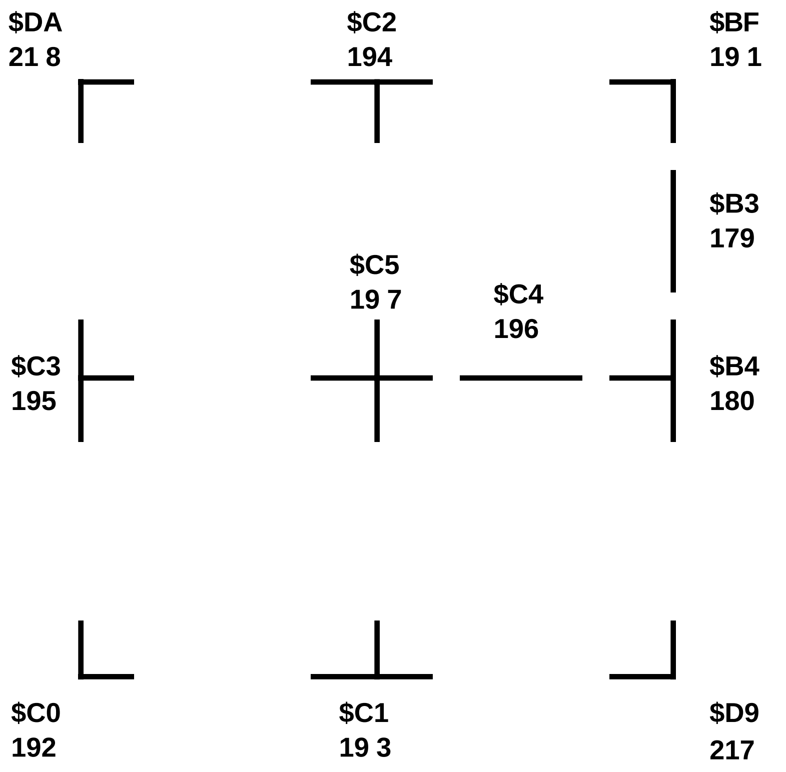
Answers to Selected Exercises Appendix A

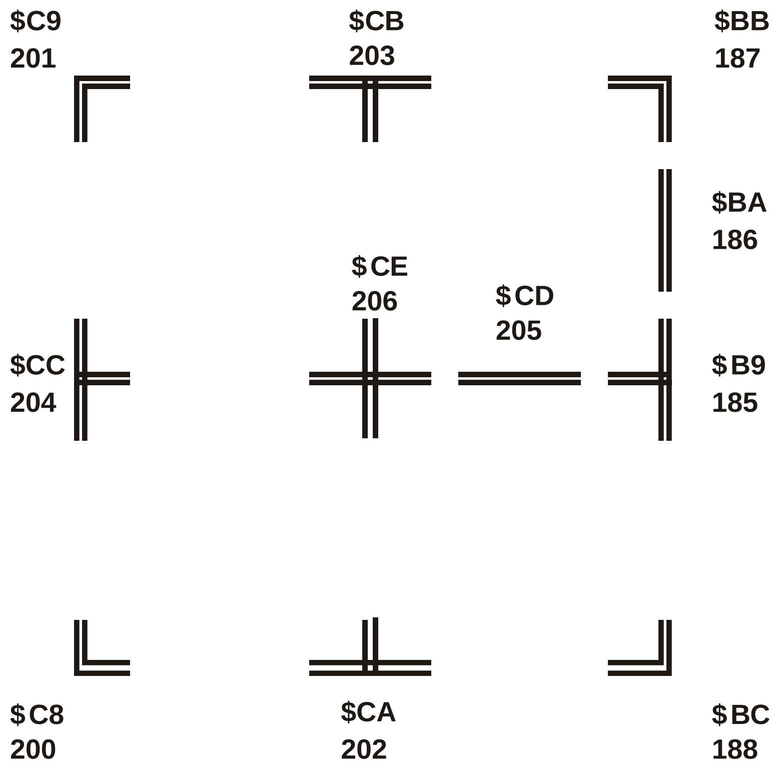
To be written.

