__hla_
    HWexcept__hla_ endp

    DfltExHndlr__hla_ proc near32
        jmp     DefaultExceptionHandler__hla_
    DfltExHndlr__hla_ endp

    _HLAMain proc near32
```

```

start   proc near32
start   endp

        call    BuildExcepts__hla_
        pushd   0
        mov     ebp, esp
        push    ebp

QuitMain__hla_::
        pushd   0
        call    dword ptr __imp__ExitProcess@4
_HLMain endp

        end

```

The "empty3.asm" Output File

You'll have to look close to see a difference between this MASM file and the one for the original "empty" program. Here are the lines that changed:

```

HWexcept__hla_ proc    near32
        jmp HardwareException__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp DefaultExceptionHandler
DfltExHndlr__hla_ endp

```

Here's the original code:

```

HWexcept__hla_ proc    near32
        jmp    shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
        jmp    shortDfltExcept__hla_
DfltExHndlr__hla_ endp

```

The difference between the two is the standard library routines that they call. By the way, if you compile this code to an .EXE file, you'll discover that the .EXE file is exactly the same size as the original code: 10,732 bytes (with HLA v2.0). However, it turns out that there is over 3K of additional data in the empty3 version of this program. What is it that the #include("stdlib.hhf") has done to this code?

Well, the stdlib.hhf header file includes the excepts.hhf header file and the excepts.hhf header file assigns the value "true" to an HLA compile-time variable (**@exceptions**) that tells HLA whether you want the full exception handling system or an abbreviated version. When the HLA compiler

encounters the `begin` clause associated with the main program, it checks the value of this compile-time variable. If it contains `true`, then HLA emits the `HWexcept_hla_` and `DfltExHndlr_hla_` procedures that transfer control to the full exception handler code (`HardwareException_hla_` and `DefaultExceptionHandler_hla_`). If the `@exceptions` compile-time variable contains `false` (the default value), then HLA emits these procedures with jumps to the shortened versions of these routines. Now the code for the full routines isn't a whole lot larger than the code for the short routines, the big difference is the amount of data. The short exception handler routines print a very short generic message (the same message for all exceptions) if they wind up being invoked. The full routines print a descriptive message that varies by the actual exception the system raises. Therefore, the full version of the exception handling code has this really big string array and all the data associated with that array is what consumes the better than 3K of additional space that the `empty3` program requires.

Since the `@exceptions` variable is a compile-time variable you can set during compilation, you can force HLA to use the shortened default exception handlers, even if you've included `stdlib.hhf` or `excepts.hhf`, by simply setting `@exceptions` to `false` prior to the `begin` clause of the main program, e.g.,

```
program empty3;
#include( "stdlib.hhf" )
?@exceptions := false;
begin empty3;
end empty3;
```

Program 2.11[^]: Modified 'empty3' Program That Uses the Short Exception Code

If you don't really need, or care about, informative exception handling in your code, and you're including the `excepts.hhf` header file (or some other header file that indirectly includes `excepts.hhf`, and this includes many of the Standard Library header files), then you can trim the size of your program down a bit by setting `@exceptions` to `false` prior to the `begin` clause of your main program. Note, however, that having nice descriptive messages is great when an exception actually occurs; so it's probably a good idea to use the full exception-handling package when you're testing and debugging your code. Then set `@exceptions` to `false` before creating your production code to shave 3K off the executable's size.

Note that you cannot trap any hardware exceptions (e.g., divide by zero) when using the short exception handler. If you want to be able to trap hardware exceptions but you don't want the overhead of the exception string messages you've got a couple of choices: (1) implement Windows structured exception handling yourself (difficult) or (2) grab the sources to the exception handling library code and remove all the message strings. Generally, 3K is such a small amount that it isn't worth the effort to try and shave this data from your code.

Later, this document will discuss the overhead associated with HLA's high-level control statements. But as long as we're on the subject of exceptions, it's probably worthwhile to explore the cost of the HLA `raise` and `try..endtry` statements. Here's a sample HLA program that exercises these statements and the corresponding MASM code:

```
program ExceptsDemo;
begin ExceptsDemo;

    #asm ;raise stmt #endasm
    raise( 1 );

    #asm ;try stmt #endasm

    try

        mov( 0, al );
```

```

    #asm ;unprotected stmt #endasm

unprotected

    mov( 1, al );

    #asm ;exception( 1 ) stmt #endasm

exception( 1 )

    mov( 2, al );

    #asm ;exception( 2 ) stmt #endasm

exception( 2 )

    mov( 3, al );

    #asm ;anyexception stmt #endasm

anyexception

    mov( 4, al );

    #asm ;endtry stmt #endasm

endtry;
mov( 5, al );

end ExceptsDemo;

```

Sample HLA Program to Demonstrate Exceptions

```

; Assembly code emitted by HLA compiler
; Version 2.0 build 483 (prototype)
; HLA compiler written by Randall Hyde
; MASM compatible output

if @Version lt 612
.586p
else
.686p
.mmx
.xmm
endif
.model flat, syscall
option nscoped
option casemap:none

offset32 equ <offset flat:>

assume fs:nothing

```

```
ExceptionPtr__hla_ equ <(dword ptr fs:[0])>

.code

    public    QuitMain__hla_
    public    DfltExHndlr__hla_
    public    _HLAMain
    public    HWexcept__hla_
    public    start
    externdef shorthwExcept__hla_:near32
    externdef abstract__hla_:near32
    externdef BuildExcepts__hla_:near32
    externdef Raise__hla_:near32
    externdef shortDfltExcept__hla_:near32

exception__hla_5 equ Raise__hla_

.data

    externdef MainPgmCoroutine__hla_:byte
    externdef __imp__MessageBoxA@16:dword
    externdef __imp__ExitProcess@4:dword

.code

HWexcept__hla_ proc near32
    jmp      shorthwExcept__hla_
HWexcept__hla_ endp

DfltExHndlr__hla_ proc near32
    jmp      shortDfltExcept__hla_
DfltExHndlr__hla_ endp

_HLAMain proc near32

start proc near32
start endp

    call    BuildExcepts__hla_
    pushd  0
    mov     ebp, esp
    push   ebp
```

```

                ;raise stmt
mov             eax, 1
jmp            Raise__hla_

                ;try stmt
pushd          offset32 exception__hla_2
push           ebp
db             064h
mov            ebp, ds:[0]
push           dword ptr [ebp+8]
mov            ebp, [esp+4]
pushd          offset32 HWexcept__hla_
db             064h
push           dword ptr ds:[0]
db             064h
mov            ds:[0], esp
mov            al, 0

                ;unprotected stmt
db             064h
mov            esp, ds:[0]
db             064h
pop            dword ptr ds:[0]
add            esp, 8
pop            ebp
add            esp, 4
mov            al, 1

                ;exception( 1 ) stmt
jmp            endtry__hla_1
exception__hla_2:
cmp            eax, 1
jne            exception__hla_3
mov            al, 2

                ;exception( 2 ) stmt
jmp            endtry__hla_1
exception__hla_3:
cmp            eax, 2
jne            exception__hla_4
mov            al, 3

                ;anyexception stmt
jmp            endtry__hla_1
exception__hla_4:
mov            al, 4

                ;endtry stmt
endtry__hla_1:
mov            al, 5
QuitMain__hla_::
pushd          0
call           dword ptr __imp__ExitProcess@4
_HLAMain endp

```

end

MASM Output File From the Exceptions Source

The purpose of this document is not to explain how structured exception handling under Windows works (upon which HLA's exception handlers are based). Therefore, I'm not going to bother explaining what any of the statements mean in the code above. Instead, the important thing is to note the amount of code that each statement or clause produces.

The **raise** statement is simple. It loads the value of its argument into EAX and then transfers control to the *Raise_hla_* standard library procedure (see the standard library sources if you're interested, it is a fairly short routine, though). As you can see, the **raise** statement doesn't generate a whole lot of code.

The **try..endtry** statement is at the other extreme. This statement probably generates more code than any other single high-level control statement that HLA provides¹. To get an idea of the amount of code generated for each clause, note that I've used the **#asm.#endasm** directive to inject comments into the MASM output file and I've used instructions of the form "mov(const, al);" to help delineate the code that HLA produces for each of the **try..endtry** clauses.

The **try..endtry** statement is very powerful and provides a sophisticated solution to the problem of exception handling. However, as you can see, the **try..endtry** sequence generates quite a bit of code (not a tremendous amount, but it add up if you place a lot of **try..endtry** statements in your program). If you're trying to write code that is as fast and as short as possible, you may produce better quality code by simply returning an error status from your procedures and functions rather than raising exceptions in those functions and relying on a **try..endtry** block to catch the exception. There is no guarantee that the explicit return value approach is faster or shorter, but it usually is shorter and faster (though it's nowhere near as convenient as **raise/try..endtry** and far more error prone). Just something to keep in mind.

15.9.4 Overhead Associated with Procedures, Iterators, and Methods

HLA was designed as a tool to teach assembly language programming to absolute beginners. Therefore, it does a couple of things that, by default, make it easier on beginners but may produce some excess code that an advanced assembly programmer would never write. One place where this is especially true is in the declaration and invocation of HLA procedures. Fortunately, HLA provides many options that let you control the extra code it emits for beginners (including turning off the code generation). This section explores the options you can use to control code generation for procedures and calls to procedures².

By default, HLA automatically generates code at the beginning of a procedure to construct the activation record for that procedure, align the stack to a double-word boundary, allocate local variables, and build a display for that procedure³. HLA also automatically generates the code to clean up the activation record and return from the procedure (and for the **@stdcall** and **@pascal** calling sequences, this code also cleans up the parameters on the stack). Sometimes this code is unnecessary (e.g., the procedure doesn't have any stack-based parameters or local variables),

-
1. Technically speaking, this is not true. Using the conjunction(&&) and disjunction (||) operators, you can generate some really large if, while, etc., statements. However, anyone who creates a really huge boolean expression is going to expect a bit of code bloat).
 2. This section will use the generic term "procedures" to mean any HLA procedure, iterator, or method, unless otherwise noted.
 3. Displays are advanced data structures that provide access to non-local automatic variables.

slightly less than efficient (e.g., you can access all the parameters and locals off ESP and you don't need to set up a stack frame with EBP), or you want to do things a little differently for some specific reason. Obviously, in these situations, HLA's default behavior is not what you want. Fortunately, HLA makes it easy to modify its behavior for a specific procedure or even change the overall default behavior.

To begin with, it's probably a good idea to look at the code HLA automatically generates for a procedure. We'll use the following example repeatedly with slight modifications in this section:

```
program ProcDemo;

    procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    var
        localVar:    dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;
```

The Generic HLA Procedure

Here's the MASM assembly output for the demo procedure above (for the sake of brevity, I'll not put the whole MASM output file here - it's roughly the same code you'll find in the empty examples):

```
demo__hla_1 proc near32
    push        ebp
    push        dword ptr [ebp-4]
; /*Get frame ptr*/
    lea        ebp, [esp+4]
    push        ebp
    sub        esp, 4
    and        esp, -4
    nop
xdemo__hla_1__hla_:
    mov        esp, ebp
    pop        ebp
    ret        16
demo__hla_1 endp
```

HLA Code Generation (MASM assembly output) for the 'demo' Procedure

Notice that the original procedure only had one instruction (a **nop**). HLA actually generates nine additional instructions inside this procedure. While some of them (e.g., the **ret** instruction) would have to be present, some fat here can be trimmed, depending on your circumstances.

The first thing that you can usually trim away is the generation of the code that builds the display. This is the second through fourth instructions above (**push, lea, push**). Displays are a special data structure that provides access to non-local automatic variables in nested procedures. 99% of the time (or better), most assembly procedures won't need a display. That's because 98% of all assembly language programmers will never nest their procedures and the 2% that do can often pull other tricks to access non-local variables without using a display. Therefore, the vast majority of the time you can eliminate these statements that set up the display from the procedure code. This is easily accomplished by supplying the **@nodisplay** procedure option, e.g.,

```

program ProcDemo;

    procedure demo( b:byte; w:word; d:dword; var refvar:dword );
@nodisplay;
    var
        localVar:    dword;

    begin demo;

        nop();

    end demo;

begin ProcDemo;
end ProcDemo;

```

HLA Demo Program with @nodisplay Option

Here's the corresponding code that HLA emits for the program above:

```

demo__hla_1 proc near32
    push    ebp
    mov     ebp, esp
    sub     esp, 4
    and     esp, -4
    nop
xdemo__hla_1__hla_:
    mov     esp, ebp
    pop    ebp
    ret    16
demo__hla_1 endp

```

HLA Code Generation for Demo With @nodisplay Option

Well, this code looks a whole lot closer to a procedure with a standard entry/exit sequence. About the only surprising piece of code here is the **and** instruction. HLA automatically emits this code to guarantee that the stack is aligned upon a four-byte boundary upon entering the procedure. If the caller has misaligned the value in ESP such that it is not an even multiple of four, certain system calls may fail. The **and** instruction above ensures that ESP is double-word aligned. Unless you mess with ESP's value (or push word values on the stack), ESP is always double-word aligned. Note that this is true even if you specify some number of local variables whose aggregate size is not

an even multiple of four (the sub instruction above reduces ESP by the number of bytes of local variables present, but HLA always rounds this value up to the next even multiple of four to keep ESP double-word aligned). If you know that ESP is double-word aligned (because you've not messed with the stack pointer), then the and instruction in the code above is superfluous. You may eliminate this extra instruction by specifying the `@noalignstack` procedure option:

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword );
  @nodisplay;
  @noalignstack;

  var
    localVar:  dword;

  begin demo;

    nop();

  end demo;

begin ProcDemo;
end ProcDemo;

```

HLA Demo Code With `@noalignstack` Option

Here's the corresponding code that HLA emits for the program above:

```

demo__hla_1 proc near32
  push    ebp
  mov     ebp, esp
  sub     esp, 4
  nop
xdemo__hla_1__hla_:
  mov     esp, ebp
  pop     ebp
  ret     16
demo__hla_1 endp

```

HLA Code Generation (MASM syntax) for Demo With `@nodisplay` Option

Now we've gotten down to the point where the code looks just like the standard entry/exit sequence you'd expect for a procedure. Of course, we could make some changes still. For example, the 80x86 CPU family supports two instructions, `enter` and `leave`, that you may use to build and destroy activation records (including displays, if necessary). While these instructions are typically slower than the discrete instructions that do the same job, they are certainly shorter and, therefore, some programmers prefer to use them. By default, HLA generates discrete instructions to build and destroy activation records. However, by using the `@enter` and `@leave` procedure options, you can tell HLA to use these instructions rather than the discrete instruction sequences:

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword );
  @nodisplay;
  @noalignstack;
  @enter;
  @leave;

  var
    localVar:   dword;

  begin demo;

    nop();

  end demo;

begin ProcDemo;
end ProcDemo;

```

HLA Demo Code With @enter and @leave Options

Here's the corresponding code that HLA emits for the program above:

```

demo__hla_1 proc near32
  enter      0, 4
  nop
xdemo__hla_1__hla_:
  leave
  ret       16
demo__hla_1 endp

```

HLA Code Generation for Demo With @enter and @leave Options

As you can see, this procedure is starting to become seriously shortened. HLA is emitting only three extra instructions (down from the original nine or so).

Of course, 'real' assembly language programmers want to write all their own code. If HLA is automatically generating anything for them, no matter how convenient, they're going to complain. Well, HLA provides the **@noframe** procedure option that eliminates all code generation other than the explicit machine instructions the programmer provides. Note that supplying **@noframe** implicitly supplies **@noalignstack** and, to a certain extent, **@nodisplay** since **@noframe** turns off all extra code generation in a procedure¹. Here are the examples above specifying **@noframe**:

```

procedure demo( b:byte; w:word; d:dword; var refvar:dword );

```

1. Note, however, that if **@noframe** is not present, HLA will still assume you want to allocate storage for a display and will consider this fact when assigning offsets to local variables found in the procedure. Therefore, it's a good idea to go ahead and specify **@nodisplay** along with **@noframe**.


```
@nodisplay; // Still should be here, see footnote
@noframe;

var
    localVar:    dword;

begin demo;

    nop();

end demo;

begin ProcDemo;
end ProcDemo;
```

HLA Demo Code With @noframe Option

Here's the corresponding code that HLA emits for the program above:

```
demo__hla_1 proc near32
    nop
demo__hla_1 endp
```

HLA Code Generation for Demo With @nodisplay and @noframe Options

Now, however, we have a problem. There is no **ret** instruction to return from this procedure. But that's okay, the "macho" assembly programmer who doesn't want HLA generating any code for them surely wants the program to fall through this procedure to the next instruction in memory, or they wouldn't have left out the **ret** instruction in the original code. Here's what the procedure would normally look like when the **@noframe** option is present:

```
procedure demo( b:byte; w:word; d:dword; var refvar:dword );
    @nodisplay; // Still should be here, see footnote
    @noframe;

    var
        localVar:    dword;

    begin demo;

        nop();
        ret( 16 );

    end demo;

begin ProcDemo;
end ProcDemo;
```

 HLA Demo Code With `@noframe` and `@nodisplay` Options (Part II)

For those who want to write all their own code and not have HLA generate anything extra code in their procedures, constantly attaching `@noframe` and `@nodisplay` to every procedure declaration can get old, fast. Fortunately, HLA provides a mechanism that lets you set the default state for all of these procedure options.

As shipped, HLA defaults to the following options: `@frame`, `@display`, `@alignstack`, `@noenter`, and `@noleave`. You can change the defaults by using these options as compile-time variables and setting them to true or false. Here are the possible options:

: Procedure Options and Their Effect on Code Generation

Option	Effect if set to true	Effect if set to false
<code>@enter</code>	HLA generates ENTER instruction to build activation records upon procedure entry. Note that <code>@frame</code> must also be true for HLA to emit this code.	HLA generates discrete instructions to build activation records upon procedure entry. Note that <code>@frame</code> must also be true for HLA to emit this code.
<code>@noenter</code>	Same as setting <code>@enter</code> to false.	Same as setting <code>@enter</code> to true.
<code>@leave</code>	HLA emits the <code>leave</code> instruction to clean up the activation record upon exit. Note that <code>@frame</code> must also be true for HLA to emit this code.	HLA emits discrete instructions (<code>mov</code> , <code>pop</code>) to clean up the activation record upon exit. Note that <code>@frame</code> must also be true for HLA to emit this code.
<code>@noleave</code>	Same as setting <code>@leave</code> to false.	Same as setting <code>@leave</code> to true.
<code>@display</code>	HLA emits instructions that allocate storage for and initialize a display structure. If <code>@enter</code> is true, HLA emits an <code>enter</code> instruction to accomplish this, otherwise it emits discrete instructions. Note that <code>@frame</code> must also be true for HLA to emit this code.	HLA does not emit any instructions that allocate or initialize the display structure.
<code>@nodisplay</code>	Same as setting <code>@display</code> false.	Same as setting <code>@display</code> true
<code>@alignstack</code>	HLA emits an <code>and</code> instruction that guarantees ESP is double-word aligned after allocating local variables. Note that <code>@frame</code> must also be true for HLA to emit this code.	HLA does not emit the <code>and</code> instruction that double-word aligns ESP.
<code>@noalignstack</code>	Same as setting <code>@alignstack</code> to false.	Same as setting <code>@alignstack</code> to true.
<code>@frame</code>	HLA generates code to construct the stack frame and other duties (e.g., align the stack if <code>@alignstack</code> is true, build the display if <code>@display</code> is true).	HLA does not generate any extra code for the procedure. It is the programmer's responsibility to write any necessary code to build the stack frame, if required.
<code>@noframe</code>	Same as setting <code>@frame</code> to false	Same as setting <code>@frame</code> to true.

The "macho" assembly language programmer will probably include the following two statements at the beginning of every HLA program they write:

```
?@noframe := true;
?@nodisplay := true;
```

The inclusion of these two statements tells HLA that the programmer is responsible for writing all the code that appears within the source file. Note that you may re-enable display and frame generation on a procedure-by-procedure basis by using the **@frame** and **@display** procedure options. See the discussion of procedure options in the HLA reference manual for more details.

Note that HLA still makes parameter names and local variable names available to your procedures when you specify the **@noframe** option. However, the offsets associated with these variables assume that you've built a standard stack frame and that you're going to reference the objects off EBP. If this is not the case, then you should not use the parameter and local variable names in your code; you'll have to use numeric offsets (say, from ESP) or, better yet, create TEXT constants that provide the necessary offsets from ESP, e.g.,

```
program ProcDemo;

?@noframe := true;
?@nodisplay := true;

procedure demo( _d:dword; var _refvar:dword );
const
    d      :text := "(type dword [esp+12])";
    refvar :text := "(type dword [esp+8])";
    localvar:text := "(type dword [esp])";
begin demo;

    pushd( 0 );    // Allocate _localVar and initialize to zero.
    mov( d, eax );
    mov( eax, localvar );
    mov( refvar, ebx );
    mov( eax, [eax] );
    add( 4, esp ); // Remove localvar from stack.
    ret( 8 );     // Return and pop parameters

end demo;

begin ProcDemo;
end ProcDemo;
```

Using TEXT Constants to Access Parameters and Local Variables

```
demo__hla_1 proc near32
    pushd    0
    mov     eax, dword ptr [esp+12]
    mov     dword ptr [esp], eax
    mov     ebx, dword ptr [esp+8]
    mov     [eax], eax
    add     esp, 4
```

```

        ret            8
demo__hla_1 endp

```

Code Generation for the Above HLA Procedure (MASM syntax output)

15.9.5 Overhead Associated with Procedure Calls

As long as you manually pass the parameters yourself and use the **call** instruction, HLA does not inject any extra instructions into your code. However, if you use HLA's high-level procedure call syntax, HLA may very well emit some extra instructions into the code stream. If this bothers you, well, don't use the high level calling syntax - stick with the manual ("pure assembly") calling syntax.

However, the high level calling syntax is very convenient, it is far more readable and maintainable, and most of the time it generates exactly the same code you're going to write by hand. Therefore, it makes sense to use it as often as you can and understand the degenerate cases (where HLA emits some bad code) so you can code those by hand when efficiency is a prime concern.

First, HLA does a great job with "pass by value" parameters when the size of the value is four, eight, or 16 bytes. Such parameters generally require only a single instruction per double word to push on the stack prior to the call¹. As the objects get larger, passing them by value gets very expensive. At some point, HLA doesn't bother trying to push the data on the stack, instead, it uses a **movsd** instruction to copy the data onto the stack. The following code shows what happens when you try to pass a 256-byte variable by value:

```

program ProcDemo;

type
    b256:byte[256];

    procedure demo( b:b256 );
    begin demo;
    end demo;

static
    c:b256;

begin ProcDemo;

    demo( c );

end ProcDemo;

```

Code That Passes a 256-byte Array by Value

```

        lea        esp, [esp-256]
        push      esi
        push      edi
        push      ecx
        pushfd

```

1. Assuming of course, you're passing the parameters on the stack and not in a register.

```
cld
lea     esi, c_hla_2
mov     ecx, 64
lea     edi, [esp+16]
rep movsd
popfd
pop     ecx
pop     edi
pop     esi
call    demo__hla_1
```

MASM Code HLA Emits for the Call to 'demo' Above

This isn't an example of HLA generating bloated code. HLA is doing a reasonable job given the request of the source code. However, HLA makes it so easy to write code that blows up like this that you can often make a mistake and pass a large data structure by value, causing HLA to generate a fair amount of slowing executing code. Actually, once you get above 64 bytes, HLA usually generates a sequence like the one above (with possibly one or two additional instructions if the object's size is not an even multiple of four bytes. So the size won't change too much as the object gets larger, but the execution time required by the **rep movsd** instruction goes up linearly with the size of the object. Moral of the story: unless there are good semantic reasons for doing so, always pass large objects by reference rather than by value. Watch out for this, because HLA will gladly emit the code to pass it by value without complaining.

Note that for parameters up to 64 bytes in size, HLA will actually emit a series of discrete push instructions. For parameters that are 16 bytes or less, this is no big deal (it only takes four push instructions to pass a 16-bit parameter by value). However, it's going to take 16 push instructions to pass a single 64-bit parameter by value to a procedure. This can cause some serious code bloat if you're doing this a lot. Moral: same as before, pass large objects by reference rather than by value (large is probably anything greater than 16 bytes in size).

HLA can go through some real gymnastics attempting to pass small parameters by value, as well. Because most modern (32-bit) operating systems always expect the stack to be double-word aligned, HLA (like most languages and OS API functions) always passes a parameter using a multiple of four bytes to hold that value. So if you're passing an object that's one, two, or three bytes in size, HLA will pass four bytes as the actual parameter. The procedure (generally, this is actually up to the programmer) ignores the extra bytes. This creates a problem when attempting to pass certain parameters on the stack; HLA solves these problems at the expense of greater code. Consider the following HLA program that has a one byte parameter and calls the procedure several different ways:

```
program smallParmDemo;

procedure byteParm( b:byte );
begin byteParm;
end byteParm;

static
    b:byte;

begin smallParmDemo;

    byteParm( b );
    byteParm( al );
    byteParm( ah );
    byteParm( (type byte [eax]) );
```

```
end smallParmDemo;
```

Procedure with a One-Byte Parameter

Here's the MASM code HLA generates for each of the calls to *byteParm*:

```

pushd    0
push     eax
mov      al, b_hla_2
mov      [esp+4], al
pop      eax
call     byteParm__hla_1
push     eax
call     byteParm__hla_1
sub      esp, 4
mov      [esp], ah
call     byteParm__hla_1
pushd    0
push     eax
mov      al, byte ptr [eax]
mov      [esp+4], al
pop      eax
call     byteParm__hla_1

```

MASM Code HLA Generates for the Calls to *byteParm*

Many of these calls have an incredible amount of bloat! Any mediocre assembly programmer can probably do a better job than this! Why is HLA so bad? The reason HLA generates some ugly code here is because HLA makes a promise that it won't change any register values when passing parameters to a procedure (just in case you're passing some additional parameters in some registers). This promise severely impacts HLA's options when it comes to copying parameter data to the stack¹. Indeed, about the only option HLA has when it needs a register is to preserve that register's contents while copying the parameter data. Consider the first call to *byteParm* above (passing the byte variable *b*). HLA first makes room for *b* on the stack by pushing a double word zero value. The HLA emits code to push the value of EAX, copy *b*'s value into AL, store AL into the stack location allocated earlier, and then restore EAX's original value.

Now the clever assembly programmer might claim that this could be done far more efficiently with a single instruction, as follows:

```
push( (type dword b) );
```

99.999% of the time, that programmer would be right; this is a much better way to pass a single byte parameter in a dword slot on the stack (this instruction pushes the value of the three bytes the follow *b* in memory, but since the procedure will ignore those three bytes anyway, who cares?). Unfortunately, this trick fails spectacularly in one very special (and, admittedly, rare) case. Consider what happens when *b* is allocated as the 4096th byte in a page and the next page in memory is not read-enabled. This is cause the program to crash. Granted, it's incredibly unlikely

1. It is interesting to note that MASM does not make this same promise. It will happily wipe out the EAX register if it needs a scratch-pad register while passing parameter data to a procedure via the INVOKE statement. I like to believe that HLA is a bit more "civilized" in this regard.

that this will ever happen in an HLA program. However, HLA's design can't assume that it won't ever happen. So HLA has to generate safe, but ugly, code.

Of course, there's nothing preventing you from recognizing this problem and manually pushing *b*'s value as a dword yourself. E.g., either of the following will work:

```
push( (type dword b) );
call byteParm;
```

-or-

```
byteParm( #{ push( (type dword b) ); }# );
```

As long as you can ensure that there are three reasonable bytes following *b*, this scheme is quite a bit more efficient than the default code HLA generates.

The second and third calls to *byteParm* in the example above are the ones where HLA actually generates halfway decent code. If the byte parameter falls in the L.O. byte of a 32-bit register, HLA will simply push the contents of that 32-bit register onto the stack. You aren't going to do any better than this (short of passing the parameter in a register, rather than on the stack). The second call, passing the byte parameter in AH (or any other byte register that is not the L.O. byte of a 32-bit register) needs two instructions: one to allocate storage on the stack (**push**) and another to copy the register's value onto the stack. An expert assembly language programmer, if they know they've got a register to play around with, can, perhaps, generate slightly better code by copying the 8-bit value to the L.O. byte of that register and then pushing the full register, e.g.,

```
mov( al, bl );
push( ebx );
```

This sequence is slightly shorter, though probably not any faster, than the code that HLA generates.

The fourth example above is really just a special case of the first example. If you look at the two code sequences, you'll notice that they are equivalent.

HLA generates less than stellar code for some of these sequences because it assumes that all registers are in use and it shouldn't modify any register values. Obviously, this is not always the case when you're calling a procedure. However, it's a rather difficult problem for HLA to automatically determine if there is a free register available that it can use while passing parameters. Fortunately, HLA provides a way for you to tell it that it can freely use one register (which is all it needs) for processing parameters: the **@use reg** procedure option. Consider the following modification of the previous program:

```
program smallParmDemo;

procedure byteParm( b:byte ); @use ebx;
begin byteParm;
end byteParm;

static
  b:byte;

begin smallParmDemo;

  byteParm( b );
  byteParm( al );
  byteParm( ah );
  byteParm( (type byte [eax]) );

end smallParmDemo;
```

byteParm and @use ebx

The `@use ebx` option tells HLA that it can freely use the EBX, BX, BL, and BH registers when generating the code to pass parameters to this procedure. Here's the (MASM-syntax) code HLA generates when you allow it to use the EBX register in this capacity:

```
mov     bl, b_hla_2
push   ebx
call   byteParm_hla_1
push   eax
call   byteParm_hla_1
sub    esp, 4
mov    [esp], ah
call   byteParm_hla_1
mov    bl, byte ptr [eax]
push   ebx
call   byteParm_hla_1
```

HLA Generated Code for the Above Calls to byteParm

As you can see, the code is much better than before (not quite as good since it still doesn't assume it can push `b` directly onto the stack, but much better nonetheless). Of course, if you want to take absolute control, you can always push the parameter manually.

Of course, the stack isn't the most efficient place to pass parameters. The x86 registers are the best place to pass parameters (subject to the constraint that they fit in the registers). Note that HLA will allow you to pass parameters in register using a high level calling syntax as follows:

```
procedure parmsInRegs ( a:dword in eax; var b:byte in ebx );
.
.
.
```

There's nothing stopping you in HLA from simply loading a register with some value prior to a call and referencing that register inside the procedure without declaring any formal parameters. The nice thing about using the high level declaration and calling syntax is that HLA will automatically move the value into the register for you if you specify an actual parameter other than the register for that parameter. However, since there's not much in the way of bloat here, there's really no sense in discussing it farther in this document. See the HLA reference manual for more details.

Reference parameters have their own special problems. As long as you're passing a non-indexed static address (that is, the address of a **static**, **readonly**, or **storage**) object by reference, HLA generates good code (a single push instruction). However, once you throw in an index register or specify an automatic variable (whose offsets are indexed off EBP), HLA has to emit an LEA instruction to compute the effective address of the operand. Since the `lea` instruction requires a register, we're back to the same problem we had with the byte-sized operand earlier. Well, the solution is the same: if you want decent code, either pass the address manually or specify an `@use` procedure option to tell HLA that it can use a register for computing effective addresses.

HLA supports several other parameter passing mechanisms. This document won't cover them for two reasons: (1) 99% of the assembly language programmers out there have probably never heard of these parameter passing mechanisms, and (2) the 1% of them who have, know that they're usually inefficient anyway (and fast/short code avoids them like the plague).

15.9.6 Bloat in the HLA Standard Library

If you want to understand the purpose of every byte in your HLA programs you don't call HLA Standard Library routines. It's not that they're incredibly poorly written, but they're "black boxes" and unless you sit down and study their source code, you have no idea what (private) data they declare, what routines they call, or anything else about their efficiency.

The HLA Standard Library routines were not written to be the fastest nor the shortest examples of HLA code. They were written to be easy to read, understand, and maintain. Furthermore, many of the routines build upon other routines. A classic example is the *stdout.puti8* routine. This procedure takes a single byte parameter. It calls the *conv.i8ToStr* procedure to convert the value to a string, and then calls the *fileio.puts* function to actually print the string (specifying the standard output file handle as the "file"). The *conv.i8ToStr* function zero extends the eight-bit value to 16 bits and calls the *conv.i16ToStr* function. The *conv.i16ToStr* function zero extends its 16-bit value to 32 bits and calls the *conv.i32ToStr* function. The *conv.i32ToStr* function converts its 32-bit value (include 24 bits of zeros at this point) to a string and the chain of calls pass the string back to the original call from *stdout.puti8*. Each of these routines (except *conv.i32ToStr*, which does all the real work) is very short and trivial. If you program winds up calling all of these routines, this is probably the most compact representation you could devise. However, this obviously requires a lot more code than had the standard library simply provided a *conv.i8ToStr* function that did the conversion directly. Furthermore, all those extra calls, plus the fact that converting a 32-bit value to a string is more expensive than converting an eight-bit value to a string, means the code is going to run a bit slower. Therefore, if speed and/or space are prime considerations in your program, avoid the HLA Standard Library (or, always start with the source code to the routine you want to call and clean it up so avoid long call chains like the one in the above example).

There is another source of bloat that is indirectly related to the HLA Standard Library. The HLA Standard Library was modeled after the standard libraries found in C, C++, and other high level languages. As a result, calling these library routines causes you to "think" like a C programmer. As any expert assembly programmer can tell you, "thinking in assembly" is the only way to write efficient assembly programs. Even if all the routines in the HLA Standard Library were written as efficiently as possible, the mindset they leave you in is not conducive to writing efficient code. Therefore, take care when using the HLA Standard Library because it can cause you to write sloppy code if you're not carefully considering what you're doing at each step in your code.

15.9.7 Taking Control with HLA Units

Reading the HLA reference manual, you might get the impression that HLA applications are written as **programs** and separately compiled modules that you link with HLA or applications in other languages are written using **units**. HLA units are actually a bit more flexible than this, if you're willing to play some games. In particular, HLA units can completely free you from the yoke of HLA compiler-generated code and give you an environment where the only instructions that appear in your executable file are those instructions you write. This section will describe how to use HLA units to achieve this.

Fundamentally, there are only a couple of differences between HLA units and HLA programs. HLA programs allow you to declare automatic variables in a global **var** section, units do not¹. The major difference, of course, is that HLA units don't have a "main program" associated with them as HLA programs do. If you look at the code that HLA generates for units and programs, you see only a couple of differences between the output files. Specifically, HLA collects all the code from the main program and creates a procedure named *_HLAMain* (*_start*). In addition, HLA emits some support code to initialize the exception handling system for programs, none of this code appears in the assembly output file for a unit. Other than these two issues, HLA units and programs are semantically equivalent.

To prove this point, the following is an HLA unit that compiles to the same exact code as the standard Hello World program.

```
unit unitAsPgm;
#include( "stdout.hhf" )
```

1. Which makes sense because VAR objects are always associated with a procedure or the HLA main program. In a unit, there is no main program with which you can associate automatic variables.

```

?@nodisplay := true;
?@noframe := true;

// Make these names public so the library routines
// and linker can find them.

procedure _HLAMain; @external;
procedure HWexcept__hla_; @external;
procedure DfltExHndlr__hla_; @external;

// The following are HLA Standard Library procedures.
// Just make 'em labels rather than procs because we
// just JMP to these labels.

label
    shorthwExcept__hla_; @external;
    shortDfltExcept__hla_; @external;
    BuildExcepts__hla_; @external;
    QuitMain; @external( "QuitMain__hla_" );

static

    // The following is the link to the Win32 API ExitProcess procedure
    // address.

