HLA v2.0 Intermediate Code Design Documentation

This document describes the internal format of HLA v2.0s intermediate code that the code generator uses. This document assumes that the reader is familiar with compiler theory and terminology. This document also assumes that the reader is comfortable with the HLA language and 80x86 assembly language in general.

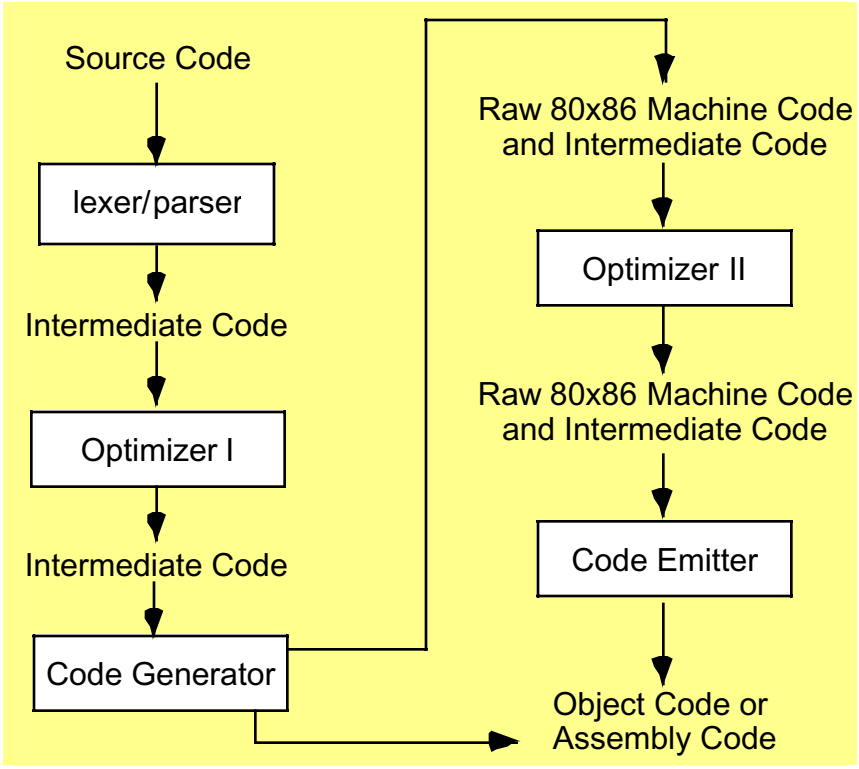
Note: this document is provided strictly as an idea of where HLA v2.0 is headed. In the year since I ve written this document, I ve completely changed how the internal representation will work. And I m sure it will change again before HLA v2.0 is complete. But this document does give a small preview of how HLA v2.0 will handle code generation, even if all the details are incorrect -RLH.

Purpose of the Intermediate Code

HLA v1.x directly emitted assembly language code within the semantic actions of the productions associated with the grammar. While this form of code generation is simple and easy, it is also quite limiting. Furthermore, attempting to emit code for a different assembler (or directly generate object code) is extremely difficult because one has to modify so many different code emission statements throughout the compiler. Although HLA v1.x was modified to emit TASM as well as MASM output, the effort to do this was quite large (especially considering the similarity of the two assemblers). An attempt to generate Gas code was abandoned when it became clear that the effort would be too great and the effort would be put to better use in the development of HLA v2.0.

One lesson learned from the development of HLA v1.x is that the compiler itself should not attempt to generate target machine code. Instead, like many HLL compilers, HLA v2.0 generates an intermediate code that a code generator can then translate into the final output. This scheme has several advantages; but the two primary advantages are that the lexer and parser can work with a single output format that is independent of the final code emission; the second advantage is that this scheme allows one to easily write a multiple back ends to the compiler that emit different assembly and object code formats. Since most of the real work is done by the parser, the back ends are very easy to write. Indeed, a third-party could easily supply a code generator for HLA.

To understand how to write a code generator, it s a good idea to first look at the HLA v2.0 s structure:



As you can see in this diagram, the lexer/parser reads the HLA source code and generates an intermediate code. Most traditional assemblers (and, to a certain extent, HLA v1.x) consist of this component and nothing more; though traditional assemblers typically emit object code directly rather than an intermediate code. Some, more powerful, assemblers (e.g., MASM and TASM) also support the Optimizer II phase, which we ll discuss in a moment.