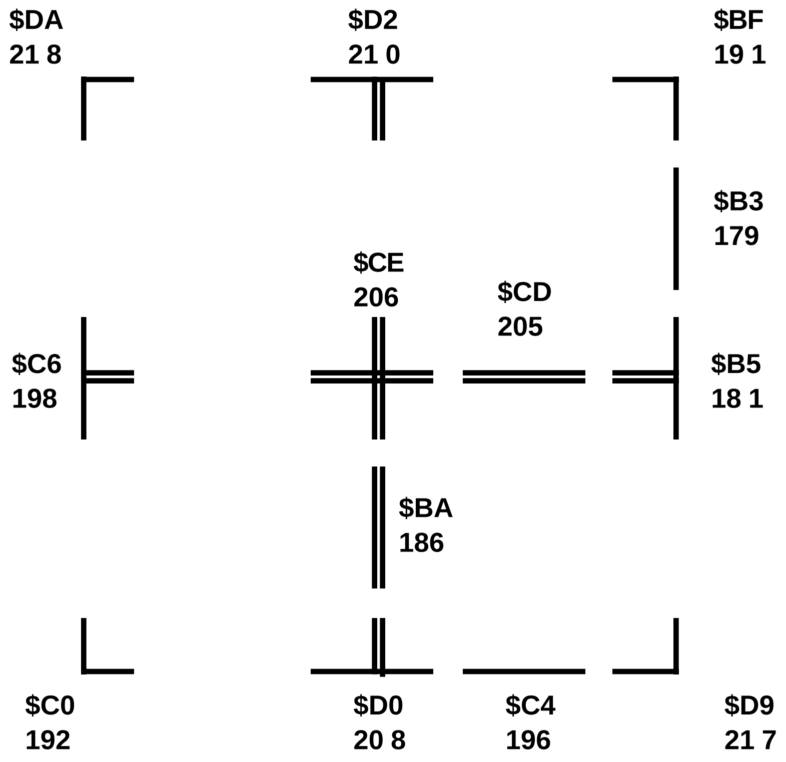
My apologies that this isn't ready yet, but other chapters and appendices in this text have a higher priority. I will get around to this appendix eventually.

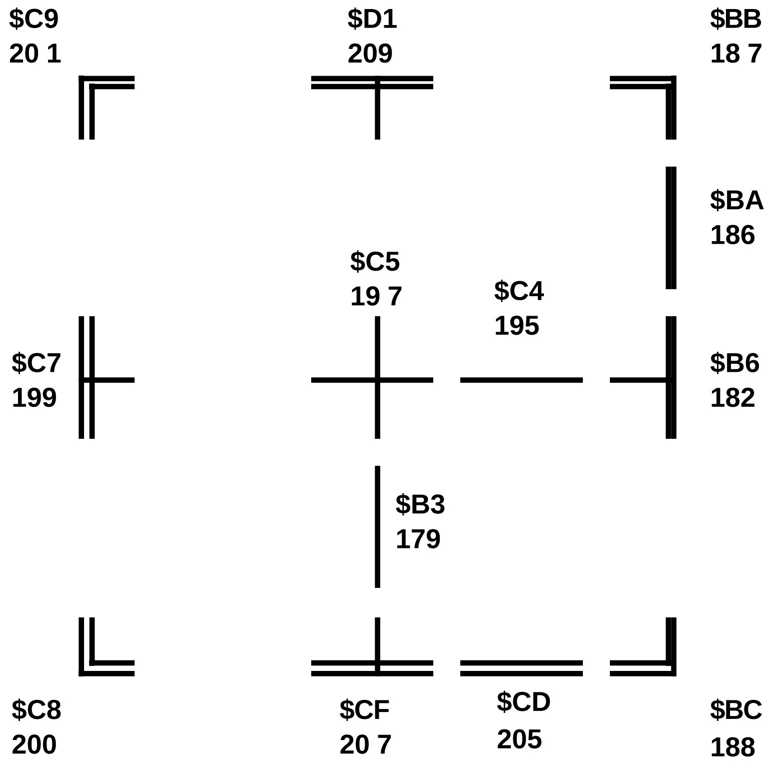
In the meantime, if you have some questions about the answers to any exercises in this text, please feel free to post a question to one of the internet newsgroups like "comp.lang.asm.x86" or "alt.lang.asm". Because of the high volume of email I receive daily, I will not answer questions sent to me via email. Note that posting the message to the net is very efficient because others get to share the solution. So please post your questions there.