    __imp__ExitProcess :dword; @external( "__imp__ExitProcess@4" );

    // The main program needs a coroutine object for
    // use by the exception handling subsystem:

    MainPgmCoroutine__hla_ : dword; @external;
    MainPgmCoroutine__hla_ : dword; @nostorage;
        dword &MainPgmVMT__hla_;
        dword 0,0,0,0;

    MainPgmVMT__hla_ : dword := &QuitMain;

// The following are needed to provide linkage to
// the HLA exception handling routines.

procedure HWexcept__hla_;
begin HWexcept__hla_;
    jmp shorthwExcept__hla_;
end HWexcept__hla_;

procedure DfltExHndlr__hla_;

```

```
begin DfltExHndlr__hla_;
    jmp shortDfltExcept__hla_;
end DfltExHndlr__hla_;

procedure _HLAMain;
begin _HLAMain;

    call    BuildExcepts__hla_;
    pushd( 0 );           // no dynamic link (previous proc's EBP).
    mov( esp, ebp );     // Set up our stack frame.
    push( ebp );         // Main's display.

    // << put main program code here >>

    stdout.put( "Hello World" nl );

end _HLAMain;

// Fall through from the above and return to Windows.
// (this needs to be outside _HLAMain because QuitMain__hla_
// needs to be a public name).

procedure QuitMain;
begin QuitMain;

    pushd(0);
    call( __imp__ExitProcess );

end QuitMain;

end unitAsPgm;
```

Hello World Program Written as a Unit

The HLA compiler instructs the linker to start program execution at the label *_HLAMain*. By writing a procedure named *_HLAMain* and making this name public (via the **external** directive), this unit provides an HLA "main program" that the OS will invoke immediately after loading the program into memory. This main program explicitly contains the instructions that the HLA compiler would normally emit for a program (the call to *BuildExcepts__HLA_* and setting up the activation record). Following the initialization code is the invocation of the `stdout.put` macro that prints "Hello World" to the standard output. One unusual feature of this code is that the *QuitMain* label has to be global and public (i.e., we can't simply put the code that returns to Windows inside the *_HLAMain* procedure because external code references this label and you can't reference local labels from outside a procedure). The alternative would be to duplicate the code, but then we wouldn't have the semantic equivalent of the original Hello World program. If you compare the assembly output of this code with the assembly output of the standard Hello World program, you'll find that the code is nearly identical (about the only real difference is the extra procedure surrounding the code that returns to the OS; of course, this does not change the executable file one byte).

Of course, it doesn't make any sense to simply duplicate the effects of an HLA program within a unit (other than to prove it can be done). The real reason for using units in this fashion is to gain complete control over the code appearing in the executable file. Specifically, I'm assuming you want to dump some of the initialization code, data structures, and support code that exist primarily for the benefit of the HLA run-time system and exception handling subsystem. Here's the bottom line, if you want to take full responsibility for all the code appearing in your HLA program, write it as a unit and create an `_HLAMain` procedure to serve as your main program (note: Linux users need to name their main program `_start`). Here's the template you should use:

```
unit barebones;

?@nodisplay := true;
?@noframe := true;

procedure _HLAMain; @external;

procedure _HLAMain;
begin _HLAMain;

    // Put the code for your main program here.

end _HLAMain;

end barebones;
```

Bare Bones HLA Program Implemented via a Unit

If you write your code using this "barebones" unit as a template, you're going to be in complete control of the code in your program. Do keep in mind that unless you initialize the exception handling system using the code given earlier (*BuildExcepts_HLA*, etc.), you'll not be able to use HLA exceptions and that pretty much means you can't call any HLA Standard Library routines (since a large percentage of those can raise an exception). However, you will have completely escaped HLA's interference with your code and the only machine instructions that will find their way into your programs are the ones you write (or the code associated with any external routines you call).

I've made a big deal about using HLA units to give you complete control over the code HLA emits. Throughout this document, I've given the impression that only hard-core, die-hard, macho, assembly language programmers would want to do this. Actually, there are many real-world applications where the code that HLA emits for programs would be inappropriate. A classic example is the need to write dynamic link libraries. Such code has to be implemented as a **unit**, you cannot use an HLA procedure for such code.

15.9.8 Hello World, Revisited

This document lamented about the size of a typical Hello World program and mentioned that it's possible to write a shorter version of the program using HLA. In this section, we'll explore how to write a short version of this program. Actually, let's forget exploring and jump right into things.

Based on what I've said about the HLA Standard Library, it should come as no surprise that the smallest Hello World program is not going to call any Standard Library routines. The most compact Hello World program is going to make direct OS API calls. Well, without further ado, here are the compact versions of the Hello World program for Windows and Linux (different versions are necessary since the OS APIs are different).

```
unit HelloWorld;
```

```

?@noframe := true;

procedure main; @external( "_HLAMain" );

static

    WriteFile:procedure
    (
        Handle:        dword;
        var buffer:    var;
        len:           dword;
        var bytesWritten: dword;
        overlapped:    dword
    );
    @use edx;
    @stdcall;
    @external( "__imp__WriteFile@20" );

    GetStdHandle:procedure
    (
        WhichHandle:int32
    );
    @stdcall;
    @external( "__imp__GetStdHandle@4" );

    ExitProcess:procedure( exitcode:dword );
    @stdcall;
    @external( "__imp__ExitProcess@4" );

procedure main;
var
    BytesWritten    :dword;
begin main;

    GetStdHandle( -11 );
    WriteFile( eax, &hwString, 13, BytesWritten, 0 );
    ExitProcess( 0 );

    hwString:    byte    "Hello World", $d, $a;

end main;

end HelloWorld;

```

Windows Version of the Short Hello World Program

Here's the Linux version of this program:

```
unit hw;
```

```
procedure main; @external( "_start" );

procedure main; @noframe;
begin main;

    // Print Hello World:

    mov( 4, eax );
    mov( 1, ebx );
    lea( ecx, helloWorld );
    mov( 12, edx );
    int( $80 );

    // return to Linux:

    mov( 1, eax );
    mov( 0, ebx );
    int( $80 );

    helloWorld: byte "Hello World", $a;

end main;

end hw;
```

Linux Version of the Short Hello World Program

16 The HLA Memory Model and Memory Addressing Modes

This chapter describes how HLA views memory at run time and how individual instructions can access memory.

16.1 The HLA Memory Model

HLA uses a variant of the standard *a.out* memory model. The *a.out* memory model organizes data in an executable file into three distinct segments (or sections) and organizes run-time memory allocation into five distinct sections. Though it is possible to create fancier memory models, the *a.out* memory model is a time-proven tried-and-true memory model that supports the creation of almost any imaginable executable file.

During compilation, HLA organizes the object code it produces into one of four sections: a *code* (or *text*) section, a *readonly* data section, a static (initialized) *data* section, and a static (uninitialized) *bss* section. (*bss* is an old assembly language term that stands for Block Started by Symbol; the modern meaning of this term is any block of variables that aren't given a non-zero initial value when loaded into memory.) When HLA writes the object file to disk, it combines the *readonly* and *text/code* sections into a single section in the object file (the *text* section); therefore, there are only three sections of interest in the object code file. The object file might also contain other data such as symbol tables, string tables, and relocation tables, but such data is not present in the run-time code and is of no interest to us here.

When the operating system loads an HLA executable file into memory, it loads the *text*, *data*, and *bss* sections into their respective regions in memory before transferring control to the main program (which will be in the *text* section). In addition to these three sections that exist in the executable file's disk image, an HLA program generally references two other sections of memory at run time: the *stack* area and the *heap*. These two areas are created dynamically at run time by the operating system and the HLA run-time system.

The *text* and the *data* sections in memory correspond almost byte-for-byte with their respective sections in the executable disk file. Indeed, the only difference between the sections in the disk image and the sections in memory at run time is that the run-time image may have been *relocated* to a new address.

The *bss* section disk image doesn't contain any actual data. This is just a data structure in the disk image that tells the operating system how much storage to set aside when loading the executable into memory. The operating system will allocate the storage, fill it with zero bytes, and then adjust all the addresses in the *text* and *data* sections that reference the *bss* section. The size of the *bss* section is solely determined by the number of bytes of variables declared in the **storage** declaration sections of the HLA program.

Different operating systems arrange the *text*, *data*, and *bss* sections differently in memory; however, all of the data in one such section usually resides in one contiguous block of run-time memory. Multi-threaded applications can have multiple stacks, but the program generally starts with one stack section. The number of heap sections in memory depends entirely on how the operating system implements memory allocation.

You should realize that the HLA *text/code* section may contain data as well as machine instructions. Data you declare in an HLA **readonly** section and any necessary constants (such as string constants that HLA generates) are merged in with the machine instructions in the *text* section.

16.2 Memory Addressing Modes in HLA

HLA supports all the 32-bit addressing modes of the Intel 80x86 instruction set¹. A memory address on the 80x86 may consist of one to three different components: a displacement (also called

1. It does not support the 16-bit addressing modes since these are not very useful under Win32 or Linux.

an offset), a base pointer, and a scaled index value. The following are the legal combinations of these components:

```
displacement
basePointer
displacement + basePointer
displacement + scaledIndex
basePointer + scaledIndex
displacement + basePointer + scaledIndex
```

The following addressing modes are legal, but are mainly useful only within an **lea** instruction:

```
scaledIndex
scaledIndex + displacement
```

HLA's syntax for memory addressing modes takes the following forms:

```
staticVarName

staticVarName [ constant ]

staticVarName[ breg32 ]
staticVarName[ ireg32 ]
staticVarName[ ireg32*index ]

staticVarName[ breg32 + ireg32 ]
staticVarName[ breg32 + ireg32*index ]

staticVarName[ breg32 + constant ]
staticVarName[ ireg32 + constant ]

staticVarName[ ireg32*index + constant ]

staticVarName[ breg32 + ireg32 + constant ]
staticVarName[ breg32 + ireg32*index + constant ]

staticVarName[ breg32 - constant ]
staticVarName[ ireg32 - constant ]
staticVarName[ ireg32*index - constant ]

staticVarName[ breg32 + ireg32 - constant ]
staticVarName[ breg32 + ireg32*index - constant ]

localVarName

localVarName [ constant ]

localVarName[ ireg32 ]
localVarName[ ireg32*index ]
```



```

localVarName[ ireg32 + constant ]
localVarName[ ireg32*index + constant ]

localVarName[ ireg32 - constant ]
localVarName[ ireg32*index - constant ]

basereg:globalVarName

basereg:globalVarName [ constant ]

basereg::globalVarName[ ireg32 ]
basereg::globalVarName[ ireg32*index ]

basereg::globalVarName[ ireg32 + constant ]
basereg::globalVarName[ ireg32*index + constant ]

basereg::globalVarName[ ireg32 - constant ]
basereg::globalVarName[ ireg32*index - constant ]

[ breg32 ]

[ breg32 + ireg32 ]
[ breg32 + ireg32*index ]

[ breg32 + constant ]

[ breg32 + ireg32 + constant ]
[ breg32 + ireg32*index + constant ]

[ breg32 - constant ]

[ breg32 + ireg32 - constant ]
[ breg32 + ireg32*index - constant ]

```

The following are legal, but are only useful within the **lea** instruction:

```

[ ireg32*index ]
[ ireg32*index + constant ]

```

"staticVarName" denotes any static variable currently in scope (local or global).

"localVarName" denotes a local, automatic, variable declared in the **var** section of the current procedure.

"basereg" denotes any general purpose 32-bit register.

"globalVarname" denotes a non-local variable declared in the **var** section of some procedure other than the current procedure.

"breg₃₂" denotes a base register and can be any general purpose 32-bit register.

"ireg₃₂" denotes an index register and may also be any general purpose register except ESP, even the same register as the base register in the address expression.

"index" denotes one of the four constants "1", "2", "4", or "8". In those address expression that have an index register without an index constant, "*1" is the default index.

Those memory-addressing modes that do not have a variable name preceding them are known as "anonymous memory locations." Anonymous memory locations do not have a data type associated with them and in many instances you must use the type coercion operator in order to keep HLA happy.

Those memory addressing modes that do have a variable name attached to them inherit the base type of the variable. Read the next section for more details on data typing in HLA.

HLA allows another way to specify addition of the various addressing mode components in an address expression - by putting the components in separate brackets and concatenating them together. The following examples demonstrate the standard syntax and the alternate syntax:

```
[ebx+2]                [ebx] [2]
[ebx+ecx*4+8]         [ebx] [ecx*4] [8]
lbl [ebp-2]           lbl [ebp] [-2]
[ ebx*8 + 5 ]         [ebx*8] [5]
```

The reason for allowing the extended syntax is because you might want to construct these addressing modes inside a macro from the individual pieces and it's much easier to concatenate two operands already surrounded by brackets than it is to pick the expressions apart and construct the standard addressing mode.

In general, the extended syntax takes one of the following forms (braces surround optional items):

```
[ constExpr ] { <<additional address items inside "[]">> }

[ base32 ] { <<additional address items inside "[]">> }

[ index32*1 ] { <<additional address items inside "[]">> }
[ index32*2 ] { <<additional address items inside "[]">> }
[ index32*4 ] { <<additional address items inside "[]">> }
[ index32*8 ] { <<additional address items inside "[]">> }

[ base32+index32 ] { <<additional address items inside "[]">> }
[ base32+index32*1 ] { <<additional address items inside "[]">> }
[ base32+index32*2 ] { <<additional address items inside "[]">> }
[ base32+index32*4 ] { <<additional address items inside "[]">> }
[ base32+index32*8 ] { <<additional address items inside "[]">> }

[ base32 + constExpr ] { <<additional address items inside "[]">> }

[ index32*1 + constExpr ] { <<additional address items inside "[]">> }
[ index32*2 + constExpr ] { <<additional address items inside "[]">> }
[ index32*4 + constExpr ] { <<additional address items inside "[]">> }
[ index32*8 + constExpr ] { <<additional address items inside "[]">> }

[ base32+index32 + constExpr ] { <<additional address items inside "[]">> }
[ base32+index32*1 + constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*2 + constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*4 + constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*8 + constExpr ] { <<additional address items inside "[]">> }
}
```

```

[ base32 - constExpr ] { <<additional address items inside "[]">> }

[ index32*1 - constExpr ] { <<additional address items inside "[]">> }
[ index32*2 - constExpr ] { <<additional address items inside "[]">> }
[ index32*4 - constExpr ] { <<additional address items inside "[]">> }
[ index32*8 - constExpr ] { <<additional address items inside "[]">> }

[ base32+index32 - constExpr ] { <<additional address items inside "[]">> }
[ base32+index32*1 - constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*2 - constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*4 - constExpr ] { <<additional address items inside "[]">> }
}
[ base32+index32*8 - constExpr ] { <<additional address items inside "[]">> }
}

```

The major restriction is that there can be at most one base register (EAX, EBX, ECX, EDX, ESI, EDI, EBP, or ESP) and at most one index register (EAX, EBX, ECX, EDX, ESI, EDI, or EBP) in the address item. An optional static object name (**static**, **readonly**, or **storage** variable) or automatic variable name (**var** objects) may precede the address item list; however, keep in mind that if an automatic variable name precedes one of these bracketed expression lists, then the EBP register (or a user-defined register if the *reg₃₂::identifier* syntax is used) is already used as the base register. Here are some examples of legal addressing modes in HLA:

```

staticVar[ebx][ecx*4][4]
localVar[edi*2]
localVar[8][edx*8]
[ebx][edx+2]

```

Note that if you specify two 32-bit registers in an address expression without specifying an explicit scaled index value (e.g., "[ebx+ecx]") then HLA gets to choose which register is the base register and which is the index register (either choice will produce the correct effective address).

Any number of constant expressions inside brackets may appear in an extended address expression. HLA computes the sum of all such constant expressions and uses that sum as the single constant value. E.g.,

```

localVar[8][edx*8][2] -- equivalent to -- localVar[edx*8 + 10]

```

16.3 Type Coercion in HLA

While an assembly language can never really be a strongly typed language, HLA is much more strongly typed than most other assembly languages.

Strong typing in an assembly language can be very frustrating. Therefore, HLA makes certain concessions to prevent the type system from interfering with the typical assembly language programmer. Within an 80x86 machine instruction, the only checking that takes place is verification that the sizes of the operands are compatible.

Despite HLA playing fast and loose with machine instructions, there are many times when you will need to coerce the type of some operand. HLA uses the following syntax to coerce the type of a memory location or register operand:

```

(type typeID memOrRegOperand)

```

There are two instances where type coercion is especially important: (1) when you need to assign a type other than **byte**, **word**, **dword**, **qword**, or **lword** to a register¹; (2) when you need to assign an anonymous memory location a type. Here are a couple of examples:

```
if( (type int32 eax) < 0 then
    inc( (type dword [ebx] ) );
endif;
```

Type coercion is very useful in HLA when manipulating pointer objects, especially pointers to classes and records. Consider the following example:

```
type
    myRec_t: record
        i:int32;
        c:char;
    endrecord;

    mrPtr_t: pointer to myRec_t;

static
    mpr: mrPtr_t;
    .
    .
    .
    malloc( @size( myRec_t ) );
    mov( eax, mpr );
    .
    .
    .
    mov( mpr, ebx );
    mov( cl, (type myRec_t [ebx]).c );
    mov( 0, (type myRec_t [ebx]).i );
```

As you can see here, whatever memory address appears inside the parentheses is treated like an object of the specified type. So you can treat that whole entity as though it were a variable of the specified type (`myRec_t` in this example) and you can apply the dot operator or any other operation that would be legal on a variable of that type.

By default, the x86 general-purpose registers have the types **byte**, **word**, or **dword** (depending, of course, on their size). Sometimes you might want to coerce these registers to a different type, especially when outputting the value of a register or comparing a register with a constant. Coercion of a register is legal as long as the coerced data type is the same size as the register, e.g.,

```
(type int32 eax)
```

Coercion like this last example is especially useful when using the register without an output statement (like `stdout.put`) or in a run-time boolean expression. Consider the following:

```
if( eax < 0 ) then
    << do something if EAX is negative>>
endif;
```

1. Probably the most common case is treating a register as a signed integer in one of HLA's high level language statements. See the section on HLA High Level Language statements for more details.

In this example, the expression is always false because EAX is a **dword** object (which is unsigned). Therefore, EAX can never be less than zero (even if EAX contains something that you want interpreted as a negative value). You can solve this problem by coerce EAX to an **int32** object:

```
if( (type int32 eax) < 0 ) then
  << do something if EAX is negative>>
endif;
```

This code example will work properly since HLA is smart enough to generate the appropriate signed comparison/conditional jump sequence when it realizes one or more of the operands are signed.

Type coercion fully supports HLA memory addressing modes. You can use any valid HLA addressing mode form in place of the address object in the type coercion expression, for example:

```
(type dword byteVar[ebx][ecx*1][2] )
```

In addition, you can also treat a type coercion operation as though it were a static identifier in an extended HLA addressing mode; that is, you can follow a type coercion operator with a set of bracketed addressing mode options:

```
(type qword [ebx] ) [ecx*8] [16]
```

17 HLA v2.x Language Reference Manual

17.1 The 80x86 Instruction Set in HLA

One of the most obvious differences between HLA and standard 80x86 assembly language is the syntax for the machine instructions. The two primary differences are the fact that HLA uses a functional notation for machine instructions and HLA arranges the operands in a (source, dest) format rather than the (dest, source) format used by Intel.

A second difference, related to the fact that HLA uses a functional notation, is that HLA allows you to *compose* instructions. That is, one instruction may appear as an operand to a second instruction, e.g.,

```
mov( mov( 0, eax ), ebx );
```

To decipher this instruction, all you need to do is to realize that at compile time each instruction returns a string that HLA substitutes in place of the composed instruction. Usually, the string an instruction returns is that instruction's destination operand. In the example above, the interior `mov` instruction's destination operand is `EAX`, so that `mov` instruction "returns" the string "EAX" which HLA substitutes for the interior `mov` instruction, producing "`mov(eax, ebx);`" as the outside instruction. HLA always processes interior instructions from left-to-right interior-first. Therefore, the above instruction is equivalent to the MASM sequence:

```
mov    eax, 0
mov    ebx, eax
```

Consider a second example:

```
add( mov( i, eax ), mov( j, ebx ) );
```

This instruction is equivalent to:

```
mov    eax, i
mov    ebx, j
add    ebx, eax
```

Although, used sparingly, instruction composition is useful and can help improve the readability of your HLA programs in certain contexts, you should be careful when using instruction composition because it can quickly produce unreadable code. Even this second example (`add(mov,mov)`) would probably prove difficult to read by most programmers.

If you need to modify the RETURNS value of an instruction (in a macro, for example), you may use the "returns" statement in HLA. This statement takes the following form:

```
returns( { statements }, "string Constant" )
```

This statement emits the code for the statement(s) between the curly braces and then returns the specified string constant as the "returns" value for this statement.

The following paragraphs describe each of the HLA machine instructions. They also describe the string each instruction yields during compile time (this is called the "returns" string). Note that some instructions return the empty string as there is no return value one could reasonably associate with them. Such instructions cannot generally be used as operands within other instructions.

These descriptions do not describe the purpose for each instruction; see an assembly text like "The Art of Assembly Language Programming" for details on the operation of each instruction.

17.2 Zero Operand Instructions (Null Operand Instructions)

Instruction	Description
aaa()	ASCII adjust for addition. Returns "ax".
aad()	ASCII adjust for division. Returns "ax".
aam()	ASCII adjust for multiplication. Returns "ax".
aas()	ASCII adjust for subtraction. Returns "ax".
cbw()	Convert byte to word (sign extension). Returns "ax"
cdq()	Convert double to quadword. Returns "eax". Note: in the future, this may return "edx:eax".
clc()	Clear carry flag. Returns "".
cld()	Clear direction flag. Returns "".
cli()	Clear interrupt flag. Returns "".
clts()	Clear task switched flag in CR0 (OS use only).
cmc()	Complement carry flag. Returns "".
cmpsb()	Compares the byte at [esi] to the byte at [edi] and increments or decrements ESI & EDI by one. Returns "".
cmpsd()	Compares the dword at [esi] to the byte at [edi] and increments or decrements ESI & EDI by four. Returns "".
cmpsw()	Compares the word at [esi] to the byte at [edi] and increments or decrements ESI & EDI by two. Returns "".
cpuid()	On entry, EAX contains zero, one, or two to determine how this instruction behaves. If EAX contains zero then this instruction returns vendor information in EAX, EBX, ECX, and EDX. If EAX contains one upon entry, EAX returns with version information and EDX contains feature information. If EAX contains two upon entry, EAX..EDX return with cache information. See the Intel documentation for more details concerning this instruction.
cwd()	Convert word to doubleword. Returns "ax". Note: in the future, this may return "dx:ax".
cwde()	Convert word to dword, extended. Returns "eax".
daa()	Decimal adjust for addition. Returns "al".
das()	Decimal adjust for subtraction. Returns "al".
hlt()	Halt instruction (OS and embedded use only).
insb()	Inputs a byte from the port specified by DX and stores the byte at [EDI], then increments or decrements EDI by one. Returns "".

insd()	Inputs a dword from the port specified by DX and stores the dword at [EDI], then increments or decrements EDI by four. Returns "".
insw()	Inputs a word from the port specified by DX and stores the word at [EDI], then increments or decrements EDI by two. Returns "".
into()	Interrupt on overflow. Returns "". Raises the ex.IntoInstr exception if the overflow flag is set when you execute this instruction.
invd()	Invalidate internal caches (OS use only).
iret()	Interrupt return. Returns "".
iretd()	Interrupt return popping 32-bit flags. Returns "".
lahf()	Load AH from flags. Returns "al".
leave()	Remove activation record from stack. Returns "".
lodsb()	Load al from [ESI] and increment ESI by one. Returns "al".
lodsd()	Load eax from [ESI] and increment ESI by four. Returns "eax".
lodsw()	Load ax from [ESI] and increment ESI by two. Returns "ax".
movsb()	Moves a byte from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by one. Returns "".
movsd()	Moves a dword from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by four. Returns "".
movsw()	Moves a word from the location specified by [ESI] to the location specified by [EDI], then increments or decrements ESI & EDI by two. Returns "".
nop()	No operation. Returns "".
outsb()	Outputs the byte at address [ESI] to the port specified by DX, then increments or decrements ESI by one. Returns "".
outsd()	Outputs the dword at address [ESI] to the port specified by DX, then increments or decrements ESI by four. Returns "".
outsw()	Outputs the word at address [ESI] to the port specified by DX, then increments or decrements ESI by two. Returns "".
popad()	Pop all general purpose 32-bit registers from stack. Returns "".
popa()	Pop all general purpose 16-bit registers from stack. Returns "".
popf()	Pop 16-bit flags register from stack. Returns "".
popfd()	Pop 32-bit flags register from stack. Returns "".
pusha()	Push all general-purpose 16-bit registers onto the stack. Returns "".
pushad()	Push all general-purpose 32-bit registers onto the stack. Returns "".
pushf()	Push 16-bit flags register onto the stack. Returns "".
pushfd()	Push 32-bit flags register onto the stack. Returns "".

rdmsr()	Read from model specific register specified by ECX into EDX:EAX (OS use only).
rdpmc()	Read performance monitoring counter specified by ECX into EDX:EAX (OS use only).
rdtsc()	Reads the "time stamp" counter and returns the 64-bit result in edx:eax.
rep.insb()	Transfers ECX bytes from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by one after each transfer. Returns "".
rep.insd()	Transfers ECX dwords from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by four after each transfer. Returns "".
rep.insw()	Transfers ECX words from the port specified by DX to the location specified by [EDI]. Increments or decrements EDI by two after each transfer. Returns "".
rep.movsb()	Copies ECX bytes from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by one after each transfer. Returns "".
rep.movsd()	Copies ECX dwords from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by four after each transfer. Returns "".
rep.movsw()	Copies ECX words from the memory location specified by [ESI] to the location specified by [EDI]. Increments or decrements EDI & ESI by two after each transfer. Returns "".
rep.outsb()	Transfers ECX bytes from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by one after each transfer. Returns "".
rep.outsd()	Transfers ECX dwords from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by four after each transfer. Returns "".
rep.outsw()	Transfers ECX words from the location specified by [ESI] to the port specified by DX. Increments or decrements EDI by two after each transfer. Returns "".
rep.stosb()	Copies CX bytes from AL to the location specified by [EDI]. Increments or decrements EDI by one after each transfer. Returns "".
rep.stosd()	Copies ECX dwords from EAX to the location specified by [EDI]. Increments or decrements EDI by four after each transfer. Returns "".
rep.stosw()	Copies ECX words from AX to the location specified by [EDI]. Increments or decrements EDI by two after each transfer. Returns "".
repe.cmpsb()	Compares ECX bytes starting at location [ESI] to the set of bytes at location [EDI] as long as the bytes are equal. The comparison stops once two unequal bytes are found. After each successful compare, this instruction increments or decrements ESI and EDI by one (and decrements ECX). Returns "".
repe.cmpsd()	Compares ECX dwords starting at location [ESI] to the set of dwords at location [EDI] as long as the dwords are equal. The comparison stops once two unequal dwords are found. After each successful compare, this instruction increments or decrements ESI and EDI by four (and decrements ECX). Returns "".
repe.cmpsw()	Compares ECX words starting at location [ESI] to the set of words at location [EDI] as long as the words are equal. The comparison stops once two unequal words are found. After each successful compare, this instruction increments or decrements ESI and EDI by two (and decrements ECX). Returns "".

repe.scasb()	Compares AL against ECX bytes starting at location [EDI] as long as the bytes are equal. The comparison stops once two unequal bytes are found. After each successful compare, this instruction increments or decrements EDI by one (and decrements ECX). Returns "".
repe.scasd()	Compares EAX against ECX dwords starting at location [EDI] as long as the dwords are equal. The comparison stops once two unequal dwords are found. After each successful compare, this instruction increments or decrements EDI by four (and decrements ECX). Returns "".
repe.scasw()	Compares AX against ECX words starting at location [EDI] as long as the words are equal. The comparison stops once two unequal words are found. After each successful compare, this instruction increments or decrements EDI by two (and decrements ECX). Returns "".
repne.cmpsb()	Compares ECX bytes starting at location [ESI] to the set of bytes at location [EDI] as long as the bytes are not equal. The comparison stops once two equal bytes are found. After each successful compare, this instruction increments or decrements ESI and EDI by one (and decrements ECX). Returns "".
repne.cmpsd()	Compares ECX dwords starting at location [ESI] to the set of dwords at location [EDI] as long as the dwords are not equal. The comparison stops once two equal dwords are found. After each successful compare, this instruction increments or decrements ESI and EDI by four (and decrements ECX). Returns "".
repne.cmpsw()	Compares ECX words starting at location [ESI] to the set of words at location [EDI] as long as the words are not equal. The comparison stops once two equal words are found. After each successful compare, this instruction increments or decrements ESI and EDI by two (and decrements ECX). Returns "".
repne.scasb()	Compares AL against ECX bytes starting at location [EDI] as long as the bytes are not equal. The comparison stops once two equal bytes are found. After each successful compare, this instruction increments or decrements EDI by one (and decrements ECX). Returns "".
repne.scasd()	Compares EAX against ECX dwords starting at location [EDI] as long as the dwords are not equal. The comparison stops once two equal dwords are found. After each successful compare, this instruction increments or decrements EDI by four (and decrements ECX). Returns "".
repne.scasw()	Compares AX against ECX words starting at location [EDI] as long as the words are not equal. The comparison stops once two equal words are found. After each successful compare, this instruction increments or decrements EDI by two (and decrements ECX). Returns "".
rsm()	Resume from system management mode (OS use only).
sahf()	Store AH into the flags register. Returns "ah".
scasb()	Compares the byte in al to the location specified by [EDI], then increments or decrements EDI by one. Returns "".
scasd()	Compares the dword in eax to the location specified by [EDI], then increments or decrements EDI by four. Returns "".
scasw()	Compares the word in ax to the location specified by [EDI], then increments or decrements EDI by two. Returns "".

stc()	Set the carry flag. Returns "".
std()	Set the direction flag. Returns "".
sti()	Set the interrupt flag. Returns "".
stosb()	Stores the byte in al to the location specified by [EDI], then increments or decrements EDI by one. Returns "".
stosd()	Stores the dword in eax to the location specified by [EDI], then increments or decrements EDI by four. Returns "".
stosw()	Stores the word in ax to the location specified by [EDI], then increments or decrements EDI by two. Returns "".
ud2()	Undefined opcode instruction. This instruction always raises an undefine opcode exception.
wbinvd()	Write back and invalidate cache (OS use only).
wait()	Coprocessor wait instruction. Returns "".
xlat()	Translate instruction. Returns "".

Note: if the NULL-Operand instructions appear as a stand-alone instruction (i.e., they are not part of an instruction composition and, thus, appear as the operand to another instruction), you can drop the "(" after the instruction as long as you terminate the instruction with a semicolon.

17.3 General Arithmetic and Logical Instructions

These instructions include `adc`, `add`, `and`, `mov`, `or`, `sbb`, `sub`, `test`, and `xor`. They all take the same basic form (substitute the appropriate mnemonic for "adc" in the syntax examples below):

Generic Form:

```
adc(source,dest);
lock.adc(source,dest);
```

Specific forms allowed:

```
adc( Reg8, Reg8 )
adc( Reg16, Reg16 )
adc( Reg32, Reg32 )

adc( const, Reg8 )
adc( const, Reg16 )
adc( const, Reg32 )

adc( const, mem )

adc( Reg8, mem )
adc( Reg16, mem )
adc( Reg32, mem )

adc( mem, Reg8 )
adc( mem, Reg16 )
adc( mem, Reg32 )
```

```

adc( Reg8, AnonMem )
adc( Reg16, AnonMem )
adc( Reg32, AnonMem )

adc( AnonMem, Reg8 )
adc( AnonMem, Reg16 )
adc( AnonMem, Reg32 )

```

Note: for the form "adc(const, mem)", if the specified memory location does not have a size or type associated with it, you must explicitly specify the size of the memory operand, e.g., "adc(5,(type byte [eax]));"

These instructions all return their destination operand as the "returns" value.

See "The Art of Assembly" for a further discussion of these instructions.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.4 The XCHG Instruction

The xchg instruction allows the following syntactical forms:

Generic Form:

```

xchg( source, dest );
lock.xchg( source, dest );

```

Specific Forms:

```

xchg( Reg8, Reg8 )
xchg( Reg8, mem )
xchg( Reg8, AnonMem)
xchg( mem, Reg8 )
xchg( AnonMem, Reg8 )

xchg( Reg16, Reg16 )
xchg( Reg16, mem )
xchg( Reg16, AnonMem)
xchg( mem, Reg16 )
xchg( AnonMem, Reg16 )

xchg( Reg32, Reg32 )
xchg( Reg32, mem )
xchg( Reg32, AnonMem)
xchg( mem, Reg32 )
xchg( AnonMem, Reg32 )

```

This instruction returns its destination operand as its "returns" value.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.5 The CMP Instruction

The "cmp" instruction uses the following general forms:

Generic:

```
cmp( LeftOperand, RightOperand );
```

Specific Forms:

```
cmp( Reg8, Reg8 );
cmp( Reg8, mem );
cmp( Reg8, AnonMem );
cmp( mem, Reg8 );
cmp( AnonMem, Reg8 );
cmp( Reg8, const );

cmp( Reg16, Reg16 );
cmp( Reg16, mem );
cmp( Reg16, AnonMem );
cmp( mem, Reg16 );
cmp( AnonMem, Reg16 );
cmp( Reg16, const );

cmp( Reg32, Reg32 );
cmp( Reg32, mem );
cmp( Reg32, AnonMem );
cmp( mem, Reg32 );
cmp( AnonMem, Reg32 );
cmp( Reg32, const );

cmp( mem, const );
```

Note that the CMP instruction's operands are ordered "dest, source" rather than the usual "source,dest" format (that is, the operands are in the same order as MASM expects them). This is to allow an intuitive use of the instruction mnemonic (that is, CMP normally reads as "compare dest to source."). We will avoid this confusion by simply referring to the operands as the "left operand" and the "right operand". Left vs. right signifies the placement of the operands around a comparison operator like "<=" (e.g., "left <= right").

For the "cmp(mem, const)" form, the memory operand must have a type or size associated with it. When using anonymous memory locations you must always coerce the type of the memory location, e.g., "cmp((type word [ebp-4]), 0);".

These instructions return their dest (first) operand as their "returns" value.

17.6 The Multiply Instructions

HLA supports several variations on the 80x86 "MUL" and IMUL instructions. The supported forms are:

Standard Syntax:

```
mul( reg8 )
mul( reg16 )
mul( reg32 )
mul( mem )

mul( reg8, al )
mul( reg16, ax )
```

```
mul( reg32, eax )

mul( mem, al )
mul( mem, ax )
mul( mem, eax )

mul( AnonMem, ax )
mul( AnonMem, dx:ax )
mul( AnonMem, edx:eax )

imul( reg8 )
imul( reg16 )
imul( reg32 )
imul( mem )

imul( reg8, al )
imul( reg16, ax )
imul( reg32, eax )

imul( mem, al )
imul( mem, ax )
imul( mem, eax )

imul( AnonMem, ax )
imul( AnonMem, dx:ax )
imul( AnonMem, edx:eax )

intmul( const, Reg16 )
intmul( const, Reg16, Reg16 )
intmul( const, mem, Reg16 )
intmul( const, AnonMem, Reg16 )

intmul( const, Reg32 )
intmul( const, Reg32, Reg32 )
intmul( const, mem, Reg32 )
intmul( const, AnonMem, Reg32 )

intmul( Reg16, Reg16 )
intmul( mem, Reg16 )
intmul( AnonMem, Reg16 )

intmul( Reg32, Reg32 )
intmul( mem, Reg32 )
intmul( AnonMem, Reg32 )
```

Extended Syntax:

```
mul( const, al )
mul( const, ax )
mul( const, eax )

imul( const, al )
imul( const, ax )
imul( const, eax )
```

The first, and probably most important, thing to note about HLA's multiply instructions is that HLA uses a different mnemonic for the extended-precision integer multiply versus the single-precision integer multiply (i.e., IMUL vs. INTMUL). Standard MASM syntax uses the same mnemonic for both instructions. There are two reasons for this change of syntax in HLA. First, there needed to be some way to differentiate the "mul(const, al)" and the "intmul(const, al)" instructions (likewise for the instructions involving AX and EAX). Second, the behavior of the INTMUL instruction is substantially different from the IMUL instruction, so it makes sense to use different mnemonics for these instructions.

The extended syntax instructions create a static data variable, initialized with the specified constant, and then specify the address of this variable as the source operand of the MUL or IMUL instruction.

These instructions return their destination operand (AX, DX:AX, or EDX:EAX for the extended precision MUL and IMUL instructions) as their "returns" value.

See "The Art of Assembly Language Programming" for more details on these instructions.

17.7 The Divide Instructions

HLA support several variations on the 80x86 DIV and IDIV instructions. The supported forms are:

Generic Forms:

```
div( source );
div( source, dest );

mod( source );
mod( source, dest );

idiv( source );
idiv( source, dest );

imod( source );
imod( source, dest );
```

Specific Forms:

```
div( reg8 )
div( reg16)
div( reg32 )
div( mem )

div( reg8, ax )
div( reg16, dx:ax)
div( reg32, edx:eax )

div( mem, ax )
div( mem, dx:ax)
div( mem, edx:eax )

div( AnonMem, ax )
div( AnonMem, dx:ax )
div( AnonMem, edx:eax )

mod( reg8 )
mod( reg16)
mod( reg32 )
```

```
mod( mem )

mod( reg8, ax )
mod( reg16, dx:ax)
mod( reg32, edx:eax )

mod( mem, ax )
mod( mem, dx:ax)
mod( mem, edx:eax )

mod( AnonMem, ax )
mod( AnonMem, dx:ax )
mod( AnonMem, edx:eax )

idiv( reg8 )
idiv( reg16)
idiv( reg32 )
idiv( mem )

idiv( reg8, ax )
idiv( reg16, dx:ax)
idiv( reg32, edx:eax )

idiv( mem, ax )
idiv( mem, dx:ax)
idiv( mem, edx:eax )

idiv( AnonMem, ax )
idiv( AnonMem, dx:ax )
idiv( AnonMem, edx:eax )

imod( reg8 )
imod( reg16)
imod( reg32 )
imod( mem )

imod( reg8, ax )
imod( reg16, dx:ax)
imod( reg32, edx:eax )

imod( mem, ax )
imod( mem, dx:ax)
imod( mem, edx:eax )

imod( AnonMem, ax )
imod( AnonMem, dx:ax )
imod( AnonMem, edx:eax )
```

Extended Syntax:

```
div( const, ax )
div( const, dx:ax )
div( const, edx:eax )
```



```

mod( const, ax )
mod( const, dx:ax )
mod( const, edx:eax )

idiv( const, ax )
idiv( const, dx:ax )
idiv( const, edx:eax )

imod( const, ax )
imod( const, dx:ax )
imod( const, edx:eax )

```

The destination operand is always implied by the 80x86 "div" and "idiv" instructions (AX, DX:AX, or EDX:EAX). HLA allows the specification of the destination operand in order to make your programs easier to read (although the use of the destination operand is optional).

The HLA divide instructions support an extended syntax that allows you to specify a constant as the divisor (source operand). HLA allocates storage in the static data segment and initializes the storage with the specified constant, and then divides the accumulator by this newly specified memory location.

The DIV and IDIV instructions return "AL", "AX", or "EAX" as their "returns" value (the quotient is left in the accumulator register). The MOD and IMOD instructions return "AH", "DX", or "EDX" as their "returns" value. Indeed, the "returns" value is the only difference between these instructions. The DIV and MOD instructions compile into the 80x86 DIV instruction; the IDIV and IMOD instructions compile into the 80x86 IDIV instruction.

See the "Art of Assembly" for a further discussion of these instructions.

17.8 Single Operand Arithmetic and Logical Instructions

These instructions include dec, inc, neg, and not. They take the following general forms (substituting the specific mnemonic as appropriate):

Generic Form:

```

dec( dest );;
lock.dec( dest );

```

Specific forms allowed:

```

dec( Reg8 );
dec( Reg16 );
dec( Reg32 );
dec( mem );

```

Note: if mem is an untyped or unsized memory location (i.e., an anonymous memory location), you must explicitly provide a size; e.g., "dec(type word [edi]);"

These instructions all return their destination operand as the "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.9 Shift and Rotate Instructions

These instructions include RCL, RCR, ROL, ROR, SAL, SAR, SHL, and SHR. These instructions support the following generic syntax, making the appropriate mnemonic substitution.

Generic Form:

```
shl ( count, dest );
```

Specific Forms:

```
shl ( const, Reg8 );
shl ( const, Reg16 );
shl ( const, Reg32 );

shl ( const, mem );

shl ( cl, Reg8 );
shl ( cl, Reg16 );
shl ( cl, Reg32 );

shl ( cl, mem );
```

The "const" operand is an unsigned integer constant between zero and the maximum number of bits in the destination operand. The forms with a memory operand must have a type or size associated with the operand; e.g., when using anonymous memory locations, you must coerce the type,

```
"shl( 2, (type dword [esi]));"
```

These instructions return their destination operand as their "returns" value. See the "Art of Assembly" for a further discussion of these instructions.

17.10 The Double Precision Shift Instructions

These instructions use the following general form (you can substitute SHRD for SHLD below):

Generic Form:

```
shld ( count, source, dest )
```

Specific Forms:

```
shld ( const, Reg16, Reg16 )
shld ( const, Reg16, mem )
shld ( const, Reg16, AnonMem )

shld ( cl, Reg16, Reg16 )
shld ( cl, Reg16, mem )
shld ( cl, Reg16, AnonMem )
```

```
shld( const, Reg32, Reg32 )
shld( const, Reg32, mem )
shld( const, Reg32, AnonMem )
```

```
shld( c1, Reg32, Reg32 )
shld( c1, Reg32, mem )
shld( c1, Reg32, AnonMem )
```

These instructions return their destination operand as the "returns" value. See the "Art of Assembly" for a further discussion of these instructions.

17.11 The Lea Instruction

These instructions use the following syntax:

```
lea( Reg32, memory )
lea( Reg32, AnonMem )
lea( Reg32, ProcID )
lea( Reg32, LabelID )
```

Extended Syntax:

```
lea( Reg32, StringConstant )
lea( Reg32, const ConstExpr )

lea( memory, Reg32 )
lea( AnonMem, Reg32 )
lea( ProcID, Reg32 )
lea( LabelID, Reg32 )
lea( StringConstant, Reg32 )
lea( const ConstExpr, Reg32 )
```

The "lea" instruction loads the specified 32-bit register with the address of the specified memory operand, procedure, or statement label. Note that in the extended syntax you can reverse the order of the operands. Since exactly one operand must be a register, there is no ambiguity between the two forms (this syntax was added to satisfy those who complained about the (reg.memory) syntax). Of course, good programming style suggests that you use only one form (either reg.memory or memory, reg) within your programs.

The extended syntax form lets you specify a constant rather than a memory address. There is no such thing as the address of a constant, but HLA will create a memory variable in the constants data segment and initialize that variable with the value of the specified memory constant and then load the address of this variable into the specified register (or push it onto the stack).

There is a subtle difference between the following two instructions:

```
lea( eax, "String" );
lea( eax, const "String" );
```

The first instruction loads EAX with the address of the first character of the literal string constant. The second form loads the EAX register with the address of a string variable (which is a pointer containing the address of the first character of the string literal).

The LEA instructions return the 32-bit register as their "returns" value.

See "The Art of Assembly" for a further discussion of the LEA instruction.

Note: HLA does not support an LEA instruction that loads a 16-bit address into a 16-bit register. That form of the LEA instruction is not very useful in 32-bit programs running on 32-bit operating systems.

17.12 The Sign and Zero Extension Instructions

The HLA MOVXX and MOVZX instructions use the following syntax:

Generic Forms:

```
movsx(source,dest);
movzx(source,dest);
```

Specific Forms:

```
movsx( Reg8, Reg16 )
movsx( Reg8, Reg32 )
movsx( Reg16, Reg32 )
movsx( mem8, Reg16 )
movsx( mem8, Reg32 )
movsx( mem16, Reg32 )

movzx( Reg8, Reg16 )
movzx( Reg8, Reg32 )
movzx( Reg16, Reg32 )
movzx( mem8, Reg16 )
movzx( mem8, Reg32 )
movzx( mem16, Reg32 )
```

These instructions sign- (MOVXX) or zero- (MOVZX) extend their source operand into the destination operand. They return their destination operand as their "returns" value.

See the "Art of Assembly" for a further discussion of these instructions.

17.13 The Push and Pop Instructions

These instructions take the following general forms:

```
pop( reg16 );
pop( reg32 );
pop( mem );

push( Reg16 )
push( Reg32 )
push( memory )

pushw( Reg16 )
pushw( memory )
pushw( AnonMem )
pushw( Const )

pushd( Reg32 )
pushd( memory )
pushd( AnonMem )
pushd( Const )
```

These instructions push or pop their specified operand. They all return their operand as their "returns" value.

17.14 Procedure Calls

HLA provides several different ways to call a procedure. Given a procedure named "MyProc", any of the following syntaxes are legal:

```
MyProc( parameter_list );
call( MyProc );
call MyProc;
```

If MyProc has a set of declared parameters, the number and types of actual parameters must match the number and types of the formal parameters. HLA will emit the code needed to push the parameter list on the stack. In the two call statements above, it is the programmer's responsibility to pass any needed parameters. For more details, see the section on procedure declarations.

In the examples above, MyProc can either be the name of an actual procedure or a procedure variable (that is a pointer to a procedure declared as "myproc:procedure(*parameters*);" in the VAR or a static section). If you need to call a procedure using an anonymous memory variable (i.e., an addressing mode like [ebx]), an untyped dword value, or via a register, you must use the syntax of the second call above, e.g., "call(ebx);". Of course, any legal HLA/80x86 address mode would be legal here.

When declaring a standard procedure, the procedure declaration syntax allows you to specify a "returns" value for that procedure, e.g.,

```
procedure MyProc; returns( "eax" );
```

HLA substitutes the string that appears as the "returns" argument for the call when using the first syntax above. For example, supposing that MyProc is a function returning its result in EAX, you could use the following to call MyProc and save the return value in the "Result" variable:

```
mov( MyProc(), Result );
```

For more details, see the section on procedure declarations.

To call a class procedure, one would use one of the following syntaxes:

```
className.ProcName( parameters );
call( className.ProcName );
call ClassName.ProcName;
```

```
objectName.ProcName( parameters );
call( objectName.ProcName );
call objectName.ProcName;
```

The difference between "className" and "objectName" is that "className" represents the actual name of the class data type whereas "objectName" represents the name of an instance of this class (i.e., a variable of type "className" declared in the VAR or a static section).

When calling a class procedure, HLA loads the ESI register with the address of the object before calling the specified procedure. Since there is no instance variable (object) associated with the className form, HLA loads ESI with zero (NULL). Inside the class procedure, you can test the value of ESI to determine if the procedure was called via the class name or an object name. This is quite useful, for example when writing constructors, to determine whether the procedure needs to allocate storage for an object. Consider the following program that demonstrates the use of an object constructor (create):

```
program demo;

#include( "memory.hhf" );
#include( "stdio.hhf" );
```

```
type
  cc: class
      var
          i: int32;

      procedure create; returns( "esi" );

  endclass;

var
  ccVar: cc;
  ccPtr: pointer to cc;

static
  ccStat: cc;

procedure cc.create; @nodisplay;
begin create;

  push( eax );
  if( esi = 0 ) then

    stdout.put( "Allocating" nl );
    malloc( @size( cc ) );
    mov( eax, esi );

  else

    stdout.put( "Already allocated" nl );

  endif;
  mov( &cc._VMT_, this._pVMT_ );
  mov( 0, this.i );
  pop( eax );

end create;

begin demo;

  // This first call to create allocates storage.

  mov( cc.create(), ccPtr );

  // In all the remaining calls, ESI is loaded with
  // the address of the object and no storage is
  // created.

  ccPtr.create();
  ccVar.create();
  ccStat.create();

end demo;
```

The `call()` statement allows any one of the following syntaxes:

```
call ProcID;
call( ProcID );
call( dwordvar );
call( anonmem );           // Addressing mode like [ebx].
call( Reg32 );
```

The second form above returns the string (if any) specified by `ProcID`'s "returns" option. The remaining `call` instructions return the empty string as their "returns" value.

You may also call an iterator procedure via the `CALL` instruction. However, it is your responsibility to set up the parameters and other state information prior to the call (see the section on iterators for more details).

17.15 The Ret Instruction

The `RET()` statement allows two syntactical forms:

```
ret( );
ret( integer_constant_expression );
```

The first form emits a simple 80x86 `RET` instruction, the second form emits the 80x86 `RET` instruction with the specified numeric constant expression value (used to remove parameters from the stack).

Normally, you would use these instructions in a procedure that has the "`@noframe`" option. Unless you know exactly what you are doing, you should never use the "`RET`" instruction inside a standard HLA procedure without this option since doing so almost always produces disastrous results. If you do use this instruction within such a procedure, it is your responsibility to deallocate local variables and the display (if any), restore `EBP`, and remove any parameters from the stack.

17.16 The Jmp Instructions

The HLA "`jmp`" instruction supports the following syntax:

```
jmp Label;
jmp ProcedureName;
jmp( dwordMemPtr );
jmp( anonMemPtr );
jmp( reg32 );
```

"`Label`" represents a statement label in the current procedure. (You are not allowed to jump to labels in other procedures in the current version of HLA. This restriction may be relaxed somewhat in future versions.) A statement label is a unique (within the current procedure) identifier with a colon after the identifier, e.g.,

```
InfiniteLoop:
    << Code inside the infinite loop >>
    jmp InfiniteLoop;
```

Jumping to a procedure transfers control to the first instruction in the specified procedure. You are responsible for explicitly pushing any parameters and the return address for that procedure.

These instructions all return the empty string as their "returns" value.

17.17 The Conditional Jump Instructions

These instructions include JA, JAE, JB, JBE, JC, JE, JG, JGE, JL, JLE, JO, JP, JPE, JPO, JS, JZ, JNA, JNAE, JNB, JNBE, JNC, JNE, JNG, JNGE, JNL, JNLE, JNO, JNP, JNS, JNZ, JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ. They all take the following generic form (substituting the appropriate instruction for "JA").

```
ja    LocalLabel;
```

"LocalLabel" must be a statement label defined in the current procedure (or a globally visible label declared in a label section or a global label defined with the "::" symbol).

These instructions all return the empty string as their "returns" value.

Note: due to the nature of the HLA compilation process, you should avoid the use of the JCXZ, JECXZ, LOOP, LOOPE, LOOPZ, LOOPNE, and LOOPNZ instructions if you are emitting assembly language source (rather than directly emitting object code). Unlike the other conditional jump instructions, these instructions have a very limited +/- 128-byte range. Unfortunately, HLA cannot detect if the branch is out of range (this task is handled by back-end assembler when producing assembly language source code), so if a range error occurs, HLA cannot warn you about this. The assembly will fail, but the result will be hard to decipher. Fortunately, these instructions are easily, and usually more efficiently, implemented using other 80x86 instructions so this should not prove to be a problem.

In a few special cases, the boolean constants "true" and "false" are legal labels. See the discussion of HLA's high-level language features for more details.

17.18 The Conditional Set Instructions

These instructions include: SETA, SETAE, SETB, SETBE, SETC, SETE, SETG, SETGE, SETL, SETLE, SETO, SETP, SETPE, SETPO, SETS, SETZ, SETNA, SETNAE, SETNB, SETNBE, SETNC, SETNE, SETNG, SETNGE, SETNL, SETNLE, SETNO, SETNP, SETNS, and SETNZ. They take the following generic forms (substituting the appropriate mnemonic for seta):

```
seta ( Reg8 )
seta ( mem )
seta ( AnonMem )
```

See the "Art of Assembly" for a further discussion of these instructions.

17.19 The Conditional Move Instructions

These instructions include CMOVA, CMOVAE, CMOVB, CMOVBE, CMOVVC, CMOVE, CMOVG, CMOVGE, CMOVL, CMOVLE, CMOVVO, CMOVVP, CMOVPO, CMOVNS, CMOVZ, CMOVNA, CMOVNAE, CMOVNB, CMOVNB, CMOVNB, CMOVNC, CMOVNE, CMOVNG, CMOVNGE, CMOVNL, CMOVNLE, CMOVNO, CMOVNP, CMOVNS, and CMOVNZ. They use the following general syntax:

```
CMOVcc ( src, dest );
```

Allowable operands:

```
CMOVcc ( reg16, reg16 );
CMOVcc ( reg32, reg32 );
CMOVcc ( mem16, reg16 );
CMOVcc ( mem32, reg32 );
```


These instructions move the data if the specified condition is true (specified by the *cc* condition). If the condition is false, these instructions behave like a no-operation.

17.20 The Input and Output Instructions

The "in" and "out" instructions use the following syntax:

```
in( port, al )
in( port, ax )
in( port, eax )

in( dx, al )
in( dx, ax )
in( dx, eax )

out( al, port )
out( ax, port )
out( eax, port )

out( al, dx )
out( ax, dx )
out( eax, dx )
```

The "port" parameter must be an unsigned integer constant in the range 0..255. The IN instructions return the accumulator register (AL, AX, or EAX) as their "returns" value. The OUT instructions return the port number (or DX) as their "returns" value.

Note that these instructions may be privileged instructions when running under Win32 or *NIX. Their use may generate a fault in certain instances or when accessing certain ports.

See the "Art of Assembly" for a further discussion of these instructions.

17.21 The Interrupt Instruction

This instruction uses the syntax "int(constant)" where the constant operand is an unsigned integer value in the range 0..255.

This instruction returns the empty string as its "returns" value.

See Chapter Six in "Art of Assembly" (DOS version) for a further discussion of this instruction. Note, however, that one generally does not use "int" under Win32 to make OS or BIOS calls. The "int \$80" instruction is what you'd normally use to make very low-level *NIX calls.

17.22 Bound Instruction

This instruction takes the following forms:

```
bound( Reg16, mem )
bound( Reg16, AnonMem )

bound( Reg32, mem )
bound( Reg32, AnonMem )
```

Extended Syntax Form:

```
bound( Reg16, constL, constH )
bound( Reg32, ConstL, ConstH )
```

These instructions return the register as their "returns" value.

The extended syntax forms emit the two constants to the static data segment and substitute the address of the first constant (`CONSTL`) as their memory operand.

The `BOUND` instruction compares the register operand against the two constants (or the two consecutive memory locations at the specified address). If the register value is outside the range specified by the operand(s), then the 80x86 CPU raises an `ex.BoundsInsr` exception. You can handle this exception using the `TRY..ENDTRY` HLL statement in HLA.

Because the `BOUND` instruction tends to be slow, and of course it consumes memory, many programmers don't use it as often as they should for fear it will make their programs less efficient. HLA solves this problem using the `"@bound"` compile-time pseudo-variable. If `@bound` contains true (the default value) then HLA will compile the `BOUND` instruction and it will behave normally. If `@bound` contains false, then HLA will not emit any code for the bound instruction (this is similar to "asserts" in C/C++). You can set the value of `@bound` in the `VAL` section or with the "?" operator, e.g.,

```
?@bound := false;

// Code that ignores BOUND instructions
.
.
.
?@bound := true;

// BOUND instructions are active again.
```

17.23 The Enter Instruction

The `ENTER` instruction uses the syntax: `"enter(const, const);"`. The first constant operand is the number of bytes of local variables in a procedure; the second constant operand is the lex level of the procedure. As a rule, you should not use this instruction (and the corresponding `LEAVE` instruction). HLA procedures automatically construct the display and activation record for you (more efficiently than when using `ENTER`).

See the "Art of Assembly" for a further discussion of this instruction and the `LEAVE` instruction.

17.24 CMPXCHG Instruction

This instruction uses the following syntax:
Generic Form:

```
cmpxchg( reg/mem, reg );
lock.cmpxchg( reg/mem, reg );
```

Specific Forms:

```
cmpxchg( Reg8, Reg8 )
cmpxchg( Reg8, Memory )
cmpxchg( Reg8, AnonMem )

cmpxchg( Reg16, Reg16 )
cmpxchg( Reg16, Memory )
cmpxchg( Reg16, AnonMem )

cmpxchg( Reg32, Reg32 )
```

```
cmpxchg( Reg32, Memory )
cmpxchg( Reg32, AnonMem )
```

This instruction returns the empty string as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.25CMPXCHG8B Instruction

This instruction uses the following syntax:

Generic Form:

```
cmpxchg( mem64 );
lock.cmpxchg8b( mem64 );
```

This instruction compares edx:eax with the specified qword operand. If the values are equal, this instruction stores the value in ECX:EBX into the destination operand; otherwise it loads the memory operand into EDX:EAX.

This instruction returns the empty string as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.26The XADD Instruction

The XADD instruction uses the following syntax:

Generic Form:

```
xadd( source, dest );
lock.xadd( source, dest );
```

Specific Forms:

```
xadd( Reg8, Reg8 )
xadd( mem, Reg8 )
xadd( AnonMem, Reg8 )

xadd( Reg16, Reg16 )
xadd( mem, Reg16 )
xadd( AnonMem, Reg16 )

xadd( Reg32, Reg32 )
xadd( mem, Reg32 )
xadd( AnonMem, Reg32 )
```

This instruction returns its destination operand as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

17.27BSF and BSR Instructions

The bit scan instructions use the following syntax (substitute BSR for BSF as appropriate):

Generic Form:

```
bsr( source, dest );
```

Specific Forms Allowed:

```
bsf( Reg16, Reg16 );
bsf( mem, Reg16 );
bsf( AnonMem, Reg16 );

bsf( Reg32, Reg32 );
bsf( mem, Reg32 );
bsf( AnonMem, Reg32 );
```

These instructions return the destination register as their "returns" value. See the "Art of Assembly" for a further discussion of these instructions.

17.28The BSWAP Instruction

This instruction takes the form "bswap(reg32)". It converts between little endian and big endian data formats in the specified 32-bit register.

It returns the 32-bit register as its "returns" value.

See the "Art of Assembly" for a further discussion of this instruction.

17.29Bit Test Instructions

This group of instructions includes BT, BTC, BTR, and BTS. They allow the following generic forms:

Generic Form:

```
bt( BitNumber, Dest );
```

Specific Forms:

```
bt( const, Reg16 );
bt( const, Reg32 );

bt( const, mem );

bt( Reg16, Reg16 );
bt( Reg16, mem );
bt( Reg16, AnonMem );

bt( Reg32, Reg32 );
bt( Reg32, mem );
bt( Reg32, AnonMem );

bt( Reg16, CharacterSetVariable );
```

```
bt( Reg32, CharacterSetVariable );
```

Substitute the BTC, BTR, or BTS mnemonic for BT in the examples above for these other instructions. The BTC, BTR, and BTS instructions also allow a "lock." prefix, e.g., "lock.bt(reg32, mem);" If the "lock." prefix is present, the instruction asserts the bus lock signal during execution. The "lock." prefix is valid only on instructions that reference memory.

These instructions return the destination operand as their "returns" value.

Notice the two special forms that allow character set variables. HLA actually casts these 16-byte objects as word or dword memory variables, but they otherwise work just fine with cset objects.

Special forms available only with the BT instruction:

```
bt( reg16, CharacterSetConstant );
bt( reg32, CharacterSetConstant );
```

These two forms return the source register (BitNumber) as their "returns" value. Note that HLA will create a phantom variable that contains the character set constant and then supplies the name of this constant, effectively making these two instructions equivalent to "bt(reg, CharacterSetVariable);".

See the "Art of Assembly" for a further discussion of these instructions.

17.30 Floating Point Instructions

HLA supports the following FPU instructions. Note: all FPU instructions have a "returns" value of "st0" unless otherwise noted.

```
fld( FPrege );
fst( FPrege );

fld( FPmem );           // Returns operand.
fst( FPmem );           // 32 and 64-bits only! Returns operand.
fstp( FPmem );          // Returns operand.

fxch( FPrege );

fld( FPmem );           // Returns operand.
fist( FPmem );          // 32 and 64-bits only! Returns operand.
fistp( FPmem );         // Returns operand.

fbld( FPmem );          // Returns operand.
fbstp( FPmem );         // Returns operand.

fadd( );
fadd( FPrege, st0 );
fadd( st0, FPrege );
fadd( FPmem );          // Returns operand.
fadd( FPconst );        // Returns operand.

faddp( );
faddp( st0, FPrege );

fmul( );
```

```
fmul( FPreg, st0 );
fmul( st0, FPreg );
fmul( FPmem );           // Returns operand.
fmul( FPconst );        // Returns operand.

fmulp( );
fmulp( st0, FPreg );

fsub( );
fsub( FPreg, st0 );
fsub( st0, FPreg );
fsub( FPmem );           // Returns operand.
fsub( FPconst );        // Returns operand.

fsubp( );
fsubp( st0, FPreg );

fsubr( );
fsubr( FPreg, st0 );
fsubr( st0, FPreg );
fsubr( FPmem );           // Returns operand.
fsubr( FPconst );        // Returns operand.

fsubrp( );
fsubrp( st0, FPreg );

fdiv( );
fdiv( FPreg, st0 );
fdiv( st0, FPreg );
fdiv( FPmem );           // Returns operand.
fdiv( FPconst );        // Returns operand.

fdivp( );
fdivp( st0, FPreg );

fdivr( );
fdivr( FPreg, st0 );
fdivr( st0, FPreg );
fdivr( FPmem );           // Returns operand.
fdivr( FPconst );        // Returns operand.

fdivrp( );
fdivrp( st0, FPreg );

fiadd( mem16 );          // Returns operand.
fiadd( mem32 );          // Returns operand.
fiadd( const );          // Returns operand.

fimul( mem16 );           // Returns operand.
fimul( mem32 );           // Returns operand.
fimul( const );           // Returns operand.

fidiv( mem16 );           // Returns operand.
fidiv( mem32 );           // Returns operand.
fidiv( mem32 );           // Returns operand.
fidiv( const );           // Returns operand.
```

```

fidivr( mem16 );      // Returns operand.
fidivr( mem32 );      // Returns operand.
fidivr( const );     // Returns operand.

fcom( );
fcom( FPreg );
fcom( FPmem );       // Returns operand.

fcomp( );
fcomp( FPreg );
fcomp( FPmem );     // Returns operand.

fucom( );
fucom( FPreg );

fucomp( );
fucomp( FPreg );

fcompp();
fucompp();

ficom( mem16 );      // Returns operand.
ficom( mem32 );      // Returns operand.
ficom( const );     // Returns operand.

ficomp( mem16 );     // Returns operand.
ficomp( mem32 );     // Returns operand.
ficomp( const );     // Returns operand.

fsqrt();             // The following all return "st0"
fscale();
fprem();
fprem1();
frndint();
fextract();
fabs();
fchs();
ftst();
fxam();

fldz();
fld1();
fldpi();
fldl2t();
fldl2e();
fldlg2();
fldln2();
f2xm1();
fsin();
fcos();
fsincos();
fptan();
fpatan();
fyl2x();
fyl2xp1();

```

```

finit();           // Returns ""
fwait();
fclex();
fincstp();
fdecstp();
fnop();
ffree( FPreG );
fldcw( mem );
fstcw( mem );
fstsw( mem );

```

See the chapter on real arithmetic in "The Art of Assembly Language Programming" for details on these instructions. Note that HLA does not support the entire FPU instruction set. HLA v2.0 actually supports the entire FPU instruction set. See the Intel documentation for more details.

17.31 Additional Floating-Point Instructions for Pentium Pro and Later Processors

The FCMOVCc instructions (cc= a, ae, b, be, na, nae, nb, nbe, e, ne, u, nu) use the following basic syntax:

```
FCMOVcc( stn, st0); // n=0..7
```

They move the specified floating point register to ST0 if the specified condition is true.

The FCOMI and FCOMIP instructions use the following syntax:

```
fcomi( st0, stn );
fcomip( st0, stn );
```

These instructions behave like their (syntactical equivalent) FCOM and FCOMP brethren except they store the status in the EFLAGS register directly rather than in the floating point status register.

17.32 MMX Instructions

HLA supports the following MMX instructions found on the Pentium and later processors (note that some instructions are only available on Pentium III and later processors; see the Intel reference manuals for details):

HLA uses the symbols mm0, mm1, ..., mm7 for the MMX register set.

The following MMX instructions all use the same syntax. The syntax is

```
mmxInstr( mmxReg, mmxReg );
mmxInstr( mem64, mmxReg );
```

mmxInstrs:

```

paddb
paddw
paddd
paddsb
paddsw
paddusb
paddusw

psubb
psubw

```


psubd
psubsb
psubsw
psubusb
psubusw

pmulhuw
pmulhw
pmullw
pmaddwd

pavgb
pavgw

pcmpeqb
pcmpeqw
pcmpeqd
pcmpgtb
pcmpgtw
pcmpgtd

packsswb
packuswb
packssdw

punpcklbw
punpcklwd
punpckldq
punpckhbw
punpckhwd
punpckhdq

pand
pandn
por
pxor

pmaxsw
pmaxub

pminsw
pminub

psadbw

The following MMX instructions require a special syntax. The syntax is listed for each instruction.

```
pextrw( constant, mmxReg, Reg32 );  
pinsrw( constant, Reg32, mmxReg );  
pmovmskb( mmxReg, Reg32 );  
pshufw( constant, mmxReg, mmxReg );  
pshufw( constant, mem64, mmxReg );  
  
movd( mem32, mmxReg );  
movd( mmxReg, mem32 );
```

```

movq( mem64, mmxReg );
movq( mmxReg, mem64 );

emms ();

```

The following MMX shift instructions also require a special syntax. They allow the following two forms:

```

mmxshift( immConst, mmxReg );
mmxshift( mmxReg, mmxReg );

```

```

psllw
pslld
psllq
psrlw
psrld
psrlq
psraw
psrad

```

Note that the `psllw`, `psrlw`, and `psraw` instructions only allow an immediate constant in the range 0..15, the `pslld`, `psrld`, and `psrad` instructions only allow constants in the range 0..31, the `psllq` and `psrlq` instructions only allow immediate constants in the range 0..63.

Please see the appropriate Intel documentation or "The Art of Assembly Language" for a discussion of the behavior of these instructions.

17.33 SSE Instructions

HLA supports the following SSE and SSE/2 instructions found on the Pentium III, IV, and later processors (note that some instructions are only available on Pentium IV and later processors; see the Intel reference manuals for details):

HLA uses the symbols `xmm0`, `xmm1`, ..., `xmm7` for the SSE register set.

SSE Instrs:

```

addsd( sseReg/mem128, sseReg );
addpd( sseReg/mem128, sseReg );
addps( sseReg/mem128, sseReg );
addss( sseReg/mem128, sseReg );
andnpd( sseReg/mem128, sseReg );
andnps( sseReg/mem128, sseReg );
andpd( sseReg/mem128, sseReg );
andps( sseReg/mem128, sseReg );

clflush( mem8 );

cmpdpd( imm8, sseReg/mem128, sseReg );
cmpps( imm8, sseReg/mem128, sseReg );
cmpsd( imm8, sseReg/mem64, sseReg );
cmpss( imm8, sseReg/mem32, sseReg );
cmpeqss( sseReg, sseReg );
cmpltss( sseReg, sseReg );
cmplss( sseReg, sseReg );
cmpneqss( sseReg, sseReg );
cmpnlts( sseReg, sseReg );
cmpnles( sseReg, sseReg );

```

```
cmpords( sseReg, sseReg );
cmpunordss( sseReg, sseReg );
cmpeqsd( sseReg, sseReg );
cmpltss( sseReg, sseReg );
cmpless( sseReg, sseReg );
cmpneqsd( sseReg, sseReg );
cmpnlts( sseReg, sseReg );
cmpnles( sseReg, sseReg );
cmpords( sseReg, sseReg );
cmpunords( sseReg, sseReg );

cmpeqps( sseReg, sseReg );
cmpltss( sseReg, sseReg );
cmpleps( sseReg, sseReg );
cmpneqps( sseReg, sseReg );
cmpnltp( sseReg, sseReg );
cmpnleps( sseReg, sseReg );
cmpordps( sseReg, sseReg );
cmpunordps( sseReg, sseReg );

cmpeqpd( sseReg, sseReg );
cmpltpd( sseReg, sseReg );
cmplepd( sseReg, sseReg );
cmpneqpd( sseReg, sseReg );
cmpnltpd( sseReg, sseReg );
cmpnlepd( sseReg, sseReg );
cmpordpd( sseReg, sseReg );
cmpunordpd( sseReg, sseReg );

comisd( sseReg/mem64, sseReg );
comiss( sseReg/mem32, sseReg );
cvt dq2pd( sseReg/mem64, sseReg );
cvt dq2pq
cvt dq2ps( sseReg/mem128, sseReg );
cvt pd2dq( sseReg/mem128, sseReg );
cvt pd2pi( sseReg/mem128, mmxReg );
cvt pd2ps( sseReg/mem128, sseReg );
cvt pi2pd( sseReg/mem64, sseReg );
cvt pi2ps( sseReg/mem64, sseReg );
cvt pi2ss
cvt ps2dq( sseReg/mem128, sseReg );
cvt ps2pd( sseReg/mem64, sseReg );
cvt ps2pi( sseReg/mem64, sseReg );
cvt sd2si( sseReg/mem64, Reg32 );
cvt si2sd( Reg32/mem32, sseReg );
cvt si2ss( sseReg/mem64, sseReg );
cvt ss2sd( sseReg/mem32, sseReg );
cvt sd2ss( Reg32/mem32, sseReg );
cvt ss2si( sseReg/mem32, Reg32 );
cvt tpd2pi( sseReg/mem128, mmxReg );
cvt tpd2dq( sseReg/mem128, sseReg );
cvt tps2dq( sseReg/mem128, sseReg );
cvt tps2pi( sseReg/mem64, mmxReg );
cvt tsd2si( sseReg/mem64, Reg32 );
cvt tss2si( sseReg/mem32, Reg32 );

divpd( sseReg/mem128, sseReg );
```

```
divps( sseReg/mem128, sseReg );
divsd( sseReg/mem64, sseReg );
divss( sseReg/mem32, sseReg );
fxsave( mem512 );
fxrstor( mem512 );
ldmxcsr( mem32 );
lfence

maskmovdqu( sseReg, sseReg );
maskmovq( mmxReg, mmxReg );
maxpd( sseReg/mem128, sseReg );
maxps( sseReg/mem128, sseReg );
maxsd( sseReg/mem64, sseReg );
maxss( sseReg/mem32, sseReg );

mfence

minpd( sseReg/mem128, sseReg );
minps( sseReg/mem128, sseReg );
minsd( sseReg/mem64, sseReg );
minss( sseReg/mem32, sseReg );

movapd( sseReg/mem128, sseReg );
movapd( sseReg, sseReg/mem128 );
movaps( sseReg/mem128, sseReg );
movaps( sseReg, sseReg/mem128 );
movdqa( sseReg/mem128, sseReg );
movdqa( sseReg, sseReg/mem128 );
movdqu( sseReg/mem128, sseReg );
movdqu( sseReg, sseReg/mem128 );
movdq2q( sseReg, mmxReg );
movhlps( sseReg, sseReg );
movhpd( mem64, sseReg );
movhpd( sseReg, mem64 );
movhps( mem64, sseReg );
movhps( sseReg, mem64 );
movlpd( mem64, sseReg );
movlpd( sseReg, mem64 );
movlps( mem64, sseReg );
movlps( sseReg, mem64 );
movlhps( sseReg, sseReg );
movmskpd( sseReg, Reg32 );
movmskps( sseReg, Reg32 );
movnti( Reg32, mem32 );
movntpd( sseReg, mem128 );
movntps( sseReg, mem128 );
movntq( mmxReg, mem64 );
movntdq( sseReg, mem128 );
movq2dq( mmxReg, sseReg );
movsdp( sseReg, sseReg );
movsdp( mem64, sseReg );
movsdp( sseReg, mem64 );
movss( sseReg, sseReg );
movss( mem32, sseReg );
movss( sseReg, mem32 );
movupd( sseReg, sseReg );
movupd( sseReg, mem128 );
```

```
movupd( mem128, sseReg );
movups( sseReg, sseReg );
movups( sseReg, mem128 );
movups( mem128, sseReg );

mulpd( sseReg/mem128, sseReg );
mulps( sseReg/mem128, sseReg );
mulss( sseReg/mem32, sseReg );
mulsd( sseReg/mem64, sseReg );

orpd( sseReg/mem128, sseReg );
orps( sseReg/mem128, sseReg );

pause

pmuludq( mmxReg/mem64, mmxReg );
pmuludq( sseReg/mem128, sseReg );

prefetcht0( mem8 );
prefetcht1( mem8 );
prefetcht2( mem8 );
prefetchnta( mem8 );

pshufd( imm8, sseReg/mem128, sseReg );
pslldq( imm8, sseReg );
psrldq( imm8, sseReg );
punpckhqdq( sseReg/mem128, sseReg );
punpcklqdq( sseReg/mem128, sseReg );

rcpps( sseReg/mem128, sseReg );
rcpss( sseReg/mem128, sseReg );
rsqrtps( sseReg/mem128, sseReg );
rsqrtss( sseReg/mem32, sseReg );

sfence;

shufpd( imm8, sseReg/mem128, sseReg );
shufps( imm8, sseReg/mem128, sseReg );
sqrtpd( sseReg/mem128, sseReg );
sqrtps( sseReg/mem128, sseReg );
sqrtsd( sseReg/mem64, sseReg );
sqrtss( sseReg/mem32, sseReg );

stmxcsr( mem32 );

subps( sseReg/mem128, sseReg );
subpd( sseReg/mem128, sseReg );
subsd( sseReg/mem64, sseReg );
subss( sseReg/mem32, sseReg );

ucomisd( sseReg/mem64, sseReg );
ucomiss( sseReg/mem32, sseReg );

unpckhpd( sseReg/mem128, sseReg );
unpckhps( sseReg/mem128, sseReg );
unpcklpd( sseReg/mem128, sseReg );
unpcklps( sseReg/mem128, sseReg );
```

```
xorpd( sseReg/mem128, sseReg );
xorps( sseReg/mem128, sseReg );
```

17.34 OS/Privileged Mode Instructions

Although HLA was originally intended for writing 32-bit flat model user mode applications, some HLA users may wish to write an operating system kernel or device drivers within HLA. Therefore, HLA provides support for various privileged instructions and instructions that manipulate segment registers on the 80x86 processor. This section describes those instructions. Normal application programs should not use these instructions (most will cause a "General Protection Fault" if you attempt to execute them).

For additional information on these instructions, please see the Intel documentation for the Pentia processors.