The second stage of the HLA compiler is an optimizer that reads the intermediate code emitted by the parser and applies transformations that make the intermediate code representation shorter, faster, or both. Note that this phase is optional in the compiler and may not be present or may be disabled by the programmer (this is a feature intended for HLA v3.0, so it may never be present at all in HLA v2.0, except as a stub; HLA, however, assumes that there is a module present that does optimization, even if that module does a null optimization). This is called the Optimizer I phase because HLAs architecture supports two different optimization phases.

The third stage in the HLA compiler is the code generator. In general, this phase is responsible for translating the intermediate code into 80x86 machine code; as an alternative, the code generator can generate assembly language for an assembler such as MASM, TASM, or Gas that provide an Optimizer II phase (see the description in the next paragraph).

The fourth stage in the HLA compiler is the Optimizer II phase. This phase operates on 80x86 machine code (instead of the intermediate code that the Optimizer I phase works with). This phase attempts to reduce all displacements in the object code to their minimum size. The Code Generator phase always generates displacements using their maximum size (i.e., typically four bytes). Many instructions support shorter displacements or immediate fields; the Optimizer II phase makes multiple passes over the raw machine code in order to reduce the size of these displacements and immediate data fields as much as possible (multiple passes may be necessary because shortening a set of displacements might reduce the distance betwee a jump and its target sufficiently to allow that jump to be optimized as well; shortening that jump on a second pass may make the distance between some other jump and its target short enough to use a smaller displacement, ..., etc.).

The last stage in the HLA compiler is the Code Emitter phase. This stage takes the raw 80x86 machine code produced by the Optimizer II phase and generates an object code file suitable for use by a linker program. Alternately, this stage could emit assembly code for an assembler that doesn t automatically reduce displacement sizes in x86 instructions.

These stages in the HLA compiler all all relatively independent of one another. The only communication between these stages of the compiler is via in-memory data structures — HLAs intermediate code representation and the symbol table. An enterprising software engineer could replace (or write) any one of these modules. For example, someone who wants to create their own assembler with their own particular syntax could replace the lexer/parser module (often called the front end) and not have to worry about optimization or code generation/emission. Similarly, if someone needed an assembler that produces a certain object file format, and no such assembler exists, then they could write a new Code Emitter module and any assembler (including HLA) that uses this module could produce the object code format that person requires.

The key interface between each of the stages in the HLA compiler is the Intermediate Code/Raw 80x86 Machine Code data structure (note that Intermediate Code and Raw 80x86 Machine Code are not two different formats, HLAs intermediate code format includes both meta-data and raw machine code; various stages of the compiler refine the pure intermediate code to pure machine code). This document defines the format for HLAs intermediate code so that other programmers can work, independently, on the various modules in the HLA (or other assembler) compiler.

Interface Between Stages

Although HLA v2.0 is written in HLA (assembly language), the interface between the modules was carefully designed to allow the use of almost any programming language. Indeed, with a little bit of effort, one could easily modify the system so that each of the stages is a different program and the information is passed between the stages in a file (this is not the default interface because it is very inefficient, but were this necessary, it could be done). For efficiency reasons, HLA v2.0 assumes that each stage is implemented via a procedure call and that all the stages (procedures) are statically linked together to form the compiler; HLA v2.0 assumes that it is possible to pass a pointer to a memory structure from one stage to the next. There is no reason you couldn t use dynamically linked library modules or any other technology as long as the modules can share a common memory space. If it is not possible to share the same memory space between the stages, then it would be very easy to add a stage between the lexer/parser and the Optimizer I stages that writes the data to a memory-mapped file (which you can share between different processes). Currently, however, HLA v2.0 simply assumes that it can pass a pointer to the start of a list from one stage to the next.

Except for the lexer/parser stage, the calling sequence to each of the phases in the compiler is the following:

```
procedure phase( IntCode:ptrToIntCode_t ); @cdecl;  // HLA Declaration
void phase( IntCode_t *IntCode );                   // C Declaration
```

Any language (e.g., C++, Delphi, etc.) that allows you to write a procedure that can be called from C with a single pointer (four-byte double word pointer), leaving that pointer on the stack to be cleaned up by the caller, should be reasonable for writing one of the stages. Here is the calling conventions the HLA compiler assumes:

- Caller passes the four-byte pointer on the stack.
- The caller removes the four-byte pointer from the stack upon return.
- The procedure need only preserve ESP and EBP.
- The procedure makes no assumptions about initialization prior to the call (e.g., standard library or run-time system initialization). Remember, each stage could be written in a different language and there s no guarantee that the main program (the lexer/parser stage) called the run-time initialization code required by a given module.
- Once a stage completes, it is never again called (during the same compilation).
- Except for the intermediate code data structure, and any previous data on the stack, the module completely cleans up after itself before it returns. It closes any open files, frees any allocated memory, etc.
- One module may only communicate information to another module via the intermediate format. No other shared, global, data structures are present, nor do two stages communicate information via a file or database.

The Intermediate Code Data Exchange Format

The HLA interrmediate code format uses the following data structures:

```
type
      intCode_t:
            record

                  Next        :pointer to intCode;
                  LineNum     :int32;
                  SourceLine  :pointer to char;
                  Offset      :dword;
                  Size        :uns32;
                  TypePrefix  :word;

            endrecord;
```

The *intCode_t* type is an abstract record definition that defines the fields that are common to all intermediate code records. The HLA compiler doesn t actually declare any objects of this type; instead, other intermediate code record simply inherit this data type.

The *Next* field forms a linked list of intermediate code records. At the highest level, this field links together the sequence of instructions and/or data records appearing within a segment/section. For example, the code (text) segment consists of a list of all the instructions appearing in the source file. The *Next* field links the statements and data definitions appearing in the code segment into a linear list. The code generator emits the instructions in the order they appear in this list. For data segments, the *Next* field links together the declarations for the particular segment (static, readonly, storage, or user-defined) so the code generator may emit the data definitions in a contiguous manner to the object file.

The *LineNum* field contains the line number in the source file associated with this instruction or data declaration. Note that for MACRO invocations and TEXT expansions, this is the line number in the original source file where the expansion began (i.e., the line number of the statement containing the MACRO or TEXT identifier).

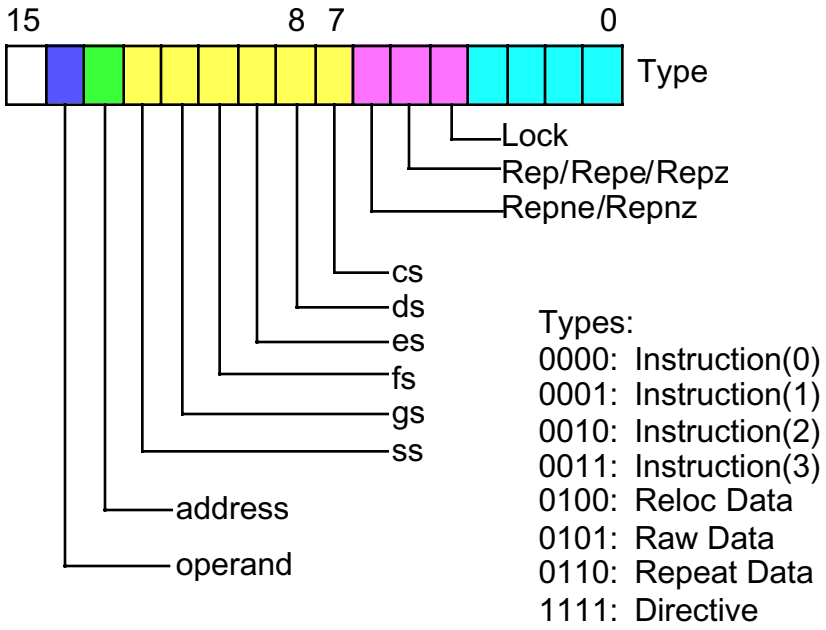
The *SourceLine* field contains a pointer to the start of the source line associated with this code/data record. For MACRO expansions, this pointer actually points into the MACRO definition (unlike the *LineNum* field above). The code generator/emitter primarily uses this field for producing assembly listings.