```
arpl( r16, r/m16 );
```

Adjusts the RPL field of a segment descriptor.

```
clts();
```

Clears the task switched flag in CR0.

```
hlt();
```

Halts the processor until an interrupt or reset comes along.

```
invd();
```

Invalidates the internal cache.

```
invlpg( mem );
```

Invalidates the TLB entry associated with the memory address specified as the source operand.

```
lar( r/m16, r16 );
lar( r/m32, r32 );
```

Load access rights from the segment descriptor specified by the first operand into the second operand.

```
lds( r32, m48 );
les( r32, m48 );
lfs( r32, m48 );
lgs( r32, m48 );
lss( r32, m48 );
```

Load a far (48-bit) segmented pointer into ds, es, fs, gs, or ss, and some other 32-bit register. Note that HLA does not support an `fword` data type. These instructions require a 48-bit memory operand, nonetheless. You may create your own 48-bit `fword` data type using a record declaration like the following:

```
type
    fword: record
        offset: dword;
```

```

        selector: word;
    endrecord;

lgdt( mem48 );
lidt( mem48 );
sgdt( mem48 );
sidt( mem48 );

```

Loads or stores the global descriptor table pointer (lgdt/sgdt) or interrupt descriptor table pointer (lidt/sidt) via the specified 48-bit memory operand. HLA does not support a 48-bit data type specifically for these instructions, but you can easily create one as follows:

```

type
    descPtr: record
        lowerLimit: word;
        baseAdrs: dword;
    endrecord

lldt( r/m16 );
sldt( r/m16 );

```

These instructions copy the specified source operand to/from the local descriptor table.

```

lsl( r/m16, r16 );
lsl( r/m32, r32 );

```

Load segment limit instruction;

```

ltreg( r/m16 );
streg( r/m16 );

```

Load and store the task register. Note that Intel uses the mnemonics "ltr" and "str" for these instructions. HLA changes these mnemonics to avoid conflicts with the commonly used "str" namespace (the HLA strings module).

```

mov( r/m16, segreg );
mov( segreg, r/m16 );

```

Copies data between an 80x86 segment register and a 16-bit register or memory location. Note that HLA uses the following register names for the segment registers:

cseg	The 80x86 CS register.
dseg	The 80x86 DS register
eseg	The 80x86 ES register
fseg	The 80x86 FS register
gseg	The 80x86 GS register
sseg	The 80x86 SS register

HLA uses these names rather than the Intel standard register names to avoid conflicts with the "cs" (cset) namespace identifier and other commonly used application identifiers. Note that CSEG may not be a destination register for the MOV instruction.

```
mov( r32, crx );    // note: x= 0, 2, 3, or 4.
mov( crx, r32 );
```

These instructions move data between one of the 32-bit registers and one of the x86's control registers. Note that HLA reserves names cr0..cr7 even though Intel doesn't currently define all eight control registers.

```
mov( r32, drx );    // note: x=0, 1, 2, 3, 6, 7
mov( drx, r32 );
```

These instructions move data between the general-purpose 32-bit registers the the x86 debug registers. Note that HLA reserves names dr0..dr7 even though the assembler doesn't currently support the user of the dr4 and dr5 registers.

```
push( segreg );
pop( segreg );
```

These instructions push and pop the x86 segment registers (cseg, dseg, eseg, fseg, gseg, and sseg). Note, however, that you cannot pop the cseg register. (see the comment earlier about HLA segment register names).

```
rdmsr();
rdpmc();
```

These instructions read model-specific registers or performance monitoring registers on the x86. The ECX register specifies the register to read, these instructions copy the data to EDX:EAX.

```
rsm();
```

Resumes from system management mode.

```
verr( r/m16 );
verw( r/m16 );
```

Verifies whether the specified code segment is readable (verr) or writable (verw) from the current privilege level.

```
wbinvd();
```

Write-back and invalidate cache.

17.35 Other Instructions and features

Currently, HLA does not support 3DNow, or certain other SIMD instructions found on later x86 processors. The intent is to add support in the near future.

Note that HLA does not support the LMSW and SMSW instructions (old, obsolete 286 instructions). Use MOV with CR0 instead.

In the meantime, if you need to use any of these instructions you can use the #ASM.#ENDASM and #EMIT directives to insert them into your programs. You can also use macros to implement any desired instructions or syntaxes you desire.

Segment overrides are possible using the segment names (cseg, dseg, eseg, fseg, gseg, and sseg) as a label before an instruction, e.g.,

```
fseg: mov( [eax], eax ); // Fetches from fs:[eax].
```


Generally, you don't need segment overrides in flat-model 32-bit OS environments. However, the operating system kernel (even flat-model OSes) sometimes need to apply a segment override, for example to support Structured Exception Handling under Windows, hence this discussion.

18 Advanced HLA Programming

18.1 Writing a DLL in HLA

Dynamic link libraries provide an efficient mechanism for sharing code and cross-language linkage. The HLA language does not require any specific syntax to create a DLL; most of the work is done by the linker. However, to successfully write and call DLLs with HLA, you must follow some standard conventions.

Acknowledgement: I learned much of the material needed to write DLLs in HLA by visiting the following web page and looking at the CRCDemo file (which demonstrates how to write DLLs in assembly language). For more information on DLLs in assembly, you might want to take a look at this page yourself:

<http://www.geocities.com/SiliconValley/Heights/7394/index.html>

I certainly acknowledge stealing lots of information and ideas from this CRC code and documentation.

18.1.1 Creating a Dynamic Link Library

Win32 Dynamic Link Libraries provide a mechanism whereby two or more programs can share the same set of library object modules on the disk. At the very least, DLLs save space on the disk; if properly written and loaded into memory, DLLs can also share run-time memory and reduce swap space usage on the hard disk.

Perhaps even more important than saving space, DLLs provide a mechanism whereby two different programming languages may communicate with one another. Although there is usually no problems calling an assembly language (i.e., HLA) module from any given high level language, DLLs do provide one higher level of generality. In order to achieve this generality, Microsoft had to carefully describe the calling mechanism between DLLs and other modules. In order to communicate data, all languages that support DLLs need to agree on the calling and parameter passing mechanisms.

Microsoft has laid down the following rules for DLLs (among others):

- Procedures/functions with a fixed parameter list use the stdcall calling mechanism.
- Procedures/functions with a variable number of parameters use the C calling mechanism.
- Parameters can be bytes, words, doublewords, pointers, or strings. Pointers are machine addresses; strings are pointers to a zero-terminated sequence of characters, and it is up to the two modules to agree on how to interpret byte, word, or dword data (e.g., char, int16, uns32, etc.)

Stdcall procedures push their parameters from left to right as they are encountered in the parameter list. In stdcall procedures, it is the procedure's responsibility to clean up the parameters pushed on the stack.

HLA uses the stdcall calling mechanism for the HLL-style procedure calls, so this simplifies the interface to DLL code when using fixed parameter lists (variable parameter lists are rare in DLLs, but should they be necessary, one can always drop down into "pure" assembly in HLA and accommodate the DLL).

The only other issue, with respect to stdcall conventions, is the naming convention. The stdcall mechanism *mangles* procedure names. In particular, a procedure name like "XXXX" is

translated to “`XXX@n`” where “`n`” is the number of bytes of parameters passed to the procedure. HLA does not automatically mangle procedure names, but using the “`external`” directive you can easily specify the mangled name.

DLLs must provide a special procedure that Windows calls to initialize the procedure. This DLL entry point must use an HLA definition like the following:

```
procedure dll( instance:dword; reason:dword; reserved:dword ); external( "_dll@12" );
```

This function must return true in AL if the DLL can be successfully initialized; it returns false if it cannot properly initialize the DLL. Note that “`dll`” and “`_dll@12`” are example names; you may use any reasonable identifiers you choose here.

The DLL initialization function always has three parameters. The second parameter is the only one of real interest to the DLL initialization code. This parameter contains the reason for calling this code, which is one of the following constants defined in the `w.hhf` header file:

- `w.DLL_PROCESS_ATTACH`
- `w.DLL_PROCESS_DETACH`
- `w.DLL_THREAD_ATTACH`
- `w.DLL_THREAD_DETACH`

The `w.DLL_XXXXX_ATTACH` values indicate that some program is linking in the DLL. During these calls, you should open any files, initialize any variables, and execute any other initialization code that may be necessary for the proper operation of the DLL. Note that, by default, all processes that attach to a DLL get their own copy of any data defined in the DLL. Therefore, you do not have to worry about disturbing previous links to the DLL during the current initialization process.

The `w.DLL_XXXXX_DETACH` values indicate that a process or thread is shutting down. During these calls, you should close any files and perform any other necessary cleanup (e.g., freeing memory) that you would normally do before a program ends.

The following code demonstrates a short DLL:

```
unit dllExample;
#include( "w.hhf" );

static
    ThisInstance: dword;

procedure dll( instance:dword; reason:dword; reserved:dword );
    @stdcall; @external( "_dll@12" );

procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw2:dword ); @stdcall; @external( "_dllFunc2@4" );

procedure dll( instance:dword; reason:dword; reserved:dword ); @nodisplay;
begin dll;

    // Save the instance value.

    mov( instance, eax );
```

```

    mov( eax, ThisInstance );

    if( reason = w.DLL_PROCESS_ATTACH ) then

        // Do this code if we're attaching this DLL to a process...

    endif;

    // Return true if successful, false if unsuccessful.

    mov( true, eax );

end dll;

procedure dllFunc1( dw:dword ); @nodisplay;
begin dllFunc1;

    mov( dw, eax );

end dllFunc1;

procedure dllFunc2( dw2:dword ); @nodisplay;
begin dllFunc2;

    push( edx );
    mov( dw2, eax );
    mul( dw2, eax );
    pop( edx );

end dllFunc2;

end dllExample;

```

As you can see here, there is very little difference between a standard unit and an HLA unit intended to become a DLL. The name mangling is one difference, placing the external declarations directly in the file (rather than in an include file) is another difference. The only functional difference is the presence of the DLL initialization procedure (“dll” in this example).

The real work in creating a DLL occurs during the link phase. You cannot compile a DLL the same way you compile a standard HLA program - some additional steps are necessary. Creating a DLL requires lots of command line parameters, so it is best to create a makefile and a “linker” file to avoid excess typing at the command line. Consider the following make file for the module above:

```

dll.dll: dll.obj
    link dll.obj @dll.linkresp

dll.obj: dll.hla
    hla -@ -c dll.hla

```

This makefile generates the dll.dll file (it will also produce several other files, dll.lib being the most important one). The real work appears in the “dll.linkresp” linker file. This file contains the following text:

```
-DLL
-entry:dll
-base:0x40000000
-out:dll.dll
-export:dll
-export:dllFunc1
-export:dllFunc2
```

The “-DLL” option tells the linker to produce a “dll.dll” and a “dll.lib” file rather than just a “dll.exe” file (note: the linker will also produce some other files, but these two are the ones important to us).

The “-entry:dll” option tells the linker that the name of the DLL initialization code is the procedure “dll”. If you change the name of your DLL initialization code, you should also change this option.

The “-base:0x40000000” option tells the linker that this DLL has a base address of 1GByte. For efficiency reasons, you should try to specify a unique value here. If two active DLLs specify the same base address, different processes cannot concurrently share the two DLLs. The programs will still operate, but they will not share the code, wasting some memory and requiring longer load times.

The “-out:dll.dll” command specifies the output name for the DLL. The suffix should be “.dll” and the base filename should be an appropriate name for your DLL (“dll” was appropriate in this case, it would not be appropriate in other cases).

The “-export” options specify the names of the external procedures you wish to make available to other modules. Alternately, you may create a “.DEF” file and use the “-DEF:deffilename.def” option to pass the exported file names on to the linker (see the Microsoft documentation for a description of DEF files).

If you run this make file, it will compile the dll.hla source file producing the dll.dll and dll.lib object modules.

18.1.2 Linking and Calling Procedures in a Dynamic Link Library

Creating a DLL in HLA is only half the battle. The other half is calling a procedure in a DLL from an HLA program. Here is a sample program that calls the DLL procedures in the previous section:

```
// Sample program that calls routines in dll.dll.
//
// Compile this with the command line option:
//
//     hla dllmain dll.lib
//
// Of course, you must build the DLL first.

program callDLL;
#include( "stdlib.hhf" );

procedure dllFunc1( dw:dword ); @stdcall; @external( "_dllFunc1@4" );
procedure dllFunc2( dw:dword ); @stdcall; @external( "_dllFunc2@4" );

begin callDLL;

    xor( eax, eax );
    dllFunc1( 12345 );
    stdout.put( "After dllFunc1, eax = ", (type uns32 eax ), nl );
```

```
dllFunc2( 100 );  
stdout.put( "After dllFunc2, eax = ", (type uns32 eax ), nl );
```

```
end callDLL;
```

To compile this main program, you would use the following HLA command line:

```
hla dllmain dll.lib
```

The "dll.lib" file contains the linkages necessary to load and link in the dll module at run-time.

18.1.3 Going Farther

This document only explains "implicitly loaded" DLLs. Implicitly loaded DLLs are always loaded into memory when the main module loads into memory. If you want to control the loading of the DLL module into memory, you will want to take a look at "explicitly loaded" DLLs. Such DLLs, however, will have to be the subject of a different example.

18.2 Compiling HLA

Source code has been shipped with the HLA releases since HLA v1.18. However, until Bison 1.875 became available, compiling HLA required a special, hacked, version of Bison that ran under Linux. This made development of HLA (particularly under Linux) a bit painful. Fortunately, as of Bison 1.875, it is now possible to compile the HLA source code using standard versions of Flex and Bison available from the Free Software Foundation (the GNU folks). You must, however, have Bison 1.875 or later to successfully translate the HLPARSE.BSN file. Under Windows, the CYGWIN package containing Flex and Bison works great. I (Randy Hyde) have never actually built HLPARSE.BSN under Linux, so I don't have any experience with this process. It may be trivial, it may be impossible. I've never tried it. I always generate HLPARSE.C under Windows using Bison and then I copy the C file over to Linux for compilation there.

First, a couple of comments about the source code: HLA v1.x and v2.x are prototype systems. This means that there are massive kludges in the code. The whole system evolved over time rather than being designed properly in the first place (no apologies for this, that's the whole purpose of a prototype). So if you looking for wonderfully structured code that's easy to follow, HLA will disappoint you. I learned quite a bit about FLEX and BISON while writing HLA and, unfortunately, it shows. There are many ways I've done things that someone who was more familiar with FLEX/Bison would have done differently (heck, there are a lot of things I would do differently, in hindsight). None of this is worth fixing since such work is better put to writing v2.x of HLA.

The HLA source code is almost 200,000 lines long. The Bison file alone is about 100,000 lines of code. Messing with HLA source code is not an undertaking for the weak of heart. Although much of the code is commented, there is very little "high level documentation" (i.e., design documentation) available that would explain why I've done certain things or to provide the general philosophy behind the code. I offer the source code in this form; it is up to you to decide whether you want to spend the time needed to figure it all out.

One note about support: I will be more than happy to answer questions about HLA in the Yahoo AoA/HLA Programming newsgroup. However, I do not have time to answer individual questions asked via email concerning the source code. I apologize ahead of time, but releasing a program of this magnitude to the public could wind up burying me with questions. Because of the possible volume of emails this product could produce, I must ignore all requests for help that arrive via email. Of course, bug reports are always welcome via email. Send everything else to one of the two aforementioned newsgroups.

I have developed HLA with the following tools:

- CodeWright Editor (it takes a decent editor to handle files in excess of 100,000 lines of code).
- Microsoft Visual C++ (v9)
- Flex
- Bison (must be 1.875 or later)
- Microsoft nmake
- GCC 2.9x (Linux, FreeBSD, and Mac OS X versions)
- HLA (a couple of modules are written in HLA itself).
- MASM v9.
- Gas (Linux, FreeBSD, Mac OSX)

I have supplied a makefile that should automatically build the HLA system for you. See the makefile in the main source directory for details. For Linux, there is a "makefile.linux" file that you should use. For FreeBSD use "makefile.freebsd" and for the Macintosh, use "makefile.mac",

HLA is probably not portable. I have made no attempt to ensure that the code compiles with anything other than GCC and MSVC++, so undoubtedly it won't compile on anything else without some effort. I have eliminated *most* of the compiler warnings, so porting to some other compilers shouldn't be too difficult.

Porting HLA to generate assembly code for an assembler other than MASM, NASM, FASM, Gas, or TASM is a *major* undertaking. TASM took a couple of weeks to pull off and TASM is mostly compatible with MASM. Gas took about a month of evenings and FASM took several weekends. Fortunately, if you choose to do this, I've made the process easier and easier with each new back-end assembler I added. Porting HLA to generate object code other than PE/COFF, ELF,

or Mach-o is a *serious* undertaking. I spent a couple of months on each of the three object formats that HLA currently supports.

Porting to other operating systems (other than Windows, Mac OSX, FreeBSD and Linux) is certainly possible. The compiler should be fairly easy to port. The real work is in porting the Standard Library. I've looked into porting HLA to QNX, but haven't pursued this for a couple of reasons: (1) QNX's version of GCC is older and has problems compiling the source code, (2) QNX doesn't really support assembly level calls to the OS so I'd have to port the HLA standard library on top of the C standard library code (which is ugly). NetBSD and OpenBSD should be easy - just a simple modification of the FreeBSD port. At one time I looked into a BeOS port, but then BeOS died, so I gave up. Solaris/Sun OS is a possibility, but now that Oracle has bought out Sun, who knows where that OS is going?

18.3 Code Generation for HLA HLL Control Structures

Note: *This is a very old and incomplete document. It was written back in the early days of HLA v1.x. While the general principles still apply, the specific examples of code generated by HLA have changed quite a bit. Nevertheless, the information is still useful to some people so I've included this document here. If there is sufficient interest, I can be convinced to update and finish this document.*

One of the principal advantages of using assembly language over high level languages is the control that assembly provides. High level languages (HLLs) represent an abstraction of the underlying hardware. Those who write HLL code give up this control in exchange for the engineering efficiencies enjoyed by HLL programmers. Some advanced HLL programmers (who have a good mastery of the underlying machine architecture) are capable of writing fairly efficient programs by recognizing what the compiler does with various high level control constructs and choosing the appropriate construct to emit the machine code they want. While this “low-level programming in a high level language” does leave the programmer at the mercy of the compiler-writer, it does provide a mechanism whereby HLL programmers can write more efficient code by choosing those HLL constructs that compile into efficient machine code.

Although the High Level Assembler (HLA) allows a programmer to work at a very low level, HLA also provides structured high-level control constructs that let assembly programmers use higher-level code to help make their assembly code more readable. Those assembly language programmers who need (or want) to exercise maximum control over their programs will probably want to avoid using these statements since they tend to obscure what is happening at a really low level. At the other extreme, those who would always use these high-level control structures might question if they really want to use assembly language in their applications; after all, if they're writing high level code, perhaps they should use a high level language and take advantage of optimizing technology and other fancy features found in modern compilers. Between these two extremes lies the typical assembly language programmer. The one who realizes that most code doesn't need to be super-efficient and is more interested in productively producing lots of software rather than worrying about how many CPU cycles the one-time initialization code is going to consume. HLA is perfect for this type of programmer because it lets you work at a high level of abstraction when writing code whose performance isn't an issue and it lets you work at a low level of abstraction when working on code that requires special attention.

Between code whose performance doesn't matter and code whose performance is critical lies a big gray region: code that should be reasonably fast but speed isn't the number one priority. Such code needs to be reasonably readable, maintainable, and as free of defects as possible. In other words, code that is a good candidate for using high level control and data structures if their use is reasonably efficient.

Unlike various HLL compilers, HLA does not (yet!) attempt to optimize the code that you write. This puts HLA at a disadvantage: it relies on the optimizer between your ears rather than the one supplied with the compiler. If you write sloppy high level code in HLA then a HLL version of the same program will probably be more efficient if it is compiled with a decent HLL compiler. For code where performance matters, this can be a disturbing revelation (you took the time and bother to write the code in assembly but an equivalent C/C++ program is faster). The purpose of this article is to describe HLA's code generation in detail so you can intelligently choose when to use HLA's high level features and when you should stick with low-level assembly language.

18.3.1 The HLA Standard Library

The HLA Standard Library was designed to make learning assembly language programming easy for beginning programmers. Although the code in the library isn't terrible, very little effort was made to write top-performing code in the library. At some point in the future this may change as work on the library progresses, but if you're looking to write very high-performance code you should probably avoid calling routines in the HLA Standard Library from (speed) critical sections of your program.

Don't get the impression from the previous paragraph that HLA's Standard Library contains a bunch of slow-poke routines, however. Many of the HLA Standard Library routines use decent algorithms and data structures so they perform quite well in typical situations. For example, the HLA string format is far more efficient than strings in C/C++. The world's best C/C++ `strlen` routine is almost always going to be slower than HLA `str.len` function. This is because HLA uses a better definition for string data than C/C++, it has little to do with the actual implementation of the `str.len` code. This is not to say that HLA's `str.len` routine cannot be improved; but the routine is very fast already.

One problem with using the HLA Standard Library is the frame of mind it fosters during the development of a program. The HLA Standard Library is strongly influenced by the C/C++ Standard Library and libraries common in other high level languages. While the HLA Standard Library is a wonderful tool that can help you write assembly code faster than ever before, it also encourages you to think at a higher level. As any expert assembly language programmer can tell you, the real benefits of using assembly language occur only when you “think in assembly” rather than in a high level language. No matter how efficient the routines in the Standard Library happen to be, if you’re “writing C++ programs with MOV instructions” the result is going to be little better than writing the code in C++ to begin with.

One unfortunate aspect of the HLA Standard Library is that it encourages you to think at a higher level and you’ll often miss a far more efficient low-level solution as a result. A good example is the set of string routines in the HLA Standard Library. If you use those routines, even if they were written as efficiently as possible, you may not be writing the fastest possible program you can because you’ve limited your thinking to string objects which are a higher level abstraction. If you did not have the HLA Standard Library laying around and you had to do all the character string manipulation yourself, you might choose to treat the objects as character arrays in memory. This change of perspective can produce dramatic performance improvement under certain circumstances.

The bottom line is this: the HLA Standard Library is a wonderful collection of routines and they’re not particularly inefficient. They’re very easy and convenient to use. However, don’t let the HLA Standard Library lull you into choosing data structures or algorithms that are not the most appropriate for a given section of your program.

18.3.2 Compiling to MASM Code -- The Final Word

The remainder of this document will discuss, in general, how HLA translates various HLL-style statements into assembly code. Sometimes a general discussion may not provide specific answers you need about HLA’s code generation capabilities. Should you have a specific question about how HLA generates code with respect to a given code sequence, you can always run the compiler and observe the output it produces. To do this, it is best to create a simple program that contains only the construct you wish to study and compile that program to assembly code. For example, consider the following very simple HLA program:

```
program t;

begin t;

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

end t;
```

If you compile this program using the command window prompt “hla -s t.hla” then HLA produces the following (MASM) assembly code output (in the “t.asm” output file)¹:

```
if      @Version lt 612
.586
else
.686
.mmx
.xmm
endif
.model flat, syscall
```

1. This code is from an older version of HLA. The actual code HLA generates today is different. But the concepts this document covers still apply.

```

offset32      equ      <offset flat:>
               assume  fs:nothing
?ExceptionPtr equ      <(dword ptr fs:[0])>
               externdef ??HWexcept:near32
               externdef ??Raise:near32

std_output_hndl equ      -11

               externdef __imp__ExitProcess@4:dword
               externdef __imp__GetStdHandle@4:dword
               externdef __imp__WriteFile@20:dword

cseg          segment page public 'code'
cseg          ends
readonly     segment page public 'data'
readonly     ends
strings      segment page public 'data'
strings      ends
dseg        segment page public 'data'
dseg        ends
bssseg      segment page public 'data'
bssseg      ends

strings      segment page public 'data'

?dfltmsg     byte      "Unhandled exception error.",13,10
?dfltmsgsize equ      34
?absmsg      byte      "Attempted call of abstract procedure or
method.",13,10
?absmsgsize  equ      55
strings      ends
dseg        segment page public 'data'
?dfmwritten  word      0
?dfmStdOut   dword     0

               public  ?MainPgmCoroutine
?MainPgmCoroutine byte 0 dup (?)
               dword   ?MainPgmVMT
               dword   0           ;CurrentSP
               dword   0           ;Pointer to stack
               dword   0           ;ExceptionContext
               dword   0           ;Pointer to last caller
?MainPgmVMT   dword   ?QuitMain
dseg         ends
cseg         segment page public 'code'

?QuitMain    proc      near32
               pushd   1
               call    dword ptr __imp__ExitProcess@4

?QuitMain    endp

cseg         ends

cseg         segment page public 'code'

```

```

??DfltExHndlr  proc    near32

                pushd  std_output_hndl
                call   __imp_GetStdHandle@4
                mov    ?dfmStdOut, eax
                pushd  0          ;lpOverlapped
                pushd  offset32 ?dfmwritten    ;BytesWritten
                pushd  ?dfltmsgsize    ;nNumberOfBytesToWrite
                pushd  offset32 ?dfltmsg    ;lpBuffer
                pushd  ?dfmStdOut      ;hFile
                call   __imp_WriteFile@20

                pushd  0
                call   dword ptr __imp_ExitProcess@4

??DfltExHndlr  endp

                public ??Raise
??Raise proc    near32
                jmp    ??DfltExHndlr
??Raise endp

                public ??HWexcept
??HWexcept proc    near32
                mov    eax, 1
                ret
??HWexcept endp

?abstract proc    near32

                pushd  std_output_hndl
                call   __imp_GetStdHandle@4
                mov    ?dfmStdOut, eax
                pushd  0          ;lpOverlapped
                pushd  offset32 ?dfmwritten    ;BytesWritten
                pushd  ?absmsgsize    ;nNumberOfBytesToWrite
                pushd  offset32 ?absmsg    ;lpBuffer
                pushd  ?dfmStdOut      ;hFile
                call   __imp_WriteFile@20

                pushd  0
                call   dword ptr __imp_ExitProcess@4

?abstract endp

                public ?HLAMain
?HLAMain proc    near32

; Set up the Structured Exception Handler record
; for this program.

                push  offset32 ??DfltExHndlr
                push  ebp
                push  offset32 ?MainPgmCoroutine

```

```

        push    offset32 ??HWexcept
        push    ?ExceptionPtr
        mov     ?ExceptionPtr, esp
        mov     dword ptr ?MainPgmCoroutine+12, esp

        pushd  0                ;No Dynamic Link.
        mov     ebp, esp        ;Pointer to Main's locals
        push   ebp              ;Main's display.
        mov     [ebp+16], esp
        cmp     eax, 0
        jne    ?1_false
        mov     eax, 1
?1_false:
        push   0
        call   dword ptr __imp__ExitProcess@4
?HLAMain
cseg    ends
end

```

The code of interest in this example is at the very end, after the comment “;Main’s display” appears in the text. The actual code sequence that corresponds to the IF statement in the main program is the following:

```

        cmp     eax, 0
        jne    ?1_false
        mov     eax, 1
?1_false:

```

Note: you can verify that this is the code emitted by the IF statement by simply removing the IF, recompiling, and comparing the two assembly outputs. You’ll find that the only difference between the two assembly output files is the four lines above. Another way to “prove” that this is the code sequence emitted by the HLA IF statement is to insert some comments into the assembly output file using HLA’s #ASM.#ENDASM directives. Consider the following modification to the “t.hla” source file:

```

program t;

begin t;

    #asm
    ; Start of IF statement:
    #endasm

    if( eax = 0 ) then

        mov( 1, eax );

    endif;

    #asm
    ; End if IF statement.
    #endasm

end t;

```

HLA's `#asm` directive tells the compiler to simply emit everything between the `#asm` and `#endasm` keywords directly to the assembly output file. In this example the HLA program uses these directives to emit a pair of comments that will bracket the code of interest in the output file. Compiling this to assembly code (and stripping out the irrelevant stuff before the HLA main program) yields the following:

```

                                public  ?HLAMain
?HLAMain                        proc    near32

                                ; Set up the Structured Exception Handler record
                                ; for this program.

                                push    offset32 ??DfltExHndlr
                                push    ebp
                                push    offset32 ?MainPgmCoroutine
                                push    offset32 ??HWexcept
                                push    ?ExceptionPtr
                                mov     ?ExceptionPtr, esp
                                mov     dword ptr ?MainPgmCoroutine+12, esp

                                pushd   0                ;No Dynamic Link.
                                mov     ebp, esp         ;Pointer to Main's locals
                                push    ebp             ;Main's display.
                                mov     [ebp+16], esp

                                ;#asm

                                ; Start of IF statement:
                                ;#endasm

                                cmp     eax, 0
                                jne     ?1_false
                                mov     eax, 1
?1_false:

                                ;#asm

                                ; End if IF statement.
                                ;#endasm

                                push    0
                                call    dword ptr __imp__ExitProcess@4
?HLAMain                        endp
cseg                             ends
end

```

This technique (embedding bracketing comments into the assembly output file) is very useful if it is not possible to isolate a specific statement in its own source file when you want to see what HLA does during compilation.

18.3.3 The HLA `if..then..endif` Statement, Part I

Although the HLA IF statement is actually one of the more complex statements the compiler has to deal with (in terms of how it generates code), the IF statement is probably the first statement that comes to mind when something thinks about high level control structures. Furthermore, you can implement most of the other control structures if you have an IF and a GOTO (JMP) statement, so it makes sense to discuss the IF statement first. Nevertheless, there is a bit of complexity that is unnecessary at this point, so we'll begin our discussion with a simplified version of the IF statement; for this simplified version we'll not consider the ELSEIF and ELSE clauses of the IF statement.

The basic HLA IF statement uses the following syntax:

```
if( simple_boolean_expression ) then
    << statements to execute if the expression evaluates true >>
endif;
```

At the machine language level, what the compiler needs to generate is code that does the following:

```
<< Evaluate the boolean expression >>
<< Jump around the following statements if the expression was false >>
<< statements to execute if the expression evaluates true >>
<< Jump to this point if the expression was false >>
```

The example in the previous section is a good demonstration of what HLA does with a simple IF statement. As a reminder, the HLA program contained

```
if( eax = 0 ) then
    mov( 1, eax );
endif;
```

and the HLA compiler generated the following assembly language code:

```
        cmp     eax, 0
        jne     ?1_false
        mov     eax, 1
?1_false:
```

Evaluation of the boolean expression was accomplished with the single “`cmp eax, 0`” instruction. The “`jne ?1_false`” instruction jumps around the “`mov eax, 1`” instruction (which is the statement to execute if the expression evaluates true) if the expression evaluates false. Conversely, if EAX is equal to zero, then the code falls through to the MOV instruction. Hence the semantics are exactly what we want for this high level control structure.

HLA automatically generates a unique label to branch to for each IF statement. It does this properly even if you nest IF statements. Consider the following code:

```
program t;
```

```

begin t;

    if( eax > 0 ) then

        if( eax < 10 ) then

            inc( eax );

        endif;

    endif;

end t;

```

The code above generates the following assembly output:

```

                cmp     eax, 0
                jna     ?1_false
                cmp     eax, 10
                jnb     ?2_false
                inc     eax
?2_false:
?1_false:

```

As you can tell by studying this code, the INC instruction only executes if the value in EAX is greater than zero and less than ten.

Thus far, you can see that HLA's code generation isn't too bad. The code it generates for the two examples above is roughly what a good assembly language programmer would write for approximately the same semantics.

18.3.4 Boolean Expressions in HLA Control Structures

The HLA IF statement and, indeed, most of the HLA control structures rely upon the evaluation of a boolean expression in order to direct the flow of the program. Unlike high level languages, HLA restricts boolean expressions in control structures to some very simple forms. This was done for two reasons: (1) HLA's design frowns upon side effects like register modification in the compiled code, and (2) HLA is intended for use by beginning assembly language students; the restricted boolean expression model is closer to the low level machine architecture and it forces them to start thinking in these terms right away.

With just a few exceptions, HLA's boolean expressions are limited to what HLA can easily compile to a CMP and a condition jump instruction pair or some other simple instruction sequence. Specifically, HLA allows the following boolean expressions:

```
operand1 relop operand2
```

relop is one of:

```

= or ==          (either one, both are equivalent)
<> or !=        (either one, both are equivalent)
<
<=
>

```


>=

A CPU flag specification.
 A CPU register.
 A boolean or byte variable.

In the expressions above `operand1` and `operand2` are restricted to those operands that are legal in a `CMP` instruction. This is because HLA translates expressions of this form to the two instruction sequence:

```
cmp( operand1, operand2 );
jXX someLabel;
```

where “jXX” represents some condition jump whose sense is the opposite of that of the expression (e.g., “`eax > ebx`” generates a “JNA” instruction since “NA” is the opposite of “>”).

Assuming you want to compare the two operands and jump around some sequence of instructions if the relationship does not hold, HLA will generate fairly efficient code for this type of expression. One thing you should watch out for, though, is that HLA’s high level statements (e.g., `IF`) make it very easy to write code like the following:

```
if( i = 0 ) then
    ...
elseif( i = 1 ) then
    ...
elseif( i = 2 ) then
    ...
.
.
.
endif;
```

This code looks fairly innocuous, but the programmer who is aware of the fact that HLA emits the following would probably not use the code above:

```
    cmp( i, 0 );
    jne lbl;
    .
    .
    .
lbl: cmp( i, 1 );
    jne lbl2;
    .
    .
    .
lbl2: cmp( i, 2 );
    .
    .
    .
```

A good assembly language programmer would realize that it’s much better to load the variable “i” into a register and compare the register in the chain of `CMP` instructions rather than compare the

variable each time. The high level syntax slightly obscures this problem; just one thing to be aware of.

HLA's boolean expressions do not support conjunction (logical AND) and disjunction (logical OR). The HLA programmer must manually synthesize expressions involving these operators. Doing so forces the programmer to link in lower level terms, which is usually more efficient. However, there are many common expressions involving conjunction that HLA could efficiently compile into assembly language. Perhaps the most common example is a test to see if an operand is within (or outside) a range specified by two constants. In a HLL like C/C++ you would typically use an expression like "(value >= low_constant && value <= high_constant)" to test this condition. HLA allows four special boolean expressions that check to see if a register or a memory location is within a specified range. The allowable expressions take the following forms:

```
register in constant .. constant
register not in constant .. constant

memory in constant .. constant
memory not in constant .. constant
```

Here is a simple example of the first form with the code that HLA generates for the expression:

```
if( eax in 1..10 ) then
    mov( 1, ebx );
endif;
```

Resulting (MASM) assembly code:

```
        cmp     eax, 1
        jb     ?1_false
        cmp     eax, 10
        ja     ?1_false
        mov     ebx, 1
?1_false:
```

Once again, you can see that HLA generates reasonable assembly code without modifying any register values. Note that if modifying the EAX register is okay, you can write slightly better code by using the following sequence:

```
        dec     eax
        cmp     eax, 9
        ja     ?1_false
        mov     ebx, 1
?1_false:
```

While, in general, a simplification like this is not possible you should always remember how HLA generates code for the range comparisons and decide if it is appropriate for the situation.

By the way, the "not in" form of the range comparison does generate slightly different code than the form above. Consider the following:

```
if( eax not in 1..10 ) then
    mov( 1, eax );
```

```
endif;
```

HLA generates the following (MASM) assembly language code for the sequence above:

```

                                cmp     eax, 1
                                jnb     ?2_true
                                cmp     eax, 10
                                jna     ?1_false
?2_true:
                                mov     eax, 1
?1_false:
```

As you can see, though the code is slightly different it is still exactly what you would probably write if you were writing the low level code yourself.

HLA also allows a limited form of the boolean expression that checks to see if a character value in an eight-bit register is a member of a character set constant or variable. These expressions use the following general syntax:

```

reg8 in CSet_Constant
reg8 in CSet_Variable

reg8 not in CSet_Constant
reg8 not in CSet_Variable
```

These forms were included in HLA because they are so similar to the range comparison syntax. However, the code they generate may not be particularly efficient so you should avoid using these expression forms if code speed and size need to be optimal. Consider the following:

```

if( al in { 'A'..'Z', 'a'..'z', '0'..'9' } ) then
    mov( 1, eax );
endif;
```

This generates the following (MASM) assembly code:

```

strings          segment page public 'data'
?1_cset          byte 00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte 0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings          ends

                 push     eax
                 movzx   eax, al
                 bt      dword ptr ?1_cset, eax
                 pop     eax
                 jnc     ?1_false
                 mov     eax, 1
?1_false:
```

This code is rather lengthy because HLA never assumes that it can disturb the values in the CPU registers. So right off the bat this code has to push and pop EAX since it disturbs the value in EAX. Next, HLA doesn't assume that the upper three bytes of EAX already contain zero, so it zero fills them. Finally, as you can see above, HLA has to create a 16-byte character set in memory in order to test the value in the AL register. While this is convenient, HLA does generate a lot of

code and data for such a simple looking expression. Hence, you should be careful about using boolean expressions involving character sets if speed and space is important. At the very least, you could probably reduce the code above to something like:

```

movzx( charToTest, eax );
bt( eax, { 'A'..'Z', 'a'..'z', '0'..'9' } );
jnc SkipMov;
mov( 1, eax );

```

SkipMov:

This generates code like the following:

```

strings          segment page public 'data'
?cset_3          byte    00h,00h,00h,00h,00h,00h,0ffh,03h
                 byte    0feh,0ffh,0ffh,07h,0feh,0ffh,0ffh,07h
strings          ends

                 movzx   eax, byte ptr ?1_charToTest[0] ;charToTest
                 bt      dword ptr ?cset_3, eax
                 jnc     ?4_SkipMov
                 mov     eax, 1

?4_SkipMov:

```

As you can see, this is slightly more efficient. Fortunately, testing an eight-bit register to see if it is within some character set (other than a simple range, which the previous syntax handles quite well) is a fairly rare operation, so you generally don't have to worry about the code HLA generates for this type of boolean expression.

HLA lets you specify a register name or a memory location as the only operand of a boolean expression. For registers, HLA will use the TEST instruction to see if the register is zero or non-zero. For memory locations, HLA will use the CMP instruction to compare the memory location's value against zero. In either case, HLA will emit a JNE or JE instruction to branch around the code to skip (e.g., in an IF statement) if the result is zero or non-zero (depending on the form of the expression).

```

register
!register

memory
!memory

```

You should not use this trick as an efficient way to test for zero or not zero in your code. The resulting code is very confusing and difficult to follow. If a register or memory location appears as the sole operand of a boolean expression, that register or memory location should hold a boolean value (true or false). Do not think that “if(eax) then...” is any more efficient than “if(eax<>0) then...” because HLA will actually emit the same exact code for both statements (i.e., a TEST instruction). The second is a lot easier to understand if you're really checking to see if EAX is not zero (rather than it contains the boolean value true), hence it is always preferable even if it involves a little extra typing.

Example:

```

if( eax != 0 ) then

    mov( 1, ebx );

```

```

endif;

if( eax ) then

    mov( 2, ebx );

endif;

```

The code above generates the following assembly instruction sequence:

```

                test    eax,eax ;Test for zero/false.
                je      ?2_false
                mov     ebx, 1
?2_false:
                test    eax,eax ;Test for zero/false.
                je      ?3_false
                mov     ebx, 2
?3_false:

```

Note that the pertinent code for both sequences is identical. Hence there is never a reason to sacrifice readability for efficiency in this particular case.

The last form of boolean expression that HLA allows is a flag designation. HLA uses symbols like `@c`, `@nc`, `@z`, and `@nz` to denote the use of one of the flag settings in the CPU FLAGS register. HLA supports the use of the following flag names in a boolean expression:

```

@c, @nc, @o, @no, @z, @nz, @s, @ns, @a, @na, @ae, @nae, @b, @nb, @be,
@nbe, @l, @nl, @g, @ne, @le, @nle, @ge, @nge, @e, @ne

```

Whenever HLA encounters a flag name in a boolean expression, it efficiently compiles the expression into a single conditional jump instruction. So the following IF statement's expression compiles to a single instruction:

```

if( @c ) then

    << do this if the carry flag is set >>

endif;

```

The above code is completely equivalent to the sequence:

```

    jnc SkipStmts;

    << do this if the carry flag is set >>

SkipStmts:

```

The former version, however, is more readable so you should use the IF form wherever practical.

18.3.5 The JT/JF Pseudo-Instructions

The JT (jump if true) and JF (jump if false) pseudo-instructions take a boolean expression and a label. These instructions compile into a conditional jump instruction (or sequence of instructions) that jump to the target label if the specified boolean expression evaluates false. The compilation of these two statements is almost exactly as described for boolean expressions in the previous section. The principle difference is that HLA sneaks in a (MASM) macro declaration because of technical issues involving code generation. Other than this one minor issue in the MASM source code, the code generation is exactly as described above.

The following are a couple of examples that show the usage and code generation for these two statements.

```

lbl2:
    jt( eax > 10 ) label;
label:
    jf( ebx = 10 ) lbl2;

; Translated Code:

?2_lbl2:
?3_BoolExpr    macro    target
                cmp     eax, 10
                ja     target
                endm
                ?3_BoolExpr    ?4_label

?4_label:
?5_BoolExpr    macro    target
                cmp     ebx, 10
                jne    target
                endm
                ?5_BoolExpr    ?2_lbl2

```

18.3.6 The HLA if..then..elseif..else..endif Statement, Part II

With the discussion of boolean expressions out of the way, we can return to the discussion of the HLA IF statement and expand on the material presented earlier. There are two main topics to consider: the inclusion of the ELSEIF and ELSE clauses and the HLA hybrid IF statement. This section will discuss these additions.

The ELSE clause is the easiest option to describe, so we'll start there. Consider the following short HLA code fragment:

```

if( eax < 10 ) then

    mov( 1, ebx );

else

    mov( 0, ebx );

endif;

```

HLA's code generation algorithm emits a JMP instruction upon encountering the ELSE clause; this JMP transfers control to the first statement following the ENDIF clause. The other difference

between the IF/ELSE/ENDIF and the IF/ENDIF statement is the fact that a false expression evaluation transfers control to the ELSE clause rather than to the first statement following the ENDIF. When HLA compiles the code above, it generates machine code like the following:

```

        cmp     eax, 10
        jnb    ?2_false    ;Branch to ELSE section if false

        mov     ebx, 1
        jmp    ?2_endif    ;Skip over ELSE section

; This is the else section:

?2_false:
        mov     ebx, 0
?2_endif:

```

About the only way you can improve upon HLA's code generation sequence for an IF/ELSE statement is with knowledge of how the program will operate. In some rare cases you can generate slightly better performing code by moving the ELSE section somewhere else in the program and letting the THEN section fall straight through to the statement following the ENDIF (of course, the ELSE section must jump back to the first statement after the ENDIF if you do this). This scheme will be slightly faster if the boolean expression evaluates true most of the time. Generally, though, this technique is a bit extreme.

The ELSEIF clause, just as its name suggests, has many of the attributes of an ELSE and an IF clause in the IF statement. Like the ELSE clause, the IF statement will jump to an ELSEIF clause (or the previous ELSEIF clause will jump to the current ELSEIF clause) if the previous boolean expression evaluates false. Like the IF clause, the ELSEIF clause will evaluate a boolean expression and transfer control to the following ELSEIF, ELSE, or ENDIF clause if the expression evaluates false; the code falls through to the THEN section of the ELSEIF clause if the expression evaluates true. The following examples demonstrate how HLA generates code for various forms of the IF..ELSEIF.. statement:

Single ELSEIF clause:

```

if( eax < 10 ) then

    mov( 1, ebx );

elseif( eax > 10 ) then

    mov( 0, ebx );

endif;

; Translated code:

        cmp     eax, 10
        jnb    ?2_false
        mov     ebx, 1
        jmp    ?2_endif
?2_false:
        cmp     eax, 10
        jna    ?3_false
        mov     ebx, 0
?3_false:
?2_endif:

```

Single ELSEIF clause with an ELSE clause:

```

if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
else
    mov( 2, ebx );
endif;

```

; Converted code:

```

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:

```

IF statement with two ELSEIF clauses:

```

if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
elseif( eax = 5 ) then
    mov( 2, ebx );
endif;

```

; Translated code:

```

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1

```



```

                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                mov     ebx, 2
?2_endif:

```

IF statement with two ELSEIF clauses and an ELSE clause:

```

if( eax < 10 ) then
    mov( 1, ebx );
elseif( eax > 10 ) then
    mov( 0, ebx );
elseif( eax = 5 ) then
    mov( 2, ebx );
else
    mov( 3, ebx );
endif;

```

; Translated code:

```

                cmp     eax, 10
                jnb     ?2_false
                mov     ebx, 1
                jmp     ?2_endif
?2_false:
                cmp     eax, 10
                jna     ?3_false
                mov     ebx, 0
                jmp     ?2_endif
?3_false:
                cmp     eax, 5
                jne     ?4_false
                mov     ebx, 2
                jmp     ?2_endif
?4_false:
                mov     ebx, 3
?2_endif:

```

This code generation algorithm generalizes to any number of ELSEIF clauses. If you need to see an example of an IF statement with more than two ELSEIF clauses, feel free to run a short example through the HLA compiler to see the result.

In addition to processing boolean expressions, the HLA IF statement supports a hybrid syntax that lets you combine the structured nature of the IF statement with the unstructured nature of typical assembly language control flow. The hybrid form gives you almost complete control over the code generation process without completely sacrificing the readability of an IF statement. The following is a typical example of this form of the IF statement:

```

if
{
    cmp( eax, 10 );
    jna false;
}

    mov( 0, eax );

endif;

```

; The above generates the following assembly code:

```

                cmp     eax, 10
                jna     ?2_false
?2_true:
                mov     eax, 0
?2_false:

```

Of course, the hybrid IF statement fully supports ELSE and ELSEIF clauses (in fact, the IF and ELSEIF clauses can have a potpourri of hybrid or traditional boolean expression forms). The hybrid forms, since they let you specify the sequence of instructions to compile, put the issue of efficiency squarely in your lap. About the only contribution that HLA makes to the inefficiency of the program is the insertion of a JMP instruction to skip over ELSEIF and ELSE clauses.

Although the hybrid form of the IF statement lets you write very efficient code that is more readable than the traditional “compare and jump” sequence, you should keep in mind that the hybrid form is definitely more difficult to read and comprehend than the IF statement with boolean expressions. Therefore, if the HLA compiler generates reasonable code with a boolean expression then by all means use the boolean expression form; it will probably be easier to read.

18.3.7 The While Statement

The only difference between an IF statement and a WHILE loop is a single JMP instruction. Of course, with an IF and a JMP you can simulate most control structures, the WHILE loop is probably the most typical example of this. The typical translation from WHILE to IF/JMP takes the following form:

```

while( expr ) do

    << statements >>

endwhile;

// The above translates to:

```

```

label:
    if( expr ) then

        << statements >>
        jmp label;

    endif;

```

Experienced assembly language programmers know that there is a slightly more efficient implementation if it is likely that the boolean expression is true the first time the program encounters the loop. That translation takes the following form:

```

    jmp testlabel;
label:

    << statements >>

testlabel:
    JT( expr ) label;    // Note: JT means jump if expression is true.

```

This form contains exactly the same number of instructions as the previous translation. The difference is that a JMP instruction was moved out of the loop so that it executes only once (rather than on each iteration of the loop). So this is slightly more efficient than the previous translation. HLA uses this conversion algorithm for WHILE loops with standard boolean expressions.

If you look at HLA's output code, you'll discover that it is really complex and messy. The reason has to do with HLA's code generation algorithm. In order to move code around in the program (required in order to move the test of the boolean expression below the statements that comprise the body of the loop) HLA writes a MASM macro at the top of the loop and then expands that macro at the bottom of the loop. The following short example demonstrates how HLA transforms WHILE statements:

```

while( eax > 0 ) do

    mov( 0, eax );

endwhile;

; Translated code:

                                jmp     ?2_continue

?2_true:
?2_while:

?2_macro      macro
?2_continue:

    cmp     eax, 0
    ja     ?2_while
    endm

    mov     eax, 0
    ?2_macro

```

```
?2_exitloop:
```

As you'll find by carefully studying this code, HLA emits a macro definition at the point it encounters the WHILE statement. Then it emits an expansion of that macro at the bottom of the loop. This effectively moves the code associated with the computation of the boolean expression to the bottom of the loop.

Because of this code motion, there is very little overhead associated with a WHILE loop that you haven't already seen (i.e., the IF statement). Therefore, with one exception, the WHILE and IF statements share the same efficiency concerns. The single exception is the hybrid WHILE statement. For technical reasons, HLA cannot move the code associated with the termination check of a hybrid WHILE loop to the bottom of the loop. Therefore, whenever you use the hybrid form of the WHILE statement HLA compiles the code you supply at the top of the loop, it adds a JMP instruction to the bottom of the loop, and that JMP instruction executes on each iteration. If this is a problem for your code, you should probably consider a different implementation of the loop.

Example of the compilation of a hybrid WHILE loop:

```
while
{
    cmp( eax, 0 );
    jne false;
}

    mov( 0, eax );

endwhile;

; Translated code:

?2_while:
?2_continue:
                cmp     eax, 0
                jne     ?2_false

?2_true:
                mov     eax, 0
                jmp     ?2_while

?2_exitloop:
?2_false:
```

18.3.8 repeat..until

To Be Written...

18.3.9 for..endfor

To Be Written...

18.3.10 forever..endfor

To Be Written...

18.3.11 break, breakif

To Be Written...

18.3.12 continue, continueif

To Be Written...

18.3.13 begin..end, exit, exitif

To Be Written...

18.3.14 foreach..endfor

To Be Written...

18.3.15 try..unprotect..exception..anyexception..endtry, raise

To Be Written...

18.4 A Modified IF..ELSE..ENDIF Statement

The IF statement is another statement that doesn't always do exactly what you want. Like the `_while.._onbreak.._endwhile` example above, it's quite possible to redefine the IF statement so that it behaves the way we want it to. In this section you'll see how to implement a variant of the IF..ELSE..ENDIF statement that nests differently than the standard IF statement.

HLA's particular variant of the IF statement has several limitations. One of the major limitations is the inability to combine logical sub-expressions using logical conjunction (and) and logical disjunction (or). It is possible to simulate conjunction and disjunction if you carefully structure your code. Consider the following example:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << statements >>
}

// Equivalent HLA version:

if( expr1 ) then

    if( expr2 ) then

        << statements >>

    endif;

endif;
```

In both cases ("C" and HLA) the `<< statements>>` block executes only if both `expr1` and `expr2` evaluate true. So other than the extra typing involved, it is often very easy to simulate logical conjunction by using two IF statements in HLA.

There is one very big problem with this scheme. Consider what happens if you modify the "C" code to be the following:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}
```

The only way to convert this to HLA (using the standard HLA high level control constructs) is by duplicating the 'false' statements. This introduces a bit of inefficiency into your code. As a result, many HLA programmers will switch to low-level control constructs or HLA's hybrid control structures in order to avoid duplicating code. Unfortunately, dropping down into low-level code may make your program harder to read. It would be nice if you could efficiently handle this situation without making your code unreadable. Fortunately, you can do exactly this by creating a new version of the IF statement using HLA's multi-part macro facilities.

Before describing how to create this new type of IF statement, we must digress for a moment and explore an interesting feature of HLA's multi-part macro expansion: KEYWORD macros do not have to use unique names. Whenever you declare an HLA KEYWORD macro, HLA accepts whatever name you choose. If that name happens to be already defined, then the KEYWORD macro name takes precedence as long as the macro is active (that is, from the point you invoke the macro name until HLA encounters the TERMINATOR macro). Therefore, the KEYWORD macro name hides the previous definition of that name until the termination of the macro. This feature applies even to the original macro name; that is, it is possible to define a KEYWORD macro with the same name as the original macro to which the KEYWORD macro belongs. This is a very useful feature because it allows you to change the definition of the macro within the scope of the opening and terminating invocations of the macro.

Although not pertinent to the IF statement we are construction, you should note that parameter and local symbols in a macro also override any previously defined symbols of the same name. So if you use that symbol between the opening macro and the terminating macro, you will get the value of the local symbol, not the global symbol. E.g.,

```
var
    i:int32;
    j:int32;
    .
    .
    .
macro abc:i;
    ?i:text := "j";
    .
    .
    .
terminator xyz;
    .
    .
    .
endmacro
    .
    .
    .
    mov( 25, i );
    mov( 10, j );
    abc
        mov( i, eax );    // Loads j's value (10), not 25 into eax.
    xyz;
```

The code above loads 10 into EAX because the "mov(i, eax);" instruction appears between the opening and terminating macros *abc..xyz*. Between those two macros the local definition of *i* takes precedence over the global definition. Since *i* is a text constant that expands to *j*, the aforementioned MOV statement is really equivalent to "mov(j, eax);". That statement, of course, loads 10 into EAX. Since this problem is difficult to see while reading your code, you should choose local symbols in multi-part macros very carefully. A good convention to adopt is to combine your local symbol name with the macro name, e.g.,

```
macro abc : i_abc;
```

You may wonder why HLA allows something so crazy to happen in your source code, in a moment you'll see why this behavior is useful (and now, with this brief message out of the way, back to our regularly scheduled discussion).

Before we digressed to discuss this interesting feature in HLA multi-part macros, we were trying to figure out how to efficiently simulate the conjunction and disjunction operators in an IF statement without resorting to low-level code. The problem in the example appearing earlier in this

section is that you would have to duplicate some code in order to convert the IF..ELSE statement properly. The following code shows this problem:

```
// "C" code employing logical-AND operator:

if( expr1 && expr2 )
{
    << 'true' statements >>
}
else
{
    << 'false' statements >>
}

// Corresponding HLA code using the "nested-IF" algorithm:

if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

else

    << 'false' statements >>

endif;
```

Note that this code must duplicate the "<< 'false' statements >>" section if the logic is to exactly match the original "C" code. This means that the program will be larger and harder to read than is absolutely necessary.

One solution to this problem is to create a new kind of IF statement that doesn't nest the same way standard IF statements nest. In particular, if we define the statement such that all IF clauses nested with an outer IF..ENDIF block share the same ELSE and ENDIF clauses. If this were the case, then you could implement the code above as follows:

```
if( expr1 ) then

    if( expr2 ) then

        << 'true' statements >>

    else

        << 'false' statements >>

    endif;

endif;
```


If *expr1* is false, control immediately transfers to the ELSE clause. If the value of *expr1* is true, the control falls through to the next IF statement.

If *expr2* evaluates false, then the program jumps to the single ELSE clause that all IFs share in this statement. Notice that a single ELSE clause (and corresponding 'false' statements) appear in this code; hence the code does not necessarily expand in size. If *expr2* evaluates true, then control falls through to the 'true' statements, exactly like a standard IF statement.

Notice that the nested IF statement above does not have a corresponding ENDIF. Like the ELSE clause, all nested IFs in this structure share the same ENDIF. Syntactically, there is no need to end the nested IF statement; the end of the THEN section ends with the ELSE clause, just as the outer IF statement's THEN block ends.

Of course, we can't actually define a new macro named "if" because you cannot redefine HLA reserved words. Nor would it be a good idea to do so even if these were legal (since it would make your programs very difficult to comprehend if the IF keyword had different semantics in different parts of the program. The following program uses the identifiers "_if", "_then", "_else", and "_endif" instead. It is questionable if these are good identifiers in production code (perhaps something a little more different would be appropriate). The following code example uses these particular identifiers so you can easily correlate them with the corresponding high level statements.

```

/*****/
/*                                     */
/* if.hla                             */
/*                                     */
/* This program demonstrates a modification of */
/* the IF..ELSE..ENDIF statement using HLA's */
/* multi-part macros.                  */
/*                                     */
/*****/

program newIF;
#include( "stdlib.hhf" )

// Macro implementation of new form of if..then..else..endif.
//
// In this version, all nested IF statements transfer control
// to the same ELSE clause if any one of them have a false
// boolean expression. Syntax:
//
// _if( expression ) _then
//
//     <<statements including nested _if clauses>>
//
// _else // this is optional
//
//     <<statements, but _if clauses are not allowed here>>
//
// _endif
//
// Note that nested _if clauses do not have a corresponding
// _endif clause. This is because the single _else and/or
// _endif clauses terminate all the nested _if clauses

```

```
// including the first one. Of course, once the code
// encounters an _endif another _if statement may begin.

// Macro to handle the main "_if" clause.
// This code just tests the expression and jumps to the _else
// clause if the expression evaluates false.

macro _if( ifExpr ):elseLbl, hasElse, ifDone;

    ?hasElse := false;
    jf(ifExpr) elseLbl;

// Just ignore the _then keyword.

keyword _then;

// Nested _if clause (yes, HLA lets you replace the main
// macro name with a keyword macro). Identical to the
// above _if implementation except this one does not
// require a matching _endif clause. The single _endif
// (matching the first _if clause) terminates all nested
// _if clauses as well as the main _if clause.

keyword _if( nestedIfExpr );
    jf( nestedIfExpr ) elseLbl;

    // If this appears within the _else section, report
    // an error (we don't allow _if clauses nested in
    // the else section, that would create a loop).

    #if( hasElse )

        #error( "All _if clauses must appear before the _else clause" )

    #endif

// Handle the _else clause here. All we need to is check to
// see if this is the only _else clause and then emit the
// jmp over the else section and output the elseLbl target.

keyword _else;
    #if( hasElse )

        #error( "Only one _else clause is legal per _if.._endif" )

    #else

        // Set hasElse true so we know that we've seen an _else
        // clause in this statement.

        ?hasElse := true;
        jmp ifDone;
        elseLbl:
```

```

        #endif

// _endif has two tasks.  First, it outputs the "ifDone" label
// that _else uses as the target of its jump to skip over the
// else section.  Second, if there was no else section, this
// code must emit the "elseLbl" label so that the false conditional(s)
// in the _if clause(s) have a legal target label.

terminator _endif;

        ifDone:
        #if( !hasElse )

                elseLbl:

        #endif

endmacro;

static
        tr:boolean := true;
        f:boolean := false;

begin newIF;

        // Real quick demo of the _if statement:

        _if( tr ) _then

                _if( tr ) _then
                _if( f ) _then

                        stdout.put( "error" nl );

                _else

                        stdout.put( "Success" );

                _endif

end newIF;

```

Just in case you're wondering, this program prints "Success" and then quits. This is because the nested "_if" statements are equivalent to the expression "true && true && false" which, of course, is false. Therefore, the "_else" portion of this code should execute.

The only surprise in this macro is the fact that it redefines the *_if* macro as a keyword macro upon invocation of the main *_if* macro. The reason this code does this is so that any nested *_if* clauses do not require a corresponding *_endif* and don't support an *_else* clause.

Implementing an ELSEIF clause introduces some difficulties, hence its absence in this example. The design and implementation of an ELSEIF clause is left to the more serious reader¹.

1. I.e., I don't even want to have to think about this problem!

18.5 Object Oriented Programming in Assembly

18.5.1 Hoopla and Hyperbole

Before discussing object-oriented programming (OOP) in assembly language, it is probably a good idea to take a step back and explore the general benefits of using OOP. After all, without such knowledge, the question of "why bother to use OOP in assembly" is unanswerable.

First of all, despite what some OOP proponents claim, object-oriented programming is not an all-encompassing facility that replaces whatever programming paradigm you currently use. Object-oriented programming techniques are a tool. When used in an appropriate fashion, that tool can save you considerable effort. When misapplied, it can make your programs considerably worse. In some sense, OOP techniques are like recursion: incredibly valuable where it's called for, but inefficient and kludgy when you attempt to use it to solve a problem for which it is not well suited. Fortunately, OOP is well suited for many applications, hence its popularity among high-level language (HLL) programmers.

One of the main benefits to object-oriented programming is that it makes it easier to reuse code. Traditionally, to reuse code you would create huge libraries of different functions and call those functions to perform common tasks. The only problem with the library approach is that in order to effectively reuse your code, you had to write very generic library routines. The result was bloated and slow code (that often handled lots of special cases that would never occur in a specific application); attempts to produce "lean and mean" library routines often meant writing dozens or even hundreds of minor variations of the same functions. It often wasn't possible to easily extend such routines to handle new requirements. This was especially difficult if the source code for the original library routines was not available.

Object-oriented programming techniques provide a solution to this problem. Through OOP-oriented features such as *inheritance* and *polymorphism*, it is possible to extend a simplified library function to handle the specific requirements of a given application without having to rewrite the entire code base.

Because OOP techniques allow you to extend a given set of library routines in ways specific to an application, you would get the impression that this programming paradigm is perfect for assembly language programmers (who want "lean and mean" code that doesn't carry around a lot of bloat). Unfortunately, there are two problems with this idea. First of all, you'll find that traditional OOP languages tend to have *huge* class libraries associated with them. And because of the "layered" approach that OOP fosters, including one, seemingly small, function can wind up including half of the library in your application (ever wondered why a "Hello World" program in Delphi is 256K?). Another problem with the OOP is that it does require a small amount of overhead to implement. This reason alone has scared many assembly language programmers away from using object-oriented programming techniques.

Despite the drawbacks and overblown expectations of OOP (that never seem to be met), OOP techniques are useful for solving many problems. The object-oriented programming paradigm is a handy tool that should appear in your programmer's toolbox - ready to use when the need arises. Just as you wouldn't use a hammer for a job that requires a screwdriver, you shouldn't use OOP in an inappropriate situation. However, when the job calls for a screwdriver, it's nice to have one handy; likewise, when OOP techniques are appropriate, they can provide a fast and efficient solution to a given programming problem.

18.5.2 Some Basic Definitions

To begin with, it's probably a good idea to define a few terms this paper will use. Without further ado:

- **CLASS:** a class is a data type template (i.e., record or structure) that specifies the data and procedure components of a "class object".
- **INSTANCE:** an instance is a block of memory with enough storage to hold the data associated with a class variable (see OBJECT).
- **OBJECT:** a variable of some class type. While there is a subtle difference between objects and instances (having to do with the lifetime of the storage bound to an object), we'll treat the two terms as synonyms for our purposes.
- **METHOD:** a procedure or function associated with a class.
- **INHERITENCE:** the ability to reuse fields from another *base* (or ancestor) class.

- **POLYMORPHISM**: an attribute of classes whereby different (types of) objects can be manipulated by the same method calls. For example, a "print" method could display the value of several different object types without requiring a single function that handles every possible data type one could dream up.
- **INFORMATION HIDING**: the use of private data fields and procedures/methods to control the access of an object's internal representation, with the hope of keeping the implementation of a data type independent from its use. This allows easy modification to the data structure without breaking any code that uses the data structure.
- **ABSTRACT DATA TYPE (ADT)**: An abstract data type is a collection of data objects and the functions (which we'll call *methods*) that operate on the data. In a pure abstract data type, the ADT's methods are the only code that has access to the data fields of the ADT; external code may only access the data using function calls to get or set data field values (these are the ADT's *accessor* methods).

18.5.3 OOP Language Facilities

As any die-hard C programmer can tell you, you don't need an "object-oriented programming language" in order to write object-oriented code. Then again, as any C++ programmer will tell you, it's far easier to write the code and the resulting code is far easier to read and maintain if you do use an object-oriented programming language when writing object-oriented applications. The same is true in assembly language - you don't need an assembler that supports object-oriented programming facilities to write object-oriented assembly code, but it's not very effective to do so.

Today, there are two and a half 80x86 assemblers that provide reasonable support for object-oriented programming in assembly language: HLA (the High-Level Assembler), TASM, and MASM. HLA and TASM directly support classes, objects, and other object-oriented programming facilities. MASM does not, but its STRUCT directive is sufficiently flexible that you can easily create macros to simulate most of the object-oriented programming facilities provided HLA and TASM. Arguably, HLA provides the most complete set of object-oriented programming facilities, so this article will use HLA in its examples. The basic concepts, however, apply to both TASM and MASM as well as HLA.

18.5.4 Classes in HLA

HLA's classes provide a good mechanism for creating abstract data types. Fundamentally, a class is little more than a RECORD declaration that allows the definition of fields other than data fields (e.g., procedures, constants, and macros). The inclusion of other program declaration objects in the class definition dramatically expands the capabilities of a class over that of a record. For example, with a class it is now possible to easily define an ADT since classes may include data and methods that operate on that data (procedures).

The principle way to create an abstract data type in HLA is to declare a class data type. Classes in HLA always appear in the TYPE section and use the following syntax:

```
classname : class
    << Class declaration section >>
endclass;
```

The class declaration section is very similar to the local declaration section for a procedure insofar as it allows CONST, VAL, VAR, and STATIC variable declaration sections. Classes also let you define macros and specify procedure, iterator, and *method* prototypes (method declarations are legal only in classes). Conspicuously absent from this list is the TYPE declaration section. You cannot declare new types within a class.

A method is a special type of procedure that appears only within a class. A little later you will see the difference between procedures and methods, for now you can treat them as being one and the same. Other than a few subtle details regarding class initialization and the use of pointers to

classes, their semantics are identical¹. Generally, if you don't know whether to use a procedure or method in a class, the safest bet is to use a method.

You do not place procedure/iterator/method code within a class. Instead you simply supply *prototypes* for these routines. A routine prototype consists of the PROCEDURE, ITERATOR, or METHOD reserved word, the routine name, any parameters, and a couple of optional procedure attributes (@USE, RETURNS, and EXTERNAL). The actual routine definition (i.e., the body of the routine and any local declarations it needs) appears outside the class.

The following example demonstrates a typical class declaration appearing in the TYPE section:

```

TYPE
  TypicalClass: class

      const
          TCconst := 5;

      val
          TCval := 6;

      var
          TCvar : uns32;          // Private field used only by TCproc.

      static
          TCstatic : int32;

      procedure TCproc( u:uns32 ); returns( "eax" );
      iterator TCiter( i:int32 ); external;
      method TCmethod( c:char );

  endclass;

```

As you can see, classes are very similar to records in HLA. Indeed, you can think of a record as being a class that only allows VAR declarations. HLA implements classes in a fashion quite similar to records insofar as it allocates sequential data fields in sequential memory locations. In fact, with only one minor exception, there is almost no difference between a RECORD declaration and a CLASS declaration that only has a VAR declaration section. Later you'll see exactly how HLA implements classes, but for now you can assume that HLA implements them the same as it does records and you won't be too far off the mark.

You can access the *TCvar* and *TCstatic* fields (in the class above) just like a record's fields. You access the CONST and VAL fields in a similar manner. If a variable of type *TypicalClass* has the name *obj*, you can access the fields of *obj* as follows:

```

      mov ( obj.TCconst, eax );
      mov( obj.TCval, ebx );
      add( obj.TCvar, eax );
      add( obj.TCstatic, ebx );
      obj.TCproc( 20 );          // Calls the TCproc procedure in
TypicalClass.
      etc.

```

If an application program includes the class declaration above, it can create variables using the *TypicalClass* type and perform operations using the above methods. Unfortunately, the application program can also access the fields of the *ADT* data type with impunity. For example, if a program created a variable *MyClass* of type *TypicalClass*, then it could easily execute instructions like

1. Note, however, that the difference between procedures and methods makes all the difference in the world to the object-oriented programming paradigm. Hence the inclusion of methods in HLA's class definitions.

“MOV(MyClass.TCvar, eax);” even though this field might be private to the implementation section. Unfortunately, if you are going to allow an application to declare a variable of type *TypicalClass*, the field names will have to be visible. While there are some tricks we could play with HLA’s class definitions to help hide the private fields, the best solution is to thoroughly comment the private fields and then exercise some restraint when accessing the fields of that class. Specifically, this means that ADTs you create using HLA’s classes cannot be “pure” ADTs since HLA allows direct access to the data fields. However, with a little discipline, you can simulate a pure ADT by simply electing not to access such fields outside the class’ methods, procedures, and iterators.

Prototypes appearing in a class are effectively FORWARD declarations. Like normal forward declarations, all procedures, iterators, and methods you define in a class must have an actual implementation later in the code. Alternately, you may attach the EXTERNAL keyword to the end of a procedure, iterator, or method declaration within a class to inform HLA that the actual code appears in a separate module. As a general rule, class declarations appear in header files and represent the interface section of an ADT. The procedure, iterator, and method bodies appear in the implementation section which is usually a separate source file that you compile separately and link with the modules that use the class.

The following is an example of a sample class procedure implementation:

```
procedure TypicalClass.TCproc( u:uns32 ); nodisplay;
  << Local declarations for this procedure >>
begin TCproc;

  << Code to implement whatever this procedure does >>

end TCProc;
```

There are several differences between a standard procedure declaration and a class procedure declaration. First, and most obvious, the procedure name includes the class name (e.g., *TypicalClass.TCproc*). This differentiates this class procedure definition from a regular procedure that just happens to have the name *TCproc*. Note, however, that you do not have to repeat the class name before the procedure name in the BEGIN and END clauses of the procedure (this is similar to procedures you define in HLA NAMESPACES).

A second difference between class procedures and non-class procedures is not obvious. Some procedure attributes (@USE, EXTERNAL, RETURNS, @CDECL, @PASCAL, and @STDCALL) are legal only in the prototype declaration appearing within the class while other attributes (@NOFRAME, @NODISPLAY, @NOALIGNSTACK, and ALIGN) are legal only within the procedure definition and not within the class. Fortunately, HLA provides helpful error messages if you stick the option in the wrong place, so you don’t have to memorize this rule.

If a class routine’s prototype does not have the EXTERNAL option, the compilation unit (that is, the PROGRAM or UNIT) containing the class declaration must also contain the routine’s definition or HLA will generate an error at the end of the compilation. For small, local, classes (i.e., when you’re embedding the class declaration and routine definitions in the same compilation unit) the convention is to place the class’ procedure, iterator, and method definitions in the source file shortly after the class declaration. For larger systems (i.e., when separately compiling a class’ routines), the convention is to place the class declaration in a header file by itself and place all the procedure, iterator, and method definitions in a separate HLA unit and compile them by themselves.

18.5.5 Objects

Remember, a class definition is just a type. Therefore, when you declare a class type you haven’t created a variable whose fields you can manipulate. An *object* is an *instance* of a class; that is, an object is a variable that is some class type. You declare objects (i.e., class variables) the same way you declare other variables: in a VAR, STATIC, or STORAGE section¹. A pair of sample object declarations follow:

1. Technically, you could also declare an object in a READONLY section, but HLA does not allow you to define class constants, so there is little utility in declaring class objects in the READONLY section.