The *Offset* field contains the numeric offset of this instruction/data value within its segment.

The *Size* field specifies the size of this instruction or data record, in bytes. Note that machine instructions are always 15 bytes or less, but data records can be any (practical) size.

The *TypePrefix* field serves two purposes: (1) it defines the type of the record data immediately following the *TypePrefix* field, and (2) for instructions this field specifies any prefix bytes appearing before the instruction opcode. This field takes the following format:

Prefix Bitmap Format and Record Type



The low-order four bytes define the record type. Values zero through three specify a machine instruction type with zero, one, two, or three operands. If these bits contain one of these values, then they specify one of the *intInstr0_t, intInstr1_t, intInstr2_t,* or *intInstr3_t* types (respectively, see the definitions below).

If the type field (low-order four bits) contains one of the values zero through three, then bits five through fourteen also specify any instruction prefix bytes appearing before the opcode. Note that prefixes sharing the same color above are mutually exclusive.

If the type field (low-order four bits) contains a value in the range four through six, then the intermediate code record defines a data record using one of the types *RelocData_t* (type=4), *RawData_t* (type=5), or *RepeatData_t* (type=6). *RelocData_t* records handle data records that specify a relocatable address as the operand(s). This intermediate code record type handles double-word data declarations containing expressions like &symbol , &symbol[n] , and &symbol1[n] - &symbol2[m] . *RawData_t* intermediate code records handle the emission of raw bytes to the object file; this is the most common data record. *RepeatData_t* intermediate code records handle repeated data (in a recusive fashion, so you can repeat a data sequence that contains repetition). Note that the prefix bits must all be zero for the data defining intermediate code records.

If the type field (low-order four bits) contains %1111 then this intermediate code record is a directive that the HLA parser emits to provide some sort of hint to the optimizers and the code generator. These directives do things such as turn code optimization on or off, deliniate procedures, parameters, data structures, and other objects, pass parameters to the linker, and so on. See the section on intermediate code directives for more details.

```
operand_t:
      record

            BaseSymbol  :symNodePtr_t;
            MinusSymbol :symNodePtr_t;
            PlusDisp    :dword;
            BaseReg     :byte;
            ScaledIndex :byte;
```

```
        AdrsMode    :byte;
        Size        :byte;

endrecord;
```

The HLA intermediate code format uses the *operand_t* data type to represent operands of an x86 machine instruction. The following paragraphs provide the definitions for each of these fields.

BaseSymbol: This field contains NULL or a pointer to an HLA symbol table entry. If this field is non-NULL, it indicates that the operand is either a memory address involving some sort of variable (VAR, STATIC, READONLY, STORAGE, or SEGMENT) along with an optional base and/or index register, or the addressing mode is immediate and the expression takes the form &staticVar or &staticVar1-&staticVar2 (with optional constant indicies on each identifier). The *BaseSymbol* field points at the symbol table entry for the memory variable s identifier (*staticVar1* in the latter case) associated with this operand.

MinusSymbol: This field contains a pointer to the symbol table entry for the second identifier in immediate expressions of the form &staticVar1-&staticVar2 . This field contains NULL if the expression does not take this form.

PlusDisp: This field contains the numeric displacement (exclusive of any symbolic variable names) associated with this operand; this field also holds the (non-relocatable) immediate constant data for the immediate addressing mode. E.g., for the operand [ebx+5] the *PlusDisp* field contains five. For an operand of the form &staticVar[4] the *PlusDisp* field contains four (the *BaseSymbol* field handles the actual offset of the *staticVar* symbol). An operand of the form &staticVar1[8] - &staticVar2[2] has a *PlusDisp* value of six since HLA computes the displacement as 8-2 (since we must subtract the offset into *staticVar2* from the offset into *staticVar*).