```
var
  T1: TypicalClass;
  T2: TypicalClass;
```

For a given class object, HLA allocates storage for each variable appearing in the VAR section of the class declaration. If you have two objects, *T1* and *T2*, of type *TypicalClass* then *T1.TCvar* is unique as is *T2.TCvar*. This is the intuitive result (similar to RECORD declarations); most data fields you define in a class will appear in the VAR declaration section.

Static data objects (e.g., those you declare in the STATIC section of a class declaration) are not unique among the objects of that class; that is, HLA allocates only a single static variable that all variables of that class share. For example, consider the following (partial) class declaration and object declarations:

```
type
  sc: class

    var
      i:int32;

    static
      s:int32;
      .
      .
      .
  endclass;

var
  s1: sc;
  s2: sc;
```

In this example, *s1.i* and *s2.i* are different variables. However, *s1.s* and *s2.s* are aliases of one another. Therefore, an instruction like “mov(5, s1.s);” also stores five into *s2.s*. Generally you use static class variables to maintain information about the whole class while you use class VAR objects to maintain information about the specific object. Since keeping track of class information is relatively rare, you will probably declare most class data fields in a VAR section.

You can also create dynamic instances of a class and refer to those dynamic objects via pointers. In fact, this is probably the most common form of object storage and access. The following code shows how to create pointers to objects and how you can dynamically allocate storage for an object:

```
var
  pSC: pointer to sc;
  .
  .
  .
  malloc( @size( sc ) );
  mov( eax, pSC );
  .
  .
  .
  mov( pSC, ebx );
  mov( (type sc [ebx]).i, eax );
```

Note the use of type coercion to cast the pointer in EBX as type *sc*.

18.5.6 Inheritance

Inheritance is one of the most fundamental ideas behind object-oriented programming. The basic idea behind inheritance is that a class inherits, or copies, all the fields from some class and then possibly expands the number of fields in the new data type. For example, suppose you created a data type *point* which describes a point in the planar (two dimensional) space. The class for this point might look like the following:

```
type
  point: class

      var
          x:int32;
          y:int32;

      method distance;

endclass;
```

Suppose you want to create a point in 3D space rather than 2D space. You can easily build such a data type as follows:

```
type
  point3D: class inherits( point );

      var
          z:int32;

endclass;
```

The INHERITS option on the CLASS declaration tells HLA to insert the fields of *point* at the beginning of the class. In this case, *point3D* inherits the fields of *point*. HLA always places the inherited fields at the beginning of a class object. The reason for this will become clear a little later. If you have an instance of *point3D* which you call *P3*, then the following 80x86 instructions are all legal:

```
mov( P3.x, eax );
add( P3.y, eax );
mov( eax, P3.z );
P3.distance();
```

Note that the *P3.distance* method invocation in this example calls the *point.distance* method. You do not have to write a separate *distance* method for the *point3D* class unless you really want to do so (see the next section for details). Just like the *x* and *y* fields, *point3D* objects inherit *point*'s methods.

18.5.7 Overriding

Overriding is the process of replacing an existing method in an inherited class with one more suitable for the new class. In the *point* and *point3D* examples appearing in the previous section, the *distance* method (presumably) computes the distance from the origin to the specified point. For a point on a two-dimensional plane, you can compute the distance using the function:

However, the distance for a point in 3D space is given by the equation:

Clearly, if you call the *distance* function for *point* for a *point3D* object you will get an incorrect answer. In the previous section, however, you saw that the *P3* object calls the distance function inherited from the *point* class. Therefore, this would produce an incorrect result.

In this situation the *point3D* data type must override the *distance* method with one that computes the correct value. You cannot simply redefine the *point3D* class by adding a *distance* method prototype:

```
type
  point3D: class inherits( point )

    var
      z:int32;

    method distance;    // This doesn't work!

endclass;
```

The problem with the *distance* method declaration above is that *point3D* already has a distance method – the one that it inherits from the *point* class. HLA will complain because it doesn't like two methods with the same name in a single class.

To solve this problem, we need some mechanism by which we can override the declaration of *point.distance* and replace it with a declaration for *point3D.distance*. To do this, you use the **OVERVERRIDE** keyword before the method declaration:

```
type
  point3D: class inherits( point )

    var
      z:int32;

    override method distance;    // This will work!

endclass;
```

The **OVERVERRIDE** prefix tells HLA to ignore the fact that *point3D* inherits a method named *distance* from the *point* class. Now, any call to the *distance* method via a *point3D* object will call the *point3D.distance* method rather than *point.distance*. Of course, once you override a method using the **OVERVERRIDE** prefix, you must supply the method in the implementation section of your code, e.g.,

```
method point3D.distance; nodisplay;

    << local declarations for the distance function >>

begin distance;

    << Code to implement the distance function >>

end distance;
```

18.5.8 Virtual Methods vs. Static Procedures

A little earlier, this chapter suggested that you could treat class methods and class procedures the same. There are, in fact, some major differences between the two (after all, why have methods if they're the same as procedures?). As it turns out, the differences between methods and procedures is crucial if you want to develop object-oriented programs. Methods provide the second feature necessary to support true polymorphism: virtual procedure calls¹. A virtual procedure call is just a fancy name for an indirect procedure call (using a pointer associated with the object). The

key benefit of virtual procedures is that the system automatically calls the right method when using pointers to generic objects.

Consider the following declarations using the *point* class from the previous sections:

```
var
    P2: point;
    P: pointer to point;
```

Given the declarations above, the following assembly statements are all legal:

```
mov( P2.x, eax );
mov( P2.y, ecx );
P2.distance();      // Calls point3D.distance.

lea( ebx, P2 );    // Store address of P2 into P.
mov( ebx, P );
P.distance();      // Calls point.distance.
```

Note that HLA lets you call a method via a pointer to an object rather than directly via an object variable. This is a crucial feature of objects in HLA and a key to implementing *virtual method calls*.

The magic behind polymorphism and inheritance is that object pointers are *generic*. In general, when your program references data indirectly through a pointer, the value of the pointer should be the address of the underlying data type associated with that pointer. For example, if you have a pointer to a 16-bit unsigned integer, you wouldn't normally use that pointer to access a 32-bit signed integer value. Similarly, if you have a pointer to some record, you would not normally cast that pointer to some other record type and access the fields of that other type¹. With pointers to class objects, however, we can lift this restriction a bit. Pointers to objects may legally contain the address of the object's type *or the address of any object that inherits the fields of that type*. Consider the following declarations that use the *point* and *point3D* types from the previous examples:

```
var
    P2: point;
    P3: point3D;
    p: pointer to point;
    .
    .
    .
    lea( ebx, P2 );
    mov( ebx, p );
    p.distance();    // Calls the point.distance method.
    .
    .
    .
    lea( ebx, P3 );
    mov( ebx, p );    // Yes, this is semantically legal.
    p.distance();    // Surprise, this calls point3D.distance.
```

Since *p* is a pointer to a *point* object, it might seem intuitive for *p.distance* to call the *point.distance* method. However, methods are *polymorphic*. If you've got a pointer to an object

1. Polymorphism literally means "many-faced." In the context of object-oriented programming polymorphism means that the same method name, e.g., *distance*, and refer to one of several different methods.

1. Of course, assembly language programmers break rules like this all the time. For now, let's assume we're playing by the rules and only access the data using the data type associated with the pointer.

and you call a method associated with that object, the system will call the actual (overridden) method associated with the object, not the method specifically associated with the pointer's class type.

Class procedures behave differently than methods with respect to overridden procedures. When you call a class procedure indirectly through an object pointer, the system will always call the procedure associated with the underlying class associated with the pointer. So had *distance* been a procedure rather than a method in the previous examples, the "p.distance();" invocation would always call *point.distance*, even if *p* is pointing at a *point3D* object. The section on Object Initialization, later in this chapter, explains why methods and procedures are different (see "Object Implementation" on page 479).

Note that iterators are also virtual; so like methods an object iterator invocation will always call the (overridden) iterator associated with the actual object whose address the pointer contains. To differentiate the semantics of methods and iterators from procedures, we will refer to the method/iterator calling semantics as *virtual procedures* and the calling semantics of a class procedure as a *static procedure*.

18.5.9 Writing Class Methods, Iterators, and Procedures

For each class procedure, method, and iterator prototype appearing in a class definition, there must be a corresponding procedure, method, or iterator appearing within the program (for the sake of brevity, this section will use the term *routine* to mean procedure, method, or iterator from this point forward). If the prototype does not contain the EXTERNAL option, then the code must appear in the same compilation unit as the class declaration. If the EXTERNAL option does follow the prototype, then the code may appear in the same compilation unit or a different compilation unit (as long as you link the resulting object file with the code containing the class declaration). Like external (non-class) procedures and iterators, if you fail to provide the code the linker will complain when you attempt to create an executable file. To reduce the size of the following examples, they will all define their routines in the same source file as the class declaration.

HLA class routines must always follow the class declaration in a compilation unit. If you are compiling your routines in a separate unit, the class declarations must still precede the code with the class declaration (usually via an #INCLUDE file). If you haven't defined the class by the time you define a routine like *point.distance*, HLA doesn't know that *point* is a class and, therefore, doesn't know how to handle the routine's definition.

Consider the following declarations for a point2D class:

```
type
  point2D: class

    const
      UnitDistance: real32 := 1.0;

    var
      x: real32;
      y: real32;

    static
      LastDistance: real32;

    method distance( fromX: real32; fromY:real32 ); returns( "st0" );
    procedure InitLastDistance;

  endclass;
```

The distance function for this class should compute the distance from the object's point to (fromX,fromY). The following formula describes this computation:

$$\sqrt{(x - fromX)^2 + (y - fromY)^2}$$

A first pass at writing the distance method might produce the following code:

```

method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( x );           // Note: this doesn't work!
    fld( fromX );       // Compute (x-fromX)
    fsub();
    fld( st0 );         // Duplicate value on TOS.
    fmul();             // Compute square of difference.

    fld( y );           // This doesn't work either.
    fld( fromY );       // Compute (y-fromY)
    fsub();
    fld( st0 );         // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;

```

This code probably looks like it should work to someone who is familiar with an object-oriented programming language like C++ or Delphi. However, as the comments indicate, the instructions that push the *x* and *y* variables onto the FPU stack don't work – HLA doesn't automatically define the symbols associated with the data fields of a class within that class' routines.

To learn how to access the data fields of a class within that class' routines, we need to back up a moment and discover some very important implementation details concerning HLA's classes. To do this, consider the following variable declarations:

```

var
    Origin: point2D;
    PtInSpace: point2D;

```

Remember, whenever you create two objects like *Origin* and *PtInSpace*, HLA reserves storage for the *x* and *y* data fields for both of these objects. However, there is only one copy of the *point2D.distance* method in memory. Therefore, were you to call *Origin.distance* and *PtInSpace.distance*, the system would call the same routine for both method invocations. Once inside that method, one has to wonder what an instruction like “fld(x);” would do. How does it associate *x* with *Origin.x* or *PtInSpace.x*? Worse still, how would this code differentiate between the data field *x* and a global object *x*? In HLA, the answer is “it doesn't.” You do not specify the data field names within a class routine by simply using their names as though they were common variables.

To differentiate *Origin.x* from *PtInSpace.x* within class routines, HLA automatically passes a pointer to an object's data fields whenever you call a class routine. Therefore, you can reference the data fields indirectly off this pointer. HLA passes this object pointer in the ESI register. This is one of the few places where HLA-generated code will modify one of the 80x86 registers behind your back: **anytime you call a class routine, HLA automatically loads the ESI register with the object's address.** Obviously, you cannot count on ESI's value being preserved across class routine class nor can you pass parameters to the class routine in the ESI register (though it is perfectly reasonable to specify “@USE ESI;” to allow HLA to use the ESI register when setting up other parameters). For class methods and iterators (but not procedures), HLA will also load the EDI register with the address of the class' *virtual method table* (see “Virtual Method Tables” on page 482). While the virtual method table address isn't as interesting as the object address, keep in mind that **HLA-generated code will overwrite any value in the EDI register when you call a method or an iterator.** Again, “EDI” is a good choice for the @USE operand for methods since HLA will wipe out the value in EDI anyway.

Upon entry into a class routine, ESI contains a pointer to the (non-static) data fields associated with the class. Therefore, to access fields like *x* and *y* (in our *point2D* example), you could use an address expression like the following:

```
(type point2D [esi].x
```

Since you use ESI as the base address of the object's data fields, it's a good idea not to disturb ESI's value within the class routines (or, at least, preserve ESI's value if you need to access the object's data fields after some point where you must use ESI for some other purpose). Note that if you call an iterator or a method you do not have to preserve EDI (unless, for some reason, you need access to the virtual method table, which is unlikely).

Accessing the fields of a data object within a class' routines is such a common operation that HLA provides a shorthand notation for casting ESI as a pointer to the class object: THIS. Within a class in HLA, the reserved word THIS automatically expands to a string of the form "(type *classname* [esi])" substituting, of course, the appropriate class name for *classname*. Using the THIS keyword, we can (correctly) rewrite the previous distance method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );      // Duplicate value on TOS.
    fmul();          // Compute square of difference.

    fld( this.y );
    fld( fromY );    // Compute (y-fromY)
    fsub();
    fld( st0 );      // Compute the square of the difference.
    fmul();

    fsqrt();

end distance;
```

Don't forget that calling a class routine wipes out the value in the ESI register. This isn't obvious from the syntax of the routine's invocation. It is especially easy to forget this when calling some class routine from inside some other class routine; don't forget that if you do this the internal call wipes out the value in ESI and on return from that call ESI no longer points at the original object. Always push and pop ESI (or otherwise preserve ESI's value) in this situation, e.g.,

```
.
.
.
fld( this.x );    // ESI points at current object.
.
.
.
push( esi );      // Preserve ESI across this method call.
SomeObject.SomeMethod();
pop( esi );
.
.
.
lea( ebx, this.x );    // ESI points at original object here.
```

The THIS keyword provides access to the class variables you declare in the VAR section of a class. You can also use THIS to call other class routines associated with the current object, e.g.,

```
this.distance( 5.0, 6.0 );
```

To access class constants and STATIC data fields you generally do not use the THIS pointer. HLA associates constant and static data fields with the whole class, not a specific object. To access these class members, just use the class name in place of the object name. For example, to access the *UnitDistance* constant in the *point2D* class you could use a statement like the following:

```
fld( point2D.UnitDistance );
```

As another example, if you wanted to update the *LastDistance* field in the *point2D* class each time you computed a distance, you could rewrite the *point2D.distance* method as follows:

```
method point2D.distance( fromX:real32; fromY:real32 ); nodisplay;
begin distance;

    fld( this.x );
    fld( fromX );    // Compute (x-fromX)
    fsub();
    fld( st0 );     // Duplicate value on TOS.
    fmul();         // Compute square of difference.

    fld( this.y );
    fld( fromY );   // Compute (y-fromY)
    fsub();
    fld( st0 );     // Compute the square of the difference.
    fmul();

    fsqrt();

    fst( point2D.LastDistance ); // Update shared (STATIC) field.

end distance;
```

To understand why you use the class name when referring to constants and static objects but you use THIS to access VAR objects, check out the next section.

Class procedures are also static objects, so it is possible to call a class procedure by specifying the class name rather than an object name in the procedure invocation, e.g., both of the following are legal:

```
Origin.InitLastDistance();
point2D.InitLastDistance();
```

There is, however, a subtle difference between these two class procedure calls. The first call above loads ESI with the address of the *Origin* object prior to actually calling the *InitLastDistance* procedure. The second call, however, is a direct call to the class procedure without referencing an object; therefore, HLA doesn't know what object address to load into the ESI register. In this case, HLA loads NULL (zero) into ESI prior to calling the *InitLastDistance* procedure. Because you can call class procedures in this manner, it's always a good idea to check the value in ESI within your class procedures to verify that HLA contains an object address. Checking the value in ESI is a good way to determine which calling mechanism is in use. Later, this chapter will discuss constructors and object initialization; there you will see a good use for static procedures and calling those procedures directly (rather than through the use of an object).

18.5.10 Object Implementation

In a high level object-oriented language like C++ or Delphi, it is quite possible to master the use of objects without really understanding how the machine implements them. One of the reasons for learning assembly language programming is to fully comprehend low-level implementation details so one can make educated decisions concerning the use of programming constructs like

objects. Further, since assembly language allows you to poke around with data structures at a very low-level, knowing how HLA implements objects can help you create certain algorithms that would not be possible without a detailed knowledge of object implementation. Therefore, this section, and its corresponding subsections, explains the low-level implementation details you will need to know in order to write object-oriented HLA programs.

HLA implements objects in a manner quite similar to records. In particular, HLA allocates storage for all VAR objects in a class in a sequential fashion, just like records. Indeed, if a class consists of only VAR data fields, the memory representation of that class is nearly identical to that of a corresponding RECORD declaration. Consider the following Student record declaration and the corresponding class:

```
type
  student: record
    Name: char[65];
    Major: int16;
    SSN: char[12];
    Midterm1: int16;
    Midterm2: int16;
    Final: int16;
    Homework: int16;
    Projects: int16;
  endrecord;

  student2: class
    Name: char[65];
    Major: int16;
    SSN: char[12];
    Midterm1: int16;
    Midterm2: int16;
    Final: int16;
    Homework: int16;
    Projects: int16;
  endclass;
```



Student RECORD Implementation in Memory



Student CLASS Implementation in Memory

If you look carefully at these two figures, you'll discover that the only difference between the class and the record implementations is the inclusion of the VMT (virtual method table) pointer field at the beginning of the class object. This field, which is always present in a class, contains the address of the class' virtual method table which, in turn, contains the addresses of all the class' methods and iterators. The VMT field, by the way, is present even if a class doesn't contain any methods or iterators.

As pointed out in previous sections, HLA does not allocate storage for STATIC objects within the object's storage. Instead, HLA allocates a single instance of each static data field that all objects share. As an example, consider the following class and object declarations:

```

type
  tHasStatic: class

    var
      i:int32;
      j:int32;
      r:real32;

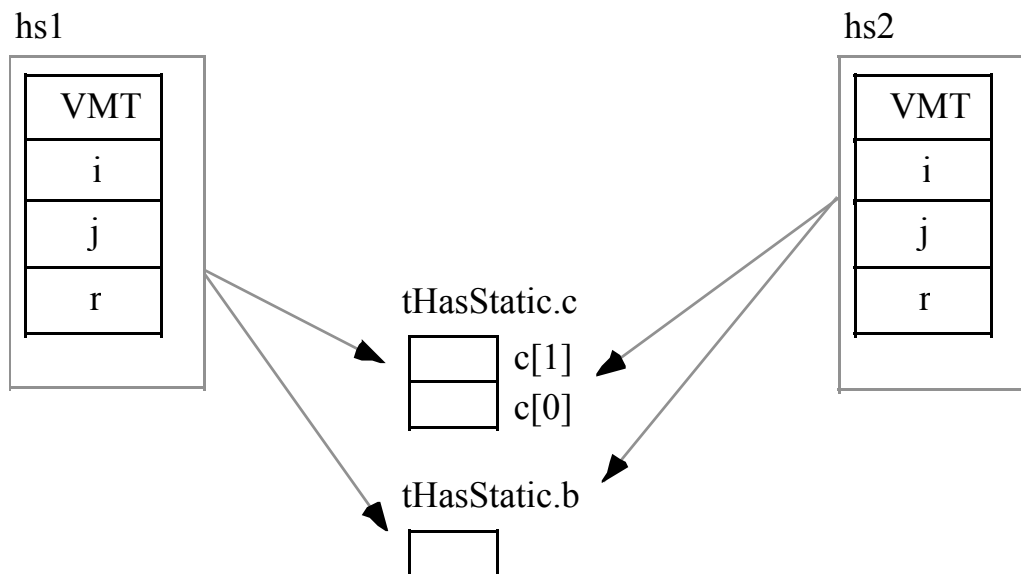
    static
      c:char[2];
      b:byte;

  endclass;

var
  hs1: tHasStatic;
  hs2: tHasStatic;

```

shows the storage allocation for these two objects in memory.



Object Allocation with Static Data Fields

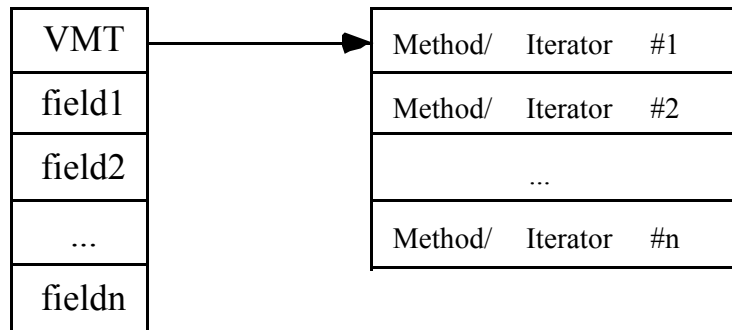
Of course, `CONST`, `VAL`, and `#MACRO` objects do not have any run-time memory requirements associated with them, so HLA does not allocate any storage for these fields. Like the `STATIC` data fields, you may access `CONST`, `VAL`, and `#MACRO` fields using the class name as well as an object name. Hence, even if `tHasStatic` has these types of fields, the memory organization for `tHasStatic` objects would still be the same as shown in .

Other than the presence of the virtual method table pointer (VMT), the presence of methods, iterators, and procedures has no impact on the storage allocation of an object. Of course, the machine instructions associated with these routines does appear somewhere in memory. So in a sense the code for the routines is quite similar to static data fields insofar as all the objects share a single instance of the routine.

18.5.10.1 Virtual Method Tables

When HLA calls a class procedure, it directly calls that procedure using a `CALL` instruction, just like any normal non-class procedure call. Methods and iterators are another story altogether. Each object in the system carries a pointer to a virtual method table which is an array of pointers to all the methods and iterators appearing within the object's class.

SomeObject



Virtual Method Table Organization

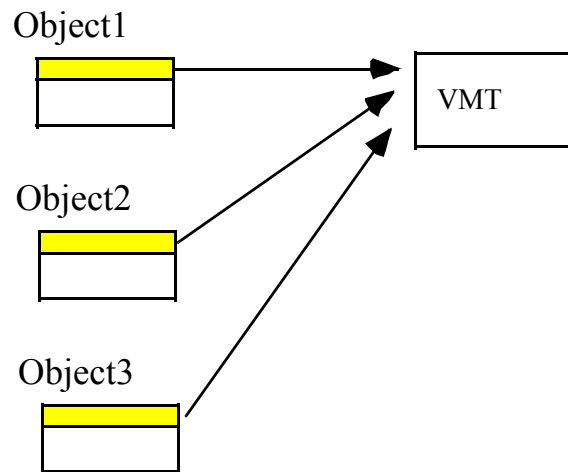
Each iterator or method you declare in a class has a corresponding entry in the virtual method table. That dword entry contains the address of the first instruction of that iterator or method. To call a class method or iterator is a bit more work than calling a class procedure (it requires one additional instruction plus the use of the EDI register). Here is a typical calling sequence for a method:

```

mov( ObjectAdrs, ESI );      // All class routines do this.
mov( [esi], edi );         // Get the address of the VMT into
EDI
call( (type dword [edi+n])); // "n" is the offset of the method's
entry                       // in the VMT.

```

For a given class there is only one copy of the VMT in memory. This is a static object so all objects of a given class type share the same VMT. This is reasonable since all objects of the same class type have exactly the same methods and iterators (see).



Note: Objects are all the same class type

All Objects That are the Same Class Type Share the Same VMT

Although HLA builds the VMT record structure as it encounters methods and iterators within a class, HLA does not automatically create the actual run-time virtual method table for you. You must explicitly declare this table in your program. To do this, you include a statement like the following in a STATIC or READONLY declaration section of your program, e.g.,

```
readonly
    VMT( classname );
```

Since the addresses in a virtual method table should never change during program execution, the READONLY section is probably the best choice for declaring VMTs. It should go without saying that changing the pointers in a VMT is, in general, a really bad idea. So putting VMTs in a STATIC section is usually not a good idea.

A declaration like the one above defines the variable *classname.VMT*. In section 18.5.11 (see “Constructors and Object Initialization” on page 487) you see that you’ll need this name when initializing object variables. The class declaration automatically defines the *classname.VMT* symbol as an external static variable. The declaration above just provides the actual definition of this external symbol.

The declaration of a VMT uses a somewhat strange syntax because you aren’t actually declaring a new symbol with this declaration, you’re simply supplying the data for a symbol that you previously declared implicitly by defining a class. That is, the class declaration defines the static table variable *classname.VMT*, all you’re doing with the VMT declaration is telling HLA to emit the actual data for the table. If, for some reason, you would like to refer to this table using a name other than *classname.VMT*, HLA does allow you to prefix the declaration above with a variable name, e.g.,

```
readonly
    myVMT: VMT( classname );
```

In this declaration, *myVMT* is an alias of *classname.VMT*. As a general rule, you should avoid aliases in a program because they make the program more difficult to read and understand. Therefore, it is unlikely that you would ever really need to use this type of declaration.

Like any other global static variable, there should be only one instance of a VMT for a given class in a program. The best place to put the VMT declaration is in the same source file as the

class' method, iterator, and procedure code (assuming they all appear in a single file). This way you will automatically link in the VMT whenever you link in the routines for a given class.

18.5.10.2 Object Representation with Inheritance

Up to this point, the discussion of the implementation of class objects has ignored the possibility of inheritance. Inheritance only affects the memory representation of an object by adding fields that are not explicitly stated in the class declaration.

Adding inherited fields from a *base class* to another class must be done carefully. Remember, an important attribute of a class that inherits fields from a base class is that you can use a pointer to the base class to access the inherited fields from that base class in another class. As an example, consider the following classes:

```

type
  tBaseClass: class
    var
      i:uns32;
      j:uns32;
      r:real32;

    method mBase;
  endclass;

  tChildClassA: class inherits( tBaseClass );
    var
      c:char;
      b:boolean;
      w:word;

    method mA;
  endclass;

  tChildClassB: class inherits( tBaseClass );
    var
      d:dword;
      c:char;
      a:byte[3];
  endclass;

```

Since both *tChildClassA* and *tChildClassB* inherit the fields of *tBaseClass*, these two child classes include the *i*, *j*, and *r* fields as well as their own specific fields. Furthermore, whenever you have a pointer variable whose base type is *tBaseClass*, it is legal to load this pointer with the address of any child class of *tBaseClass*; therefore, it is perfectly reasonable to load such a pointer with the address of a *tChildClassA* or *tChildClassB* variable, e.g.,

```

var
  B1: tBaseClass;
  CA: tChildClassA;
  CB: tChildClassB;
  ptr: pointer to tBaseClass;
  .
  .
  .
  lea( ebx, B1 );
  mov( ebx, ptr );
  << Use ptr >>
  .
  .

```

```

.
lea( eax, CA );
mov( ebx, ptr );
<< Use ptr >>
.
.
.
lea( eax, CB );
mov( eax, ptr );
<< Use ptr >>

```

Since *ptr* points at an object of *tBaseClass*, you may legally (from a semantic sense) access the *i*, *j*, and *r* fields of the object where *ptr* is pointing. It is not legal to access the *c*, *b*, *w*, or *d* fields of the *tChildClassA* or *tChildClassB* objects since at any one given moment the program may not know exactly what object type *ptr* references.

In order for inheritance to work properly, the *i*, *j*, and *r* fields must appear at the same offsets all child classes as they do in *tBaseClass*. This way, an instruction of the form “mov((type *tBaseClass* [ebx]).i, eax);” will correct access the *i* field even if EBX points at an object of type *tChildClassA* or *tChildClassB*. shows the layout of the child and base classes:



Layout of Base and Child Class Objects in Memory

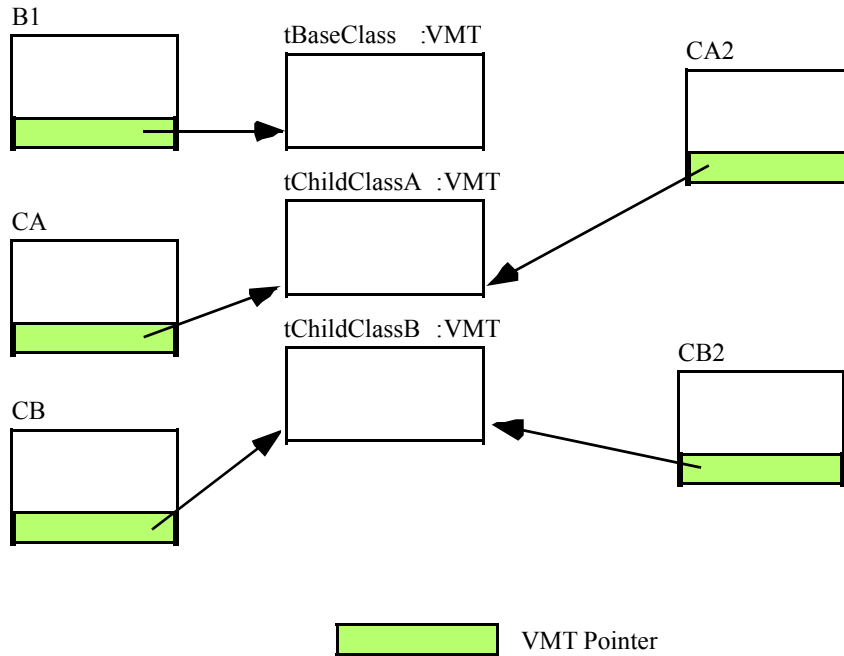
Note that the new fields in the two child classes bear no relation to one another, even if they have the same name (e.g., field *c* in the two child classes does not lie at the same offset). Although the two child classes share the fields they inherit from their common base class, any new fields they add are unique and separate. Two fields in different classes share the same offset only by coincidence.

All classes (even those that aren't related to one another) place the pointer to the virtual method table at offset zero within the object. There is a single VMT associated with each class in a program; even classes that inherit fields from some base class have a VMT that is (generally) different than the base class' VMT. shows how objects of type *tBaseClass*, *tChildClassA* and *tChildClassB* point at their specific VMTs:

```

var
  B1: tBaseClass ;
  CA: tChildClassA ;
  CB: tChildClassB ;
  CB2: tChildClassB ;
  CA2: tChildClassA ;

```



Virtual Method Table References from Objects

A virtual method table is nothing more than an array of pointers to the methods and iterators associated with a class. The address of the first method or iterator appearing in a class is at offset zero, the address of the second appears at offset four, etc. You can determine the offset value for a given iterator or method by using the `@offset` function. If you want to call a method or iterator directly (using 80x86 syntax rather than HLA's high level syntax), you code use code like the following:

```

var
  sc: tBaseClass;
  .
  .
  .
  lea( esi, sc );           // Get the address of the object (& VMT).
  mov( [esi], edi );       // Put address of VMT into EDI.
  call( (type dword [edi+@offset( tBaseClass.mBase )] ) );

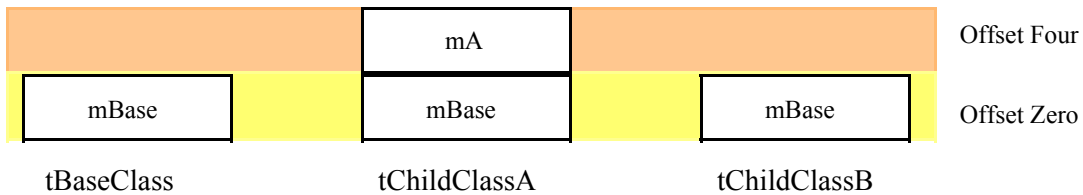
```

Of course, if the method has any parameters, you must push them onto the stack before executing the code above. Don't forget, when making direct calls to a method, that you must load ESI with the address of the object. Any field references within the method will probably depend upon ESI containing this address. The choice of EDI to contain the VMT address is nearly arbitrary. Unless you're doing something tricky (like using EDI to obtain run-time type information), you could use any register you please here. As a general rule, you should use EDI

when simulating class iterator/method calls because this is the convention that HLA employs and most programmers will expect this.

Whenever a child class inherits fields from some base class, the child class' VMT also inherits entries from the base class' VMT. For example, the VMT for class *tBaseClass* contains only a single entry – a pointer to method *tBaseClass.mBase*. The VMT for class *tChildClassA* contains two entries: a pointer to *tBaseClass.mBase* and *tChildClassA.mA*. Since *tChildClassB* doesn't define any new methods or iterators, *tChildClassB*'s VMT contains only a single entry, a pointer to the *tBaseClass.mBase* method. Note that *tChildClassB*'s VMT is identical to *tBaseClass*' VMT. Nevertheless, HLA produces two distinct VMTs. This is a critical fact that we will make use of a little later. shows the relationship between these VMTs:

Virtual Method Tables for Derived (inherited) Classes



Virtual Method Tables for Inherited Classes

Although the VMT always appears at offset zero in an object (and, therefore, you can access the VMT using the address expression “[ESI]” if ESI points at an object), HLA actually inserts a symbol into the symbol table so you may refer to the VMT symbolically. The symbol *_pVMT_* (pointer to Virtual Method Table) provides this capability. So a more readable way to access the VMT pointer (as in the previous code example) is

```
lea( esi, sc );
mov( (type tBaseClass [esi])._pVMT_, edi );
call( (type dword [edi+@offset( tBaseClass.mBase )] ) );
```

If you need to access the VMT directly, there are a couple ways to do this. Whenever you declare a class object, HLA automatically includes a field named *_VMT_* as part of that class. *_VMT_* is a static array of double word objects. Therefore, you may refer to the VMT using an identifier of the form *classname._VMT_*. Generally, you shouldn't access the VMT directly, but as you'll see shortly, there are some good reasons why you need to know the address of this object in memory.

18.5.11 Constructors and Object Initialization

If you've tried to get a little ahead of the game and write a program that uses objects prior to this point, you've probably discovered that the program inexplicably crashes whenever you attempt to run it. We've covered a lot of material in this chapter thus far, but you are still missing one crucial piece of information – how to properly initialize objects prior to use. This section will put the final piece into the puzzle and allow you to begin writing programs that use classes.

Consider the following object declaration and code fragment:

```
var
  bc: tBaseClass;
  .
  .
  .
  bc.mBase();
```


Remember that variables you declare in the VAR section are uninitialized at run-time. Therefore, when the program containing these statements gets around to executing *bc.mBase*, it executes the three-statement sequence you've seen several times already:

```
lea( esi, bc);
mov( [esi], edi );
call( (type dword [edi+@offset( tBaseClass.mBase ) ] ) );
```

The problem with this sequence is that it loads EDI with an undefined value assuming you haven't previously initialized the *bc* object. Since EDI contains a garbage value, attempting to call a subroutine at address “[EDI+@offset(tBaseClass.mBase)]” will likely crash the system. Therefore, before using an object, you must initialize the *_pVMT_* field with the address of that object's VMT. One easy way to do this is with the following statement:

```
mov( &tBaseClass._VMT_, bc._pVMT_ );
```

Always remember, **before using an object, be sure to initialize the virtual method table pointer for that field.**

Although you must initialize the virtual method table pointer for all objects you use, this may not be the only field you need to initialize in those objects. Each specific class may have its own application-specific initialization that is necessary. Although the initialization may vary by class, you need to perform the same initialization on each object of a specific class that you use. If you ever create more than a single object from a given class, it is probably a good idea to create a procedure to do this initialization for you. This is such a common operation that object-oriented programmers have given these initialization procedures a special name: *constructors*.

Some object-oriented languages (e.g., C++) use a special syntax to declare a constructor. Others (e.g., Delphi) simply use existing procedure declarations to define a constructor. One advantage to employing a special syntax is that the language knows when you define a constructor and can automatically generate code to call that constructor for you (whenever you declare an object). Languages, like Delphi, require that you explicitly call the constructor; this can be a minor inconvenience and a source of defects in your programs. HLA does not use a special syntax to declare constructors – you define constructors using standard class procedures. As such, you will need to explicitly call the constructors in your program; however, you'll see an easy method for automating this in a later section of this chapter.

Perhaps the most important fact you must remember is that **constructors must be class procedures**. You must not define constructors as methods (or iterators). The reason is quite simple: one of the tasks of the constructor is to initialize the pointer to the virtual method table and you cannot call a class method or iterator until after you've initialized the VMT pointer. Since class procedures don't use the virtual method table, you can call a class procedure prior to initializing the VMT pointer for an object.

By convention, HLA programmers use the name *Create* for the class constructor. There is no requirement that you use this name, but by doing so you will make your programs easier to read and follow by other programmers.

As you may recall, you can call a class procedure via an object reference or a class reference. E.g., if *clsProc* is a class procedure of class *tClass* and *Obj* is an object of type *tClass*, then the following two class procedure invocations are both legal:

```
tClass.clsProc();
Obj.clsProc();
```

There is a big difference between these two calls. The first one calls *clsProc* with ESI containing zero (NULL) while the second invocation loads the address of *Obj* into ESI before the call. We can use this fact to determine within a method the particular calling mechanism.

18.5.12 Dynamic Object Allocation Within the Constructor

As it turns out, most programs allocate objects dynamically using *malloc* and refer to those objects indirectly using pointers. This adds one more step to the initialization process – allocating storage for the object. The constructor is the perfect place to allocate this storage. Since you probably won't need to allocate all objects dynamically, you'll need two types of constructors: one

that allocates storage and then initializes the object, and another that simply initializes an object that already has storage.

Another constructor convention is to merge these two constructors into a single constructor and differentiate the type of constructor call by the value in ESI. On entry into the class' *Create* procedure, the program checks the value in ESI to see if it contains NULL (zero). If so, the constructor calls *malloc* to allocate storage for the object and returns a pointer to the object in ESI. If ESI does not contain NULL upon entry into the procedure, then the constructor assumes that ESI points at a valid object and skips over the memory allocation statements. At the very least, a constructor initializes the pointer to the VMT; therefore, the minimalist constructor will look like the following:

```
procedure tBaseClass.mBase; nodisplay;
begin mBase;

    if( ESI = 0 ) then

        push( eax );    // Malloc returns its result here, so save it.
        malloc( @size( tBaseClass ) );
        mov( eax, esi ); // Put pointer into ESI;
        pop( eax );

    endif;

    // Initialize the pointer to the VMT:
    // (remember, "this" is shorthand for (type tBaseClass [esi])"

    mov( &tBaseClass._VMT_, this._pVMT_ );

    // Other class initialization would go here.

end mBase;
```

After you write a constructor like the one above, you choose an appropriate calling mechanism based on whether your object's storage is already allocated. For pre-allocated objects (i.e., those you've declared in VAR, STATIC, or STORAGE sections¹ or those you've previously allocated storage for via *malloc*) you simply load the address of the object into ESI and call the constructor. For those objects you declare as a variable, this is very easy – just call the appropriate *Create* constructor:

```
var
    bc0: tBaseClass;
    bcp: pointer to tBaseClass;
    .
    .
    .
    bc0.Create(); // Initializes pre-allocated bc0 object.
    .
    .
    .
    malloc( @size( tBaseClass ) ); // Allocate storage for bcp object.
    mov( eax, bcp );
    .
    .
    .
    bcp.Create(); // Initializes pre-allocated bcp object.
```

1. You generally do not declare objects in READONLY sections because you cannot initialize them.

Note that although *bcp* is a pointer to a *tBaseClass* object, the *Create* method does not automatically allocate storage for this object. The program already allocates the storage earlier. Therefore, when the program calls *bcp.Create* it loads ESI with the address contained within *bcp*; since this is not NULL, the *tBaseClass.Create* procedure does not allocate storage for a new object. By the way, the call to *bcp.Create* emits the following sequence of machine instructions:

```
mov( bcp, esi );  
call tBaseClass.Create;
```

Until now, the code examples for a class procedure call always began with an LEA instruction. This is because all the examples to this point have used object variables rather than pointers to object variables. Remember, a class procedure (method/iterator) call passes the address of the object in the ESI register. For object variables HLA emits an LEA instruction to obtain this address. For pointers to objects, however, the actual object address is the *value* of the pointer variable; therefore, to load the address of the object into ESI, HLA emits a MOV instruction that copies the value of the pointer into the ESI register.

In the example above, the program preallocates the storage for an object prior to calling the object constructor. While there are several reasons for preallocating object storage (e.g., you're creating a dynamic array of objects), you can achieve most simple object allocations like the one above by calling a standard *Create* method (i.e., one that allocates storage for an object if ESI contains NULL). The following example demonstrates this:

```
var  
  bcp2: pointer to tBaseClass;  
  .  
  .  
  .  
  tBaseClass.Create(); // Calls Create with ESI=NULL.  
  mov( esi, bcp2 ); // Save pointer to new class object in bcp2.
```

Remember, a call to a *tBaseClass.Create* constructor returns a pointer to the new object in the ESI register. It is the caller's responsibility to save the pointer this function returns into the appropriate pointer variable; the constructor does not automatically do this for you.

18.6 Compiling Resource Scripts Using HLA

HLA's compile-time language facilities provide the ability to embed domain-specific languages directly in an HLA source file. This paper discusses how to create a domain-specific embedded language that handles Windows Resources. This mini-language not only provides access to these resources in your HLA source files, but it also creates a resource script file (.rc file) that you may compile with the Microsoft Resource Compiler (RC.EXE).

18.6.1 The Motivation

Working with resources when writing Win32 assembly language programs is usually a two-step process. First, you write some assembly code that requests a resource object from the executable file; then you write a resource script file that matches the resources, via some numeric identifier, with the actual resource file on the disk. The problem with this approach is that you have to maintain (and keep consistent) two sets of source files - an HLA/assembly source file and a resource script (.rc) file. The reason you have to maintain two files is because the assembler associates names with numeric values in a different way than Microsoft's resource compiler. The resource compiler uses C's "#define" syntax, which is not compatible with constant declarations in assembly language. Therefore, you have to create a set of definitions like the following for the resource compiler:

```
#define resource_1 101
#define resource_2 102
#define resource_3 2005
```

When working in assembly language (e.g., HLA), you need to use statements like the following to declare these symbolic names with these values:

```
const
    resource_1 := 101;
    resource_2 := 102;
    resource_3 := 2005;
```

Although entering these two sets of constant definitions twice is a big pain, the real problem comes when you modify either set of definitions and find that you need to edit the other set to keep them consistent. At the very least, we'd like to be able to maintain one set of declarations to avoid consistency problems.

Another problem with using resource scripts is that you have to maintain two separate files - an assembly language source file and a resource script file. While breaking up programs into multiple files isn't always a bad idea, the resource file often contains common things (like strings) that you'd like to find easily when working on small assembly projects. Sometimes, it's just more convenient to put the resources in the same source file as your assembly source code. One final problem with using a separate resource file is that the resource scripting language is radically different from assembly syntax. It would be nice to be able to declare resources in an assembly language source file like any other object and have the assembler handle the details of creating the resource script (or compiling the resource) for you.

18.6.2 The HLA Solution

Although processing script files is a pipe dream within most assemblers, HLA's compile-time language provides sufficient capability to achieve this. Here are some of the HLA features that give us the capability to create our own language within HLA:

- Context-free macros
- The ability to create (user-defined) output files during assembly
- The ability to execute system commands during assembly
- Conditional assembly and compile-time loops
- Powerful compile-time string processing facilities

The approach we will take here is to define a new "HLA declaration section" using a context-free macro. Within this section a programmer will declare Win32 resource objects. HLA will create

a resource script (.rc) file on the basis of the data appearing in this section, and will define a set of symbolic constants by which the rest of the HLA program can refer to those objects. The basic syntax for this new section will be the following:

```
resource( "filename.rc" )
  <<resource definitions>>
endresource;
```

Between the `resource` and `endresource` statements, this code will construct the resource script file using the filename you specify in the `resource` statement. Upon encountering the `endresource` statement, HLA will close the script file and then execute Microsoft's "rc.exe" program to compile the resource code. The declarations between these two statements will also generate symbols that the HLA code can use. In general, there will only be a single resource declaration section in any one given HLA source file; the design of the macros that handle this declaration section will assume that this is the case. In particular, you should avoid nesting `resource/endresource` declaration sections. Though HLA allows this syntax, the efficiency of the macros' execution (at compile-time) is based on the assumption that you've only got one `resource/endresource` declaration section in an HLA program.

18.6.3 The Resource..Endresource Declaration Section

To Be Written....

18.7 Structures in Assembly Language Programs

Structures, or records, are an abstract data type that allows a programmer to collect different objects together into a single, composite, object. Structures can help make programs easier to read, write, modify, and maintain. Used appropriately, they can also help your programs run faster. Despite the advantages that structures offer, their appearance in assembly language is a relatively recent phenomenon (in the past two decades, or so), and many assemblers still do not support this facility. Furthermore, many "old-timer" assembly language programmers attempt to argue that the appearance of records violates the whole principle of "assembly language programming." This article will certain refute such arguments and describe the benefits of using structures in an assembly language program.

Despite the fact that records have been available in various assembly languages for years (e.g., Microsoft's MASM assembler introduced structures in 80x86 assembly language in the 1980s), the "lack of support for structures" is a common argument against assembly language by HLL programmers who don't know much about assembly. In some respects, their ignorance is justified - many assemblers don't support structures or records. A second goal of this article is to educate assembly language programmers to counter claims like "assembly language doesn't support structures." Hopefully, that same education will convince those assembly language programmers who've never bothered to use structures, to consider their use.

This article will use the term "record" to denote a structure/record to avoid confusion with the more general term "data structure". Note, however, that the terms "record" and "structure" are synonymous in this article.

18.7.1 What is a Record (Structure)?

The whole purpose of a record is to let you encapsulate different, but logically related, data into a single package. Here is a typical record declaration, in HLA using the RECORD / ENDRECORD declaration:

```
type
  student:
    record
      Name:      string;
      Major:     int16;
      SSN:       char[12];
      Midterm1:  int16;
      Midterm2:  int16;
      Final:     int16;
      Homework:  int16;
      Projects:  int16;
    endrecord;
```

The field names within the record must be unique. That is, the same name may not appear two or more times in the same record. However, in reasonable assemblers (like HLA) that support true structures, all the field names are local to that record. With such assemblers, you may reuse those field names elsewhere in the program.

The RECORD/ENDRECORD type declaration may appear in a variable declaration section (e.g., an HLA STATIC or VAR section) or in a TYPE declaration section. In the previous example the *Student* declaration appears in an HLA TYPE section, so this does not actually allocate any storage for a *Student* variable. Instead, you have to explicitly declare a variable of type *Student*. The following example demonstrates how to do this:

```
var
  John: Student;
```

This allocates 28 bytes of storage: four bytes for the Name field (HLA strings are four-byte pointers to character data found elsewhere in memory), 12 bytes for the SSN field, and two bytes for each of the other six fields.

If the label *John* corresponds to the *base address* of this record, then the *Name* field is at offset *John+0*, the *Major* field is at offset *John+4*, the *SSN* field is at offset *John+6*, etc.

To access an element of a structure you need to know the offset from the beginning of the structure to the desired field. For example, the *Major* field in the variable *John* is at offset 4 from the base address of *John*. Therefore, you could store the value in AX into this field using the instruction

```
mov( ax, (type word John[4]) );
```

Unfortunately, memorizing all the offsets to fields in a record defeats the whole purpose of using them in the first place. After all, if you've got to deal with these numeric offsets why not just use an array of bytes instead of a record?

Well, as it turns out, assemblers like HLA that support true records commonly let you refer to field names in a record using the same mechanism C/C++ and Pascal use: the dot operator. To store AX into the *Major* field, you could use “mov(ax, John.Major);” instead of the previous instruction. This is much more readable and certainly easier to use.

18.7.2 Record Constants

HLA lets you define record constants. In fact, HLA is probably unique among x86 assemblers insofar as it supports both symbolic record constants and literal record constants. Record constants are useful as initializers for static record variables. They are also quite useful as compile-time data structures when using the HLA compile-time language (that is, the macro processor language). This section discusses how to create record constants.

A record literal constant takes the following form:

```
RecordTypeName: [ List_of_comma_separated_constants ]
```

The *RecordTypeName* is the name of a record data type you've defined in an HLA TYPE section prior to this point. To create a record constant you must have previously defined the record type in a TYPE section of your program.

The constant list appearing between the brackets are the data items for each of the fields in the specified record. The first item in the list corresponds to the first field of the record, the second item in the list corresponds to the second field, etc. The data types of each of the constants appearing in this list must match their respective field types. The following example demonstrates how to use a literal record constant to initialize a record variable:

```
type
  point:
    record
      x:int32;
      y:int32;
      z:int32;
    endrecord;

static
  Vector: point := point:[ 1, -2, 3 ];
```

This declaration initializes *Vector.x* with 1, *Vector.y* with -2, and *Vector.z* with 3.

You can also create symbolic record constants by declaring record objects in the CONST or VAL sections of an HLA program. You access fields of these symbolic record constants just as you would access the field of a record variable, using the dot operator. Since the object is a constant, you can specify the field of a record constant anywhere a constant of that field's type is legal. You can also employ symbolic record constants as record variable initializers. The following example demonstrates this:

```
type
  point:
    record
      x:int32;
```

```

        y:int32;
        z:int32;
    endrecord;

const
    PointInSpace: point := point:[ 1, 2, 3 ];

static
    Vector: point := PointInSpace;
    XCoord: int32 := PointInSpace.x;

```

18.7.3 Arrays of Records

It is a perfectly reasonable operation to create an array of records. To do so, you simply create a record type and then use the standard array declaration syntax when declaring an array of that record type. The following example demonstrates how you could do this:

```

type
    recElement:
        record
            << fields for this record >>
        endrecord;
    .
    .
    .
static
    recArray: recElement[4];

```

Naturally, you can create multidimensional arrays of records as well. You would use the standard row or column major order functions to compute the address of an element within such records. The only thing that really changes (from the discussion of arrays) is that the size of each element is the size of the record object.

```

static
    rec2D: recElement[ 4, 6 ];

```

18.7.4 Arrays and Records as Record Fields

Records may contain other records or arrays as fields. Consider the following definition:

```

type
    Pixel:
        record
            Pt:          point;
            color:      dword;
        endrecord;

```

The definition above defines a single point with a 32 bit color component. When initializing an object of type `Pixel`, the first initializer corresponds to the `Pt` field, *not the x-coordinate field*. **The following definition is incorrect:**

```

static
    ThisPt: Pixel := Pixel:[ 5, 10 ]; // Syntactically incorrect!

```

The value of the first field (“5”) is not an object of type *point*. Therefore, the assembler generates an error when encountering this statement. HLA will allow you to initialize the fields of *Pixel* using declarations like the following:


```

static
  ThisPt: Pixel := Pixel:[ point:[ 1, 2, 3 ], 10 ];
  ThatPt: Pixel := Pixel:[ point:[ 0, 0, 0 ], 5 ];

```

Accessing *Pixel* fields is very easy. Like a high level language you use a single period to reference the *Pt* field and a second period to access the *x*, *y*, and *z* fields of *point*:

```

  stdout.put( "ThisPt.Pt.x = ", ThisPt.Pt.x, nl );
  stdout.put( "ThisPt.Pt.y = ", ThisPt.Pt.y, nl );
  stdout.put( "ThisPt.Pt.z = ", ThisPt.Pt.z, nl );
  .
  .
  .
  mov( eax, ThisPt.Color );

```

You can also declare *arrays* as record fields. The following record creates a data type capable of representing an object with eight points (e.g., a cube):

```

type
  Object8:
    record
      Pts:          point [8];
      Color:       dword;
    endrecord;

```

There are two common ways to nest record definitions. As noted earlier in this section, you can create a record type in a TYPE section and then use that type name as the data type of some field within a record (e.g., the *Pt:point* field in the *Pixel* data type above). It is also possible to declare a record directly within another record without creating a separate data type for that record; the following example demonstrates this:

```

type
  NestedRecs:
    record
      iField: int32;
      sField: string;
      rField:
        record
          i:int32;
          u:uns32;
        endrecord;
      cField:char;
    endrecord;

```

Generally, it's a better idea to create a separate type rather than embed records directly in other records, but nesting them is perfectly legal and a reasonable thing to do on occasion.

18.7.5 Controlling Field Offsets Within a Record

By default, whenever you create a record, most assemblers automatically assign the offset zero to the first field of that record. This corresponds to records in a high level language and is the intuitive default condition. In some instances, however, you may want to assign a different starting offset to the first field of the record. The HLA assembler provides a mechanism that lets you set the starting offset of the first field in the record.

The syntax to set the first offset is

```

name:
  record := startingOffset;
        << Record Field Declarations >>
  endrecord;

```

Using the syntax above, the first field will have the starting offset specified by the *startingOffset* *int32* constant expression. Since this is an *int32* value, the starting offset value can be positive, zero, or negative.

One circumstance where this feature is invaluable is when you have a record whose base address is actually somewhere within the data structure. The classic example is an HLA string. An HLA string uses a record declaration similar to the following:

```

record
  MaxStrLen: dword;
  length: dword;
  charData: char[xxxx];
endrecord;

```

However, HLA string pointers do not contain the address of the *MaxStrLen* field; they point at the *charData* field. The *str.strRec* record type found in the HLA Standard Library Strings module uses a record declaration similar to the following:

```

type
  strRec:
    record := -8;
      MaxStrLen: dword;
      length: dword;
      charData: char;
    endrecord;

```

The starting offset for the *MaxStrLen* field is -8. Therefore, the offset for the *length* field is -4 (four bytes later) and the offset for the *charData* field is zero. Therefore, if EBX points at some string data, then “(type str.strRec [ebx]).length” is equivalent to “[ebx-4]” since the *length* field has an offset of -4.

18.7.6 Aligning Fields Within a Record

To achieve maximum performance in your programs, or to ensure that your records properly map to records or structures in some high level language, you will often need to be able to control the alignment of fields within a record. For example, you might want to ensure that a *dword* field’s offset is an even multiple of four. You use the *ALIGN* directive in a record declaration to do this. The following example shows how to align some fields on important boundaries:

```

type
  PaddedRecord:
    record
      c: char;
        align(4);
      d: dword;
      b: boolean;
        align(2);
      w: word;
    endrecord;

```

Whenever HLA encounters the *ALIGN* directive within a record declaration, it automatically adjusts the following field’s offset so that it is an even multiple of the value the *ALIGN* directive specifies. It accomplishes this by increasing the offset of that field, if necessary. In the example above, the fields would have the following offsets: *c*:0, *d*:4, *b*:8, *w*:10.

If you want to ensure that the record's size is a multiple of some value, then simply stick an `ALIGN` directive as the last item in the record declaration. HLA will emit an appropriate number of bytes of padding at the end of the record to fill it in to the appropriate size. The following example demonstrates how to ensure that the record's size is a multiple of four bytes:

```
type
  PaddedRec:
    record
      << some field declarations >>

      align(4);

    endrecord;
```

Be aware of the fact that the `ALIGN` directive in a `RECORD` only aligns fields in memory if the record object itself is aligned on an appropriate boundary. Therefore, you must ensure appropriate alignment of any record variable whose fields you're assuming are aligned.

If you want to ensure that all fields are appropriately aligned on some boundary within a record, but you don't want to have to manually insert `ALIGN` directives throughout the record, HLA provides a second alignment option to solve your problem. Consider the following syntax:

```
type
  alignedRecord3 :
    record[4]
      << Set of fields >>
    endrecord;
```

The "[4]" immediately following the `RECORD` reserved word tells HLA to start all fields in the record at offsets that are multiples of four, regardless of the object's size (and the size of the objects preceding the field). HLA allows any integer expression that produces a value in the range 1..4096 inside these parenthesis. If you specify the value one (which is the default), then all fields are packed (aligned on a byte boundary). For values greater than one, HLA will align each field of the record on the specified boundary. For arrays, HLA will align the field on a boundary that is a multiple of the array element's size. The maximum boundary HLA will round any field to is a multiple of 4096 bytes.

Note that if you set the record alignment using this syntactical form, any `ALIGN` directive you supply in the record may not produce the desired results. When HLA sees an `ALIGN` directive in a record that is using field alignment, HLA will first align the current offset to the value specified by `ALIGN` and then align the next field's offset to the global record align value.

Nested record declarations may specify a different alignment value than the enclosing record, e.g.,

```
type
  alignedRecord4 : record[4]
    a:byte;
    b:byte;
    c:record[8]
      d:byte;
      e:byte;
    endrecord;
    f:byte;
    g:byte;
  endrecord;
```

In this example, HLA aligns fields a, b, f, and g on dword boundaries, it aligns d and e (within c) on eight-byte boundaries. Note that the alignment of the fields in the nested record is true only within that nested record. That is, if c turns out to be aligned on some boundary other than an eight-

byte boundary, then *d* and *e* will not actually be on eight-byte boundaries; they will, however be on eight-byte boundaries relative to the start of *c*.

In addition to letting you specify a fixed alignment value, HLA also lets you specify a minimum and maximum alignment value for a record. The syntax for this is the following:

```
type
  recordname : record[maximum : minimum]
    << fields >>
endrecord;
```

Whenever you specify a maximum and minimum value as above, HLA will align all fields on a boundary that is at least the minimum alignment value. However, if the object's size is greater than the minimum value but less than or equal to the maximum value, then HLA will align that particular field on a boundary that is a multiple of the object's size. If the object's size is greater than the maximum size, then HLA will align the object on a boundary that is a multiple of the maximum size. As an example, consider the following record:

```
type
  r: record[ 4:1 ];
    a:byte;           // offset 0
    b:word;           // offset 2
    c:byte;           // offset 4
    d:dword[2];      // offset 8
    e:byte;           // offset 16
    f:byte;           // offset 17
    g:qword;          // offset 20
endrecord;
```

Note that HLA aligns *g* on a dword boundary (not qword, which would be offset 24) since the maximum alignment size is four. Note that since the minimum size is one, HLA allows the *f* field to be aligned on an odd boundary (since it's a byte).

If an array, record, or union field appears within a record, then HLA uses the size of an array element or the largest field of the record or union to determine the alignment size. That is, HLA will align the field without the outermost record on a boundary that is compatible with the size of the largest element of the nested array, union, or record.

HLA sophisticated record alignment facilities let you specify record field alignments that match that used by most major high level language compilers. This lets you easily access data types used in those HLLs without resorting to inserting lots of ALIGN directives inside the record.

18.7.7 Using Records/Structures in an Assembly Language Program

In the "good old days" assembly language programmers typically ignored records. Records and structures were treated as unwanted stepchildren from high-level languages, that weren't necessary in "real" assembly language programs. Manually counting offsets and hand-coding literal constant offsets from a base address was the way "real" programmers wrote code in early PC applications. Unfortunately for those "real programmers", the advent of sophisticated operating systems like Windows and Linux put an end to that nonsense. Today, it is very difficult to avoid using records in modern applications because too many API functions require their use. If you look at typical Windows and Linux include files for C or assembly language, you'll find hundreds of different structure declarations, many of which have dozens of different members. Attempting to keep track of all the field offsets in all of these structures is out of the question. Worse, between various releases of an operating system (e.g., Linux), some structures have been known to change, thus exacerbating the problem. Today, it's unreasonable to expect an assembly language programmer to manually track such offsets - most programmers have the reasonable expectation that the assembler will provide this facility for them.

18.7.8 Implementing Structures in an Assembler

Unfortunately, properly implementing structures in an assembler takes considerable effort. A large number of the "hobby" (i.e., non-commercial) assemblers were not designed from the start to support sophisticated features such as records/structures. The symbol table management routines in most assemblers use a "flat" layout, with all of the symbols appearing at the same level in the symbol table database. To properly support structures or records, you need a hierarchical structure in your symbol table database. The bad news is that it's quite difficult to retrofit a hierarchical structure over the top of a flat database (i.e., the symbol "hobby assembler" symbol table). Therefore, unless the assembler was originally designed to handle structures properly, the result is usually a major hacked-up kludge.

Four assemblers I'm aware of, MASM, TASM, OPTASM, and HLA, handle structures well. Most other assemblers are still trying to simulate structures using a flat symbol table database, with varying results.

Probably the first attempt people make at records, when their assembler doesn't support them properly, is to create a list of constant symbols that specify the offsets into the record. Returning to our first example (in HLA):

```
type
  student:
    record
      Name:      string;
      Major:     int16;
      SSN:       char[12];
      Midterm1:  int16;
      Midterm2:  int16;
      Final:     int16;
      Homework:  int16;
      Projects:  int16;
    endrecord;
```

One attempt might be the following:

```
const
  Name := 0;
  Major := 4;
  SSN := 6;
  Midterm1 := 18;
  Midterm2 := 20;
  Final := 22;
  Homework := 24;
  Projects := 26;
  size_student := 28;
```

With such a set of declarations, you could reserve space for a student "record" by reserving "size_student" bytes of storage (which almost all assemblers handle okay) and then you can access fields of the record by adding the constant offset to your base address, e.g.,

```
static
  John : byte[ size_student ];
  .
  .
  .
  mov( John[Midterm1], ax );
```

There are several problems with this approach. First of all, the field names are global and must be globally unique. That is, you cannot have two record types that have the same fieldname (as is possible with the assembler supports true records). The second problem, which is fundamentally more problematic, is the fact that you can attach these constant offsets to any object, not just a

"student record" type object. For example, suppose "ClassAverage" is an array of words, there is nothing stopping you from writing the following when using constant equate values to simulate record offsets:

```
mov( ClassAverage[ Midterm1 ], ax );
```

Finally, and probably the most damning criticism of this approach, is that it is very difficult to maintain code that accesses structures in this manner. Inserting fields into the middle of a record, changing data types, and coming up with globally unique names can create all sorts of problems. Many high-level language programmers who've tried to learn assembly language have given up after discovering that they had to maintain records in this fashion in an assembly language program (too bad they didn't start off with a reasonable assembler that properly supports structures).

Manually maintaining all the constant offsets is a maintenance nightmare. So somewhere along the way, some assembly language programmers figured out that they could write macros to handle the declaration of constant offsets for them. For example, here's how you could do this in an HLA program:

```
program t;

#macro struct( _structName_, _dcls_[] ):
    _dcl_, _id_, _type_, _colon_, _offset_;

    ?_offset_ := 0;
    ?_dcl_:string;
    #for( _dcl_ in _dcls_ )

        ?_colon_ := @index( _dcl_ , 0, ":" );
        #if( _colon_ = -1 )

            #error
            (
                "Expected <id>:<type> in struct definition, encountered: ",
                _dcl_
            )

        #else

            ?_id_ := @substr( _dcl_, 0, _colon_ );
            ?_type_ := @substr( _dcl_, _colon_+1, @length( _dcl_ ) -
                _colon_ );
            ?@text( _id_ ) := _offset_;
            ?_offset_ := _offset_ + @size( @text( _type_ ) );

        #endif;

    #endifor
    ?_structName_:text := "byte[" + @string( _offset_ ) + "];

#endmacro

struct( threeItems, i:byte, j:word, k:dword )

static
    aStruct: threeItems;
```

```
begin t;  
  
    mov( (type byte aStruct[i]), al );  
    mov( (type word aStruct[j]), ax );  
    mov( (type dword aStruct[k]), eax );  
  
end t;
```

The "struct" macro expects a set of valid HLA variable declarations supplied as macro arguments. It generates a set of constants using the supplied variable names whose offsets are adjusted according to the size of the objects previously appearing in the list. In this example, HLA creates the following equates:

```
i = 0  
j = 1  
k = 3
```

This declaration also creates a "data type" named "threeItems" which is equivalent to "byte[7]" (since there are seven bytes in this record) that you may use to create variables of type "threeItems", as is done in this example.

Creating structures with macros solves one of the three major problems: it makes it easier to maintain the constant equates list, as you do not have to manually adjust all the constants when inserting and removing fields in a record. This does not, however, solve the other problems (particularly, the global identifier problem).

While fancier macros could be written, macros that generate identifiers like "objectname_fieldName" that help solve the globally unique problem, the bottom line is that these hacks begin to fail when you attempt to declare nested records, arrays within records, and arrays of records (possibly containing nested records and arrays of records). The bottom line is this: assemblers that don't properly support structures are going to have problems when you've got to work with data structures from high-level languages (e.g., OS API calls, where the OS is written in C, such as Windows and Linux). You're much better off using an assembler that fully supports structures (and other advanced data types) if you need to use structures in your programs.