BaseReg: This field serves two purposes. For register operands, this field specifies which register to use. For memory operands that use a base register, this field specifies the base register to use. Note that the *AdrsMode* field specifies whether this field is a register, base register, or is unused. This field uses the following values:

Table 1: BaseReg Values

BaseReg	Register
0	AL
1	CL
2	DL
3	BL
4	AH
5	CH
6	DH
7	BH
8	AX
9	CX
A	DX
B	BX
C	SP

Table 1: BaseReg Values

BaseReg	Register
D	BP
E	SI
F	DI
10	EAX
11	ECX
12	EDX
13	EBX
14	ESP
15	EBP
16	ESI
17	EDI
18	ST0
19	ST1
1A	ST2
1B	ST3
1C	ST4
1D	ST5
1E	ST6
1F	ST7
20	MM0
21	MM1
22	MM2
23	MM3
24	MM4
25	MM5
26	MM6
27	MM7
28	XMM0
29	XMM1
2A	XMM2

Table 1: BaseReg Values

BaseReg	Register
2B	XMM3
2C	XMM4
2D	XMM5
2E	XMM6
2F	XMM7
30	DR0
31	DR1
32	DR2
33	DR3
34	DR4
35	DR5
36	DR6
37	DR7
38	N/A (TR0)
39	N/A (TR1)
3A	N/A (TR2)
3B	TR3
3C	TR4
3D	TR5
3E	TR6
3F	TR7
40	CR0
41	CR1
42	CR2
43	CR3

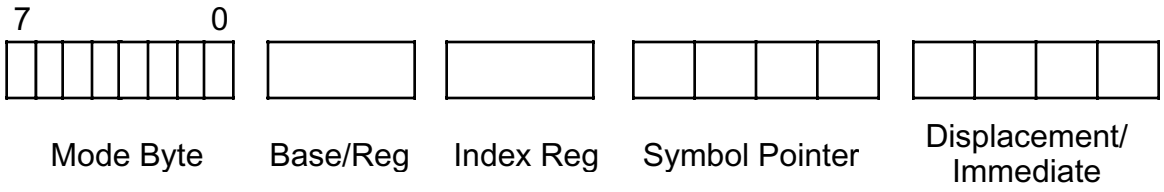
Note that if the *AdrsMode* byte specifies a memory address using a base register, then the only legal values in the *BaseReg* field are the values for the 32-bit general purpose registers (EAX..EDI).

ScaledIndex: This field specifies an index register and a scaling factor. This field is only meaningful if the *AdrsMode* byte specifies a scaled indexed addressing mode. This field uses the following format:

```
intInstr0_t:
      record inherits( intCode )

            SIB         :byte;
            ModRegRM    :byte;

            Instruction :
                  union
                        InstrTkn    :dword
                        bytes       :byte[ 3];
                        record
                              prefix      :byte;
                              opcode      :byte;
                              ORreg       :byte;
                        endrecord;
                  endunion;

      endrecord;

intInstr1_t:
      record inherits( intInstr0_t )

            Operand1    :operand_t;

      endrecord;

intInstr2_t:
      record inherits( intInstr1_t )

            Operand2    :operand_t;

      endrecord;

intInstr3_t:
      record inherits( intInstr2_t )

            Operand3    :operand_t;

      endrecord;

RelocData_t:
      record inherits( intCode_t )

            BaseSymbol  :symNodePtr_t;
            MinusSymbol :symNodePtr_t;
            PlusDisp    :dword;

      endrecord;

RawData_t:
      record inherits( intCode_ t )

            data  :byte[ 1];  // Actually Size bytes long.
```

```
        endrecord;

    RepeatData_t:
        record inherits( intCode_t )

            RepeatCnt   :uns32;
            PtrToData   :pointer to intCode_t;

        endrecord;

    intDirective_t:
        record inherits( intCode )

            DirVal      :word;
            symbol      :symNodePtr_t;

        endrecord;
```

Operand Format



Mode [0:3]
- 0000 - immediate
- 0001 - register
- 0010 - [disp]
- 0011 - Mem
- 0100 - [reg]
- 0101 - [reg + disp]
- 0110 - [reg + reg*scale]
- 0111 - [reg + reg*scale + disp]
- 1000 - Mem [reg]
- 1101 - Mem [reg + disp]
- 1110 - Mem [reg + reg*scale]
- 1111 - Mem [reg + reg*scale + disp]

Mode [4:5]
- 00 - scale = 1
- 01 - scale = 2
- 10 - scale = 4
- 11 - scale = 8

Mode [6]-
- Reserved

Mode [7]-
- 0- Ignore Operand
- 1- L.O. 6 bits describe operand

Raw x86 Instruction Format