Index

- (negation) operator in constant expressions 127
- operator (subtraction, set difference) in constant expressions 130
- operator in constant expressions 30, 131
- or != operator 330
- Symbols
 - 31, 130, 131, 330
 - ^ operator in constant expressions 31, 131
 - ! (not operator) in constant expressions 30, 126
 - !(boolean_expression) operator 329, 332
 - != operator in constant expressions 131
 - !memory operator 329
 - !register operator 329
 - ? command-line option 87
 - @ command-line option 86
 - @@ command-line option 86
 - @a 331
 - @abs function 261
 - @addofs1st function 277
 - @ae 331
 - @align 186
 - @Align procedure option 172
 - @alignstack 186
 - @alignstack procedure option 172, 184, 185
 - @arity function 275
 - @b 331
 - @basetype function 273
 - @basetype function 272
 - @be 331
 - @bound function 278
 - @bound pseudo-variable 417
 - @byte compile-time function 261
 - @byte function 262
 - @c 331
 - @Cdecl procedure option 346
 - @cdecl procedure option 172, 173
 - @ceil function 262
 - @char compile-time function 261
 - @class function 274
 - @cos function 262
 - @cset compile-time function 261
 - @curdir function 277
 - @curlex function 277
 - @curobject function 277
 - @curoffset function 277
 - @date function 262
 - @defined function 275
 - @delete function 265
 - @dim function 275
 - @display 185
 - @display procedure option 172, 184, 185
 - @dword compile-time function 261
 - @e 331
 - @elements function 275
 - @elementsize function 274
 - @enter 186
 - @enter procedure option 174
 - @enumsize function 278
 - @env function 262
 - @EOS function 271
 - @errorprefix pseudo variable 277
 - @eval function 258
 - @exactlynChar function 268
 - @exactlynCset function 267
 - @exactlyniChar function 269
 - @exactlyntomChar function 269
 - @exactlyntomCset function 267
 - @exactlyntomiChar function 269
 - @exceptions function 278
 - @exp function 262
 - @External option (in variable declarations) 33
 - @External procedures 183
 - @extract function 262
 - @firstnChar function 268
 - @firstnCset function 267
 - @firstniChar function 269
 - @floor function 263
 - @FORWARD declarations 184
 - @frame 186
 - @g 331
 - @ge 331

- @index function 265
- @insert function 265
- @int8/@int16/@int32/@int64/@int128
compile-time functions 261
- @into function 278
- @isalpha function 263
- @isalphanum function 263
- @isclass function 276
- @isconst function 276
- @isdigit function 263
- @IsExternal function 275
- @isfreq function 276
- @islower function 263
- @ismem function 276
- @isreg function 276
- @isreg16 function 276
- @isreg32 function 276
- @isreg8 function 276
- @isspace function 263
- @istype function 276
- @isupper function 263
- @isxdigit function 263
- @l 331
- @lastobject function 277
- @le 331
- @leave 186
- @leave procedure option 174
- @length function 265
- @lex function 274
- @linenumber function 259, 276, 277
- @localoffset function 277, 278
- @locals function 275
- @log function 263
- @log10 function 263
- @lowercase function 265
- @lword compile-time function 261
- @match compile-time function 282,
296
- @match2 compile-time function 296
- @matchChar compile-time function
285
- @matchID function 270
- @matchIntConst function 270
- @matchiStr function 269
- @matchNumericConst function 270
- @matchRealConst function 270
- @matchStr function 269
- @matchStrConst function 271
- @matchToiStr function 270
- @matchToStr function 270
- @max function 263
- @min function 263
- @minparmsize function 278
- @na 331
- @nae 331
- @name function 272
- @nb 331
- @nbe 331
- @nc 331
- @ne 331
- @nge 331
- @nl 331
- @nle 331
- @no 331
- @noalignstack 186
- @noalignstack procedure option 172
- @nodisplay 185
- @Nodisplay option 221
- @nodisplay procedure option 172
- @noenter 186
- @noenter procedure option 174, 184,
185
- @noframe 186
- @noframe procedure option 172
- @noleave 186
- @noleave procedure option 174, 184,
185
- @nOrLessChar function 268
- @nOrLessCset function 267
- @nOrLessiChar function 269
- @nOrMoreChar function 269
- @nOrMoreCset function 267
- @nOrMoreiChar function 269
- @NOSTORAGE 33
- @ns 331
- @ntomChar function 269
- @ntomCset function 267
- @ntomiChar function 269
- @nz 331
- @o, 331
- @odd function 263
- @offset function 274

- @oneChar compile-time function 285
- @oneChar function 268
- @oneCset function 266
- @oneiChar function 269
- @oneOrMoreChar function 268
- @oneOrMoreCset function 267
- @oneOrMoreiChar function 269
- @oneOrMoreWS function 271
- @optstring function 279
- @parmoffset function 277
- @Pascal procedure option 346
- @pascal procedure option 172, 173
- @pclass function 275
- @peekChar function 268
- @peekCset function 266
- @peekiChar function 269
- @peekWS function 271
- @ptype function 272
- @qword compile-time function 261
- @random function 263
- @randomize function 264
- @read compile-time function 302
- @real32/@real64/@real80 compile-time functions 261
- @REG function 271, 272
- @REG32 function 272
- @REG8 function 271
- @Returns procedure option 172, 346
- @returns procedure option 173
- @rindex function 265
- @s 331
- @section function 280
- @sin function 264
- @size function 274
- @sort function 264
- @sqrt function 264
- @staticname function 274
- @Stdcall procedure option 346
- @stdcall procedure option 172, 173
- @strbrk function 265
- @string
 - operator 280
- @string compile-time function 261
- @strset function 265
- @strspan function 265
- @substr function 265
- @tan function 264
- @text function 280
- @text operator 250
- @time function 264
- @tokenize function 265
- @tostring
 - operator 280
- @tostring operator 257
- @trace function 279
- @trim function 266
- @type function 272
- @typename function 272
- @uns8/@uns16/@uns32/@uns64/
 - @uns128 compile-time functions 261
- @uppercase function 266
- @uptoChar function 268
- @uptoCset function 266
- @uptoiChar function 269
- @uptoiStr function 270
- @uptoStr function 270
- @use procedure option 173
- @use reg32 procedure option 172
- @word compile-time function 261
- @WSorEOS function 271
- @WSthenEOS function 271
- @z 331
- @zeroOrMoreChar function 268
- @zeroOrMoreCset function 267
- @zeroOrMoreiChar function 269
- @zeroOrMoreWS function 271
- @zeroOrOneChar function 268
- @zeroOrOneCset function 266
- @zeroOrOneiChar function 269
- * (multiplication) operator in constant expressions 128
- *NIX 54
- / (division) operator in constant expressions 129
- & operator in constant expressions 31, 131
- && operator 329, 332
- #(...)# in macro parameters 250
- #{ ... }# sequence for manually passing parameters 215
- #{...}# parameter quoting mechanism

- 198
- # {...}# sequence (to create thunks) 213
 - # {...}#" code brackets in boolean expressions 333
 - #append 301
 - #asm..#endasm directives 299
 - #closeread compile-time statement 302
 - #closewrite statement 301
 - #else clause 36, 303
 - #elseif clause in #if statement 36, 303
 - #emit directive 299
 - #endif clause 36, 303
 - #endwhile 150
 - #ERROR directive 301
 - #for..#endfor statement 303
 - #if statement 35, 302
 - #Include directive 35, 245, 246
 - #include directive 347
 - #IncludeOnce directive 247
 - #KEYWORD reserved word 252
 - #linker directive 246
 - #match..#endmatch 298
 - #openread compile-time statement 302
 - #openwrite statement 301
 - #PRINT directive 301
 - #regex..#endregex statement 283
 - #return clause 283
 - #system directive 300
 - #TERMINATOR keyword 252
 - #text..#endtext statement 281
 - #while 150
 - #while..#endwhile statement 303
 - #write statement 301
 - + (addition, set union, string concatenation) operator in constant expressions 130
 - = operator in constant expressions 30, 131
 - = or == operator 330
 - == operator in constant expressions 131
 - > operator 330
 - > operator in constant expressions 31, 131
 - >= operator 330
 - >= operator in constant expressions 31, 131
 - >> operator (shift right) in constant expressions 130
 - | operator in constant expressions 31, 131
 - || operator 329, 332
- Numerics
- 80x86 instruction set 36, 397
- A
- aaa instruction 398
 - aad instruction 398
 - aam instruction 398
 - aas instruction 398
 - ABSTRACT keyword 232
 - Abstract methods and abstract base classes 232
 - Accessing fields of a structure 494
 - Accessing the fields of a class 478
 - Accessor methods 469
 - Activation record 112
 - adc instruction 36, 402
 - add instruction 36, 402
 - Addition operator (+) in constant expressions 130
 - Address of a class procedure 239
 - Address of a memory object (calculation) 410
 - Address of a method 239
 - Address of an iterator 239
 - Address-of operator 239
 - align directive 154
 - align directive (in records) 108
 - align procedure option 173
 - Aligning fields within a record 497
 - Allocating objects dynamically 488
 - always exception handling clause 326 and instruction 36, 402
 - AND operator in boolean expressions 332
 - Anonymous records 106
 - Anonymous unions 107
 - ANYEXCEPTION clause in the TRY..ENDTRY statement 325
 - Arithmetic and logical instructions 36, 402

- Arithmetic instructions 38, 408
 - Array constants 118
 - Array data types 28, 105
 - Arrays as structure fields 495
 - Arrays of records 495
 - Art of Assembly Language Programming 5, 7
 - Assembler control compile-time functions 272
 - Automatic code generation in procedures 186
- B**
- b**
 - name command-line option 86
 - Back-end assembler command line parameters 86
 - Back-end assemblers 59
 - Backtracking 294
 - Base classes 484
 - begin..end block 337
 - Big endian data format 42, 419
 - Binary constants 29, 116, 151, 152, 153
 - Binary object file output name (command-line option) 86
 - Bit scan instructions 42, 419
 - Bit test instructions 42, 419
 - Bitwise type transfer functions 261
 - Block-structured language 218
 - Boolean constants 29, 117
 - Boolean data type 102
 - Boolean expressions for high-level language statements 329
 - bound instruction 41, 416
 - Branch out of range errors 415
 - break and breakif statements 336
 - bsf instruction 419
 - bsr instruction 42, 419
 - bswap instruction 42, 419
 - bt instruction 42, 419
 - btc instruction 42, 419
 - btr instruction 42, 419
 - bts instruction 42, 419
 - Built-in types 151
 - Byte data type 102
- C**
- c command-line option 86
 - Call instruction 192
 - call instruction 412
 - Calling a class procedure 412
 - Calling a procedure 39, 412
 - Calling conventions 348
 - Calling HLA Procedures 192
 - Calling HLA procedures from another language 349
 - Calling methods and class procedures 234
 - Calling procedures written in a different language 349
 - Cascading exceptions 327
 - case clause in switch statement 339
 - Case neutrality 27, 100, 347
 - Case-insensitive character matching (compile-time function) 286
 - cbw instruction 398
 - cd 50
 - CDecl procedure option 348
 - cdq instruction 398
 - Changing the Location of HLA 58
 - char compile-time function 260
 - Character constants 29, 117
 - Character data type 102
 - Character set constants 118
 - Character set data type 102
 - Character set union operator (+) in constant expressions 130
 - Class data types 222
 - Class Methods, Iterators, and Procedures 476
 - Class procedure address 239
 - Class procedures 235
 - Classes 469
 - clc instruction 398
 - cld instruction 398
 - cli instruction 398
 - clts instruction 398
 - cmc instruction 398
 - cmova instruction 40, 415
 - cmovae instruction 40, 415
 - cmovb instruction 40, 415
 - cmovbe instruction 40, 415

- cmovc instruction 40, 415
- cmove instruction 40, 415
- cmovg instruction 40, 415
- cmovge instruction 40, 415
- cmovl instruction 40, 415
- cmovle instruction 40, 415
- cmovna instruction 40, 415
- cmovnae instruction 40, 415
- cmovnb instruction 40, 415
- cmovnbe instruction 40, 415
- cmovnc instruction 40, 415
- cmovne instruction 40, 415
- cmovng instruction 40, 415
- cmovnge instruction 40, 415
- cmovnl instruction 40, 415
- cmovnle instruction 40, 415
- cmovno instruction 40, 415
- cmovnp instruction 40, 415
- cmovns instruction 40, 415
- cmovnz instruction 40, 415
- cmovo instruction 40, 415
- cmovp instruction 40, 415
- cmovpe instruction 40, 415
- cmovpo instruction 40, 415
- cmovs instruction 40, 415
- cmovz instruction 40, 415
- cmp instruction 37, 404
- cmpsb instruction 398
- cmpsd instruction 398
- cmpsw instruction 398
- cmpxchg instruction 41, 417, 418
- Command line parameters 86
- Comments 26, 93
- Comparison operators in constant expressions 131
- Compile only (command-line option) 86
- Compile-time "?" statement
 - ? compile-time statement 146
- Compile-time language 245
- Compiling simple programs with HIDE 4
- Composing instructions 397
- Composite data types 28, 105
- Computing the address of a memory operand 410
- Conditional compilation statements 35, 302
- Conditional jump instructions 40, 415
- Conditional move instructions 40, 415
- Conditional set instructions 40, 415
- Conjunction in boolean expressions 332
- const declaration section 142
- const keyword 142
- Const sections 33
- Constant expressions 30, 124
- Constants
 - array 118
 - binary 29, 116
 - boolean 29, 117
 - character 29, 117
 - character set 118
 - decimal 29, 115
 - _display_ 182
 - floating-point 29, 116
 - hexadecimal 29, 115
 - literal 29, 115
 - numeric 29, 115
 - _parms_ 182
 - pointer 30, 123
 - record 119
 - regular expression 123
 - string 30, 117
 - structured 30, 118
 - Unicode character 117
 - Unicode string 117
 - union 120
- Constructors 233, 487, 489
- Context free macros 252
- continue and continueif statements 336
- Controlling field offsets within a record 496
- Conversion functions (compile-time language) 260
- cpuid instruction 398
- Create procedure for an object 489
- Creating a New Project in RadASM 41
- Creating a Virtual Method Table 234
- cset compile-time function 260
- Cset data type 102, 152
- Customizing HLA 57
- cwd instruction 398
- cwde instruction 398

D

daa instruction 398
 das instruction 398
 Data Types 27, 102
 Data types
 array 28, 105
 composite 28
 enumerated 103
 pointer 111
 record 28, 106
 thunks 112
 union 105
 dec instruction 38, 408
 Decimal constants 29, 115
 Declaration section in an HLA program 135
 Declarations 32
 array 105
 const 142
 constants 33
 enum 103
 external procedure 183
 forward procedure 184
 label 135
 macros 34
 namespace 167
 overloaded procedure, iterator, and method 177
 proc 167
 procedure 31, 171, 175
 readonly 165
 record 106
 static 33, 160
 storage 164
 type 32, 150
 union 105
 var 153
 default clause in switch statement 339
 Default include file directory 86
 Deferred macro parameter expansion 258
 Defining symbols on the HLA command line 86
 dir 50
 Disjunction in boolean expressions 332
 Display (accessing non-local variables)

220

_display_array constant 182
 _display_variable 183, 220
 Displaying messages during compilation 301
 div instruction 38, 406
 Div operator in constant expressions 129
 Divide instructions 38, 406
 Division operator in constant expressions 129
 Dot operator 494
 Dot operator (field name selection) 132
 Double precision shift instructions 38, 409
 DUP operator (array constants) 119
 Dword data type 102
 dxx command-line option 86
 dxx=yy command-line option 86
 Dynamic Object Allocation 488

E

e
 name command-line option 86
 Eager evaluation of macro parameters 258
 Editing HLA source files within RadASM 55
 Effective address calculations 410
 ELF code generation for FreeBSD 87
 ELF code generation for Linux OS 87
 ELSE 328
 ELSEIF statement 328
 emms instruction 425
 endconst keyword 142
 ENDIF 328
 endlabel keyword 136
 endswitch clause in switch statement 339
 endval keyword 147
 endvar keyword 154
 enter instruction 41, 417
 enum 103
 Enumerated data types 102, 103
 Errors 301
 Exception handling in HLA 321

- Exception numbers 323
 - Executable output filename (command-line parameter) 86
 - exit and exitif statements 337, 338
 - EXIT statement 191
 - External compilation units 345
 - External declarations 345
 - External identifiers 27, 100, 101
 - External procedure declarations 183
 - External symbol names (non-HLA identifiers) 346
 - External symbols 27, 100
- F
- f2xm1 instruction 422
 - fabs instruction 422
 - fadd instruction 420
 - faddp instruction 420
 - FASM as back-end assembler (command-line option) 86
 - fasm command-line option 86
 - fbld instruction 420
 - fbstp instruction 420
 - fchs instruction 422
 - fclex instruction 423
 - FCMOVcc instructions 423
 - fcom instruction 422
 - fcomi instruction 423
 - fcomip instruction 423
 - fcomp instruction 422
 - fcompp instruction 422
 - fcos instruction 422
 - fdecstp instruction 423
 - fdiv instruction 421
 - fdivp instruction 421
 - fdivr instruction 421
 - fdivrp instruction 421
 - ffree instruction 423
 - fiadd instruction 421
 - ficom instruction 422
 - ficompp instruction 422
 - fidiv instruction 421
 - fidivr instruction 422
 - Field alignment within a record 497
 - Field Offsets Within a Record 496
 - fild instruction 420
 - fimul instruction 421
 - _finalize_string 240
 - Finalizers 240
 - finestp instruction 423
 - finit instruction 422
 - fist instruction 420
 - fistp instruction 420
 - fld instruction 420
 - fldl instruction 422
 - fldcw instruction 423
 - fldl2e instruction 422
 - fldl2t instruction 422
 - fldlg2 instruction 422
 - fldln2 instruction 422
 - fldpi instruction 422
 - fldz instruction 422
 - Floating point constants 116
 - Floating point instructions 42, 420
 - Floating-point constants 29
 - fmul instruction 420
 - fmulp instruction 421
 - fnop instruction 423
 - for..endfor statement 334
 - foreach..endfor statement 342
 - forever..endfor statement 336
 - FORWARD declarations 241
 - Forward procedure declarations 184
 - fpatan instruction 422
 - fprem instruction 422
 - fprem1 instruction 422
 - fptan instruction 422
 - frame procedure option 172, 184, 185
 - freebsd command-line option 87
 - FreeBSD OS ELF code generation 87
 - frndint instruction 422
 - fscale instruction 422
 - fsin instruction 422
 - fsincos instruction 422
 - fsqrt instruction 422
 - fst instruction 420
 - fstcw instruction 423
 - fstp instruction 420
 - fstsw instruction 423
 - fsub instruction 421
 - fsubp instruction 421
 - fsubr instruction 421

- fsubrp instruction 421
 - fst instruction 422
 - fucom instruction 422
 - fucomp instruction 422
 - fucompp instruction 422
 - Function overloading 348
 - fwait instruction 423
 - fxam instruction 422
 - fxch instruction 420
 - fxtract instruction 422
 - fyl2x instruction 422
 - fyl2xp1 instruction 422
- G**
- Gas 1
 - Gas as back-end assembler (command-line option) 86
 - gas command-line option 86
 - gasx command-line option 86
 - Generating a linker response file 86
 - greedy evaluation 295
- H**
- Header files 347
 - Hello World 49
 - Hexadecimal constants 29, 115
 - HIDE 1
 - auto completion 17
 - compiling simple programs 4
 - global settings 11
 - menus 4
 - project file format 18
 - project manager 14
 - project panel 3
 - settings 10
 - High level language statements 321
 - HLA command line options 86
 - hla command-line option 87
 - HLA Compile-Time Language 245
 - HLA customization 57
 - HLA Design Goals 5
 - HLA environment variables 46
 - HLA installation under Windows 45
 - HLA Integrated Development Environment 1
 - HLA internal operation 84
 - HLA language elements 93
 - HLA program format 31
 - HLA source file output (command-line option) 87
 - HLA Standard Library 45, 54
 - HLA type compatibility 104
 - HLA.INI initialization file 77
 - hlabe command-line option 87
 - HLAINC environment variable 48
 - HLALIB environment variable 48
 - hlt instruction 398
 - Human readable source file format (command-line option) 86
 - Hybrid high level boolean expressions 332
 - Hybrid parameter passing in HLA 215, 216
- I**
- i
 - path command-line option 86
 - IDE 24
 - Identifiers 27, 100
 - idiv instruction 38, 406
 - IF statement 328
 - imod instruction 406
 - imul instruction 37, 405
 - in instruction 41, 416
 - IN operator 331
 - in operator 329, 331
 - in operator in constant expressions 131
 - IN reg parameter specification 171
 - inc instruction 38, 408
 - Include file directory 86
 - Include files 35, 246
 - Index operator (selecting an array element) 133
 - Infinite loops 336
 - Inheritance 473, 484
 - Inherited fields in records 107
 - Inheritance 228
 - inherits keyword 107
 - INHERITS keyword (classes) 473
 - INHERITS reserved word 228
 - _initialize_string 240
 - Initializers 240

- Input and output instructions 41, 416
 - insb instruction 398
 - insd instruction 399
 - Installation under Windows 45
 - Installing HLA 45
 - Installing HLA under FreeBSD 54
 - Installing HLA under Linux 54
 - Installing HLA under Mac OSX 54
 - Installing RadASM 31
 - Instances (of a class) 471
 - Instruction composition 397
 - insw instruction 399
 - int instruction 41, 416
 - Int128 data type 102
 - Int16 data type 102
 - Int32 data type 102
 - Int64 data type 102
 - Int8 data type 102
 - int8/int16/int32/int64/int128 compile-time functions 260
 - Integrated Development Environments 24
 - Internal name of the HLA main program 86
 - intmul instruction 37, 405
 - into instruction 399
 - invd instruction 399
 - iret instruction 399
 - iretd instruction 399
 - Iterator address 239
 - Iterator declarations (overloaded) 177
 - Iterators 342
- J**
- ja instruction 40, 415
 - jae instruction 40, 415
 - jb instruction 40, 415
 - jbe instruction 40, 415
 - jc instruction 40, 415
 - jcxz instruction 40, 415
 - je instruction 40, 415
 - jecxz instruction 40, 415
 - jg instruction 40, 415
 - jge instruction 40, 415
 - jl instruction 40, 415
 - jle instruction 40, 415
 - jmp instruction 40, 414
 - jna instruction 40, 415
 - jnae instruction 40, 415
 - jnb instruction 40, 415
 - jnbe instruction 40, 415
 - jnc instruction 40, 415
 - jne instruction 40, 415
 - jng instruction 40, 415
 - jnge instruction 40, 415
 - jnl instruction 40, 415
 - jnle instruction 40, 415
 - jno instruction 40, 415
 - jnp instruction 40, 415
 - jns instruction 40, 415
 - jnz instruction 40, 415
 - jo instruction 40, 415
 - jp instruction 40, 415
 - jpe instruction 40, 415
 - jpo instruction 40, 415
 - js instruction 40, 415
 - JT and JF medium level statements 341
 - jz instruction 40, 415
- K**
- Kleene Plus 284
 - Kleene Star 284
- L**
- label declaration section 135
 - label keyword 135
 - lahf instruction 399
 - Lazy (pass by lazy evaluation) parameter option 171
 - Lazy bersus greedy evaluation 295
 - Lazy evaluation parameters 213
 - lea instruction 39, 410
 - leave instruction 399, 417
 - level=h command-line option 87
 - level=l command-line option 87
 - level=m command-line option 87
 - level=v command-line option 87
 - Lex level 219
 - Lexical analysis 309
 - Lexical Scope 218
 - lib
 - path command-line option 86

- license command-line option 86
 - Linker command-line parameters 86
 - Linker response file 86
 - Linking HLA code with other languages 347
 - linux command-line option 87
 - Linux OS ELF code generation 87
 - Literal constants 29, 115
 - Literal record constants 494
 - Little endian data format 42, 419
 - Local symbols in macros 34, 249
 - Local symbols in multi-part macros 255
 - lods instruction 399
 - lods instruction 399
 - lodsw instruction 399
 - Logical AND operator in constant expressions 131
 - Logical instructions 36, 38, 402, 408
 - Logical OR operator in constant expressions 131
 - Logical XOR operation in constant expressions 131
 - Lookahead 292
 - loop instruction 40, 415
 - loope instruction 40, 415
 - loopn instruction 40, 415
 - loopz instruction 40, 415
 - LWord data type 102
 - lxxxxx command-line option 86
- M**
- m command-line option 86
 - Mac OSX/Mach-o code generation 87
 - Mach-o code generation for Mac OS X 87
 - macos command-line option 87
 - Macro invocations 256
 - Macro parameters 256, 257
 - Macros 34, 248
 - Macros as compile-time functions 305
 - main
 - name command-line option 86
 - Make files in RadASM 25
 - Make menu in RadASM 79
 - Mangled names 347
 - Map files 86
- MASM** 1
- MASM as back-end assembler (command-line option) 87
 - masm command-line option 87
 - Memory addressing modes 42, 390
 - Method address 239
 - Method declarations (overloaded) 177
 - Methods 235, 469
 - mkdir 50
 - MMX instructions 42, 423
 - mod instruction 406
 - MOD operator (remainder) in constant expressions 129
 - mov instruction 36, 402
 - movd instruction 424
 - movq instruction 425
 - movsb instruction 399
 - movsd instruction 399
 - movsw instruction 399
 - movsx instruction 39, 411
 - movzx instruction 39, 411
 - mul instruction 37, 404
 - Multidimensional arrays 28, 105
 - Multi-part macros 252
 - Multiplication operator (*) in constant expressions 128
 - Multiply instructions 37, 404
- N**
- Name (pass by name) parameter option 171
 - Name mangling 347
 - Name parameters 213
 - namespace declaration section 167
 - Naming conventions 347
 - NASM 1
 - NASM as back-end assembler (command-line option) 87
 - nasm command-line option 87
 - neg instruction 38, 408
 - Negated String Matching 287
 - Negation operator in constant expressions 127
 - Nesting record definitions 496
 - New style procedure declarations 175
 - Non-object calls of class procedures 236

- nop instruction 399
 - not in operator 329, 331
 - not instruction 38, 408
 - Not operator (constant expressions) 30, 126
 - Null operand instructions 36
 - Number data types 103
 - Numeric constants 29, 115
 - Numeric data types 103
 - Numeric Set Constants 116
- O**
- obj
 - path command-line option 86
 - Object file placement during compilation (command-line option) 86
 - Object Initialization 487
 - Object-oriented programming 222
 - Objects 471
 - operator in constant expressions 133
 - OR operator in boolean expressions 332
 - Ordinal data types 103
 - out instruction 41, 416
 - outsb instruction 399
 - outsd instruction 399
 - outsw instruction 399
 - overload macro 177
 - Overloaded procedure declarations 177
 - overloads keyword 177
 - OVERRIDE keyword 229
 - overrides keyword 108
 - Overriding a method 473
 - Overriding precedence in constant expressions 132
- P**
- p
 - path command-line option 86
 - packssdw instruction 424
 - packsswbv 424
 - packuswb instruction 424
 - paddb instruction 423
 - padd instruction 423
 - Padding a record to some number of bytes 498
 - paddsb instruction 423
 - paddsw instruction 423
 - paddusb instruction 423
 - paddusw instruction 423
 - paddw instruction 423
 - pand instruction 424
 - pandn instruction 424
 - Parameters 191
 - Parameters passed on the stack 348
 - Parenthesis in macro parameters 35, 250
 - `_parms_` constant 182
 - Pass by lazy evaluation 213
 - Pass by name parameters 213
 - Pass by reference 203
 - Pass by result parameters 203
 - Pass by value 193
 - Pass by value/result 203
 - Passing by value
 - Byte-Sized Parameters 194
 - Passing byte value
 - Double-word-sized parameters 200
 - Large parameters 202
 - Lword-sized parameters 201
 - Quad-word-sized parameters 200
 - Tbyte-sized parameters 201
 - Word-sized parameters 198
 - Passing parameters in registers 171, 216
 - Path specifications in RadASM 77
 - pavgb instruction 424
 - pavgw instruction 424
 - pcmpeqb instruction 424
 - pcmpeqd instruction 424
 - pcmpeqw instruction 424
 - pcmpgtb instruction 424
 - pcmpgtd instruction 424
 - pcmpgtw instruction 424
 - PE/COFF object code generation 87
 - pextrw instruction 424
 - pinsrw instruction 424
 - PL/360 1
 - PL/M 1
 - pmaddwd instruction 424
 - pmaxsw instruction 424
 - pmaxub instruction 424
 - pminsw instruction 424

- pminub instruction 424
 - pmovmskb instruction 424
 - pmulhuw instruction 424
 - pmulhw instruction 424
 - pmullw instruction 424
 - Pointer constants 30, 123
 - Pointer types 111
 - polymorphism 475
 - pop instruction 39, 411
 - popa instruction 399
 - popad instruction 399
 - popf instruction 399
 - popfd instruction 399
 - por instruction 424
 - Precompiling regular expressions 297
 - Primitive data types 27, 102
 - Private fields in a class 100
 - Private fields in a class 471
 - proc declaration section 167
 - Procedure calls 39, 191, 412
 - Procedure declarations 31, 171
 - procedure declarations 175
 - Procedure declarations (overloaded) 177
 - Procedure options 185
 - Program Structure 31
 - Program structure 134
 - Project organization in RadASM 24
 - Project Panel in HIDE 3
 - Project types in RadASM 78
 - Prototype software 7
 - psadbw instruction 424
 - Pseudo-variables 277
 - pshufw instruction 424
 - pslld instruction 425
 - psllq instruction 425
 - psllw instruction 425
 - psrad instruction 425
 - psraw instruction 425
 - psrld instruction 425
 - psrlq instruction 425
 - psrlw instruction 425
 - psubb instruction 423
 - psubd instruction 424
 - psubsb instruction 424
 - psubsw instruction 424
 - psubusb instruction 424
 - psubusw instruction 424
 - psubw instruction 423
 - punpckhbw instruction 424
 - punpckhdq instruction 424
 - punpckhwd instruction 424
 - punpcklbw instruction 424
 - punpckldq instruction 424
 - punpcklwd instruction 424
 - push and pop instructions 39, 411
 - pusha instruction 399
 - pushad instruction 399
 - pushd instruction 39, 411
 - pushf instruction 399
 - pushfd instruction 399
 - pushw instruction 39, 411
 - _pVMT_ 233, 488
 - pxor instruction 424
- Q**
- QWord data type 102
- R**
- r
 - name command-line option 86
 - RadASM execution 31
 - RadASM installation 31
 - RadASM project management 32
 - RADASM.INI file 74
 - RadASM/HLA Integrated Development Environment 24
 - RAISE statement 321
 - raise statement 327
 - Range checking 41, 416
 - rcl instruction 38, 409
 - rcr instruction 409
 - rdmsr instruction 400
 - rdpmc instruction 400
 - rdtsc instruction 400
 - readonly declaration section 165
 - Real (Floating Point) Constants 29
 - Real constants 116
 - Real128 102
 - Real32 data type 102
 - real32/real64/real80 compile-time functions 260

Real64 data type 102
 Real80 data type 102
 Record constants 119, 494
 Record data types 28, 106
 Record field alignment 497
 Record offsets 111, 496
 Records as record fields 495
 Recursive file inclusion (prevention)
 247
 Reference parameters 203
 Register parameters 216
 Regular expression constants 123
 Regular expression macros 281
 rep.insb instruction 400
 rep.insd instruction 400
 rep.insw instruction 400
 rep.movsb instruction 400
 rep.movsd instruction 400
 rep.movsw instruction 400
 rep.outsb instruction 400
 rep.outsd instruction 400
 rep.outsw instruction 400
 rep.stosb instruction 400
 rep.stosd instruction 400
 rep.stosw instruction 400
 repe.cmpsb instruction 400
 repe.cmpsd instruction 400
 repe.cmpsw instruction 400
 repe.scasb instruction 401
 repe.scasd instruction 401
 repe.scasw instruction 401
 repeat..until statement 334
 repne.cmpsb instruction 401
 repne.cmpsd instruction 401
 repne.cmpsw instruction 401
 repne.scasd instruction 401
 repne.scasw instruction 401
 Reraising an exception 327
 Reserved words 27, 93
 Result (pass by result) parameter option
 171
 Result parameters 203
 ret instruction 40, 414
 RET with NOFRAME option 189
 RETURNS statement 397
 returns statement 217

rol instruction 38, 409
 ror instruction 409
 Rotate instructions 38, 409
 rsm instruction 401
 Running HLA 49
 Running RadASM 31

S

s command-line option 86
 sahf instruction 401
 sal instruction 38, 409
 sar instruction 409
 sbb instruction 36, 402
 scasb instruction 401
 scasd instruction 401
 scasw instruction 401
 Scoping rules 218
 Selecting field names 132
 Set intersection operator (*) in constant
 expressions 128
 Set union operator (+) in constant expres-
 sions 130
 seta instruction 40, 415
 setae instruction 40, 415
 setb instruction 40, 415
 setbe instruction 40, 415
 setc instruction 40, 415
 sete instruction 40, 415
 setg instruction 40, 415
 setge instruction 40, 415
 setl instruction 40, 415
 setle instruction 40, 415
 setna instruction 40, 415
 setnae instruction 40, 415
 setnb instruction 40, 415
 setnbe instruction 40, 415
 setnc instruction 40, 415
 setne instruction 40, 415
 setng instruction 40, 415
 setnge instruction 40, 415
 setnl instruction 40, 415
 setnle instruction 40, 415
 setno instruction 40, 415
 setnp instruction 40, 415
 setns instruction 40, 415
 setnz instruction 40, 415

- seto instruction 40, 415
 - setp instruction 40, 415
 - setpe instruction 40, 415
 - setpo instruction 40, 415
 - sets instruction 40, 415
 - Setting auxiliary paths for HLA files 59
 - Setting default procedure options 185
 - setz instruction 40, 415
 - Sevag Krikorian 1
 - Shift and rotate instructions 38, 409
 - Shift left operator (<) 130
 - Shift right operator (>>) in constant expressions 130
 - shl instruction 38, 409
 - shld instruction 409
 - shr instruction 409
 - shrd instruction 409
 - Sign and zero extension instructions 39, 411
 - Signed data types 103
 - Signed vs. unsigned comparisons in boolean expressions 330
 - source command-line option 86
 - Source file format output from HLA 86
 - Special symbols and punctuation 26, 93
 - Specifying an external symbols name 346
 - Static class fields 237
 - Static data objects in a class 472
 - static declaration section 160
 - Static member functions 235
 - Static Procedures (in a class) 474
 - Static section 33
 - stc instruction 402
 - std instruction 402
 - Stdcall procedure option 348
 - sti instruction 402
 - storage declaration section 164
 - stosb instruction 402
 - stosd instruction 402
 - stosw instruction 402
 - str.strRec definition 497
 - string compile-time function 260
 - String concatenation operator (+) in constant expressions 130
 - String Constants 30
 - String constants 117
 - String data type 102
 - String representation 497
 - String/Pattern matching compile-time function 266
 - Structure, accessing fields of... 494
 - Structured constants 30, 118
 - Structured goto statement 337
 - Structures as structure fields 495
 - sub instruction 36, 402
 - Subtraction operator (-) in constant expressions 130
 - super keyword 224
 - switch..case..default..endswitch statement 339
 - sym command-line option 87
 - Symbol related compile-time functions 272
 - Symbol table display 90
 - symbol table dump after compile (command-line option) 87
- T
- TASM 1
 - TASM as back-end assembler (command-line option) 87
 - tasm command-line option 87
 - TByte data type 102
 - Temporary working file directory (command-line option) 86
 - test command-line option 87
 - test instruction 36, 402
 - Text data type 103
 - THEN 328
 - THIS 478
 - This (reference to class object) 233
 - thread command-line option 86
 - Thread-safe code generation 86
 - THUNK constants (pass by name/lazy parameters) 213
 - Thunk data type 103
 - Thunks 112
 - try..always..endtry statement 326
 - TRY..EXCEPTION..ENDTRY statement 321

- Type checking *124*
 - Type coercion *44, 125, 394*
 - Type compatibility *104*
 - type declaration section *150*
 - Type promotion *124*
 - Type sections *32*
- U
- UCR Standard Library for 80x86 Programmers *5*
 - ud2 instruction *402*
 - Unicode Character Constants *117*
 - Unicode data type *102*
 - Unicode String Constants *117*
 - Union constants *120*
 - Union data types *105*
 - Units *345*
 - UNPROTECTED clause in the TRY..ENDTRY statement *324*
 - Uns128 data type *102*
 - Uns16 data type *102*
 - Uns32 data type *102*
 - Uns64 data type *102*
 - Uns8 data type *102*
 - Unsigned data types *103*
 - until clause in repeat..until statement *334*
 - Untyped reference parameters *172, 207, 208*
 - User-defined compilation errors *301*
 - User-defined exceptions *323*
- V
- v command-line option *87*
 - Val (pass by value) parameter option *171*
 - val keyword *146*
 - Valres (pass by value/result) parameter option *171*
 - Value parameters *193*
 - Value/Result parameters *203*
 - Var (pass by reference) parameter option *171*
 - VAR (untyped reference parameters) *172*
 - var declaration section *153*
 - var keyword *153*
 - Var type (untyped reference parameters) *208*
 - Variable parameter lists in macros *248*
 - _vars_ constant *182*
 - Constants
 - _vars_ *182*
 - Verbose compile switch (command-line option) *87*
 - Virtual member functions *235*
 - Virtual method calls *475*
 - Virtual method table *481*
 - Virtual Method Table pointer *233*
 - Virtual Method Tables *482*
 - Virtual method tables *233*
 - Virtual Methods *474*
 - _VMT_ *233, 483*
 - VMT *233, 481, 483*
- W
- w command-line option *87*
 - wait instruction *402*
 - wbinvd instruction *402*
 - WChar data type *102*
 - Webster *7*
 - WHILE..ENDWHILE statement *333*
 - win32 command-line option *87*
 - Win32 OS PE/COFF object code generation *87*
 - Windows (GUI) applications *90*
 - Windows API external names *346*
 - Windows installation of HLA *45*
 - Windows Structured Exception Handler *323*
 - Word data type *102*
 - Working files directory (command-line option) *86*
- X
- x
 - name command-line option *86*
 - xadd instruction *42, 418*
 - XCHG instruction *37, 403*
 - xlat instruction *402*
 - xor instruction *36, 402*

Y

Yield *112*

Z

Zero extension instruction *39, 411*

Zero operand instructions *36*

ZString data type *102*