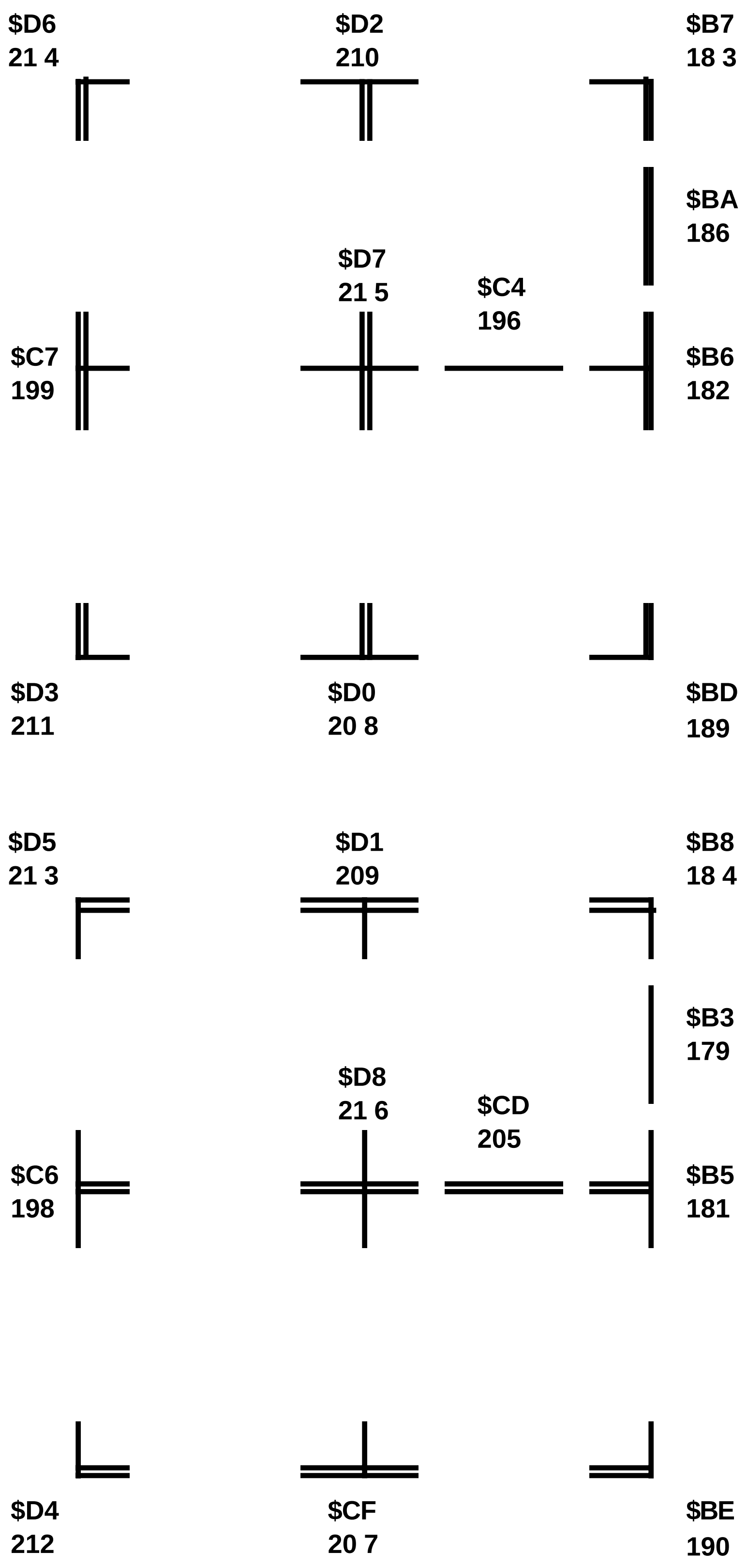
Console Graphic Characters Appendix B






$DA
21 8
$D2
21 0
$BF
19 1
$B3
179
$CE
206
$CD
205
$C6
198
$B5
18 1
$BA
186
$C0
192
$D0
20 8
$C4
196
$D9
21 7


$C9
20 1
$D1
209
$BB
18 7
$BA
186
$C5
19 7
$C4
195
$C7
199
$B6
182
$B3
179
$C8
200
$CF
20 7
$CD
205
$BC
188


$D6
21 4
$D2
210
$B7
18 3
$BA
186
$D7
21 5
$C4
196
$C7
199
$B6
182
$D3
211
$D0
20 8
$BD
189
$D5
21 3
$D1
209
$B8
18 4
$B3
179
$D8
21 6
$CD
205
$C6
198
$B5
181
$D4
212
$CF
20 7
$BE
190

HLA Programming Style Guidelines — Appendix C

C.1 Introduction

Most people consider assembly language programs difficult to read. While there are a multitude of reasons why people feel this way, the primary reason is that assembly language does not make it easy for programmers to write readable programs. This doesn't mean it's impossible to write readable programs, only that it takes an extra effort on the part of an assembly language programmer to produce readable code.

One of the design goals of the High Level Assembler (HLA) was to make it possible for assembly language programmers to write readable assembly language programs. Nevertheless, without discipline, pandemonium will result in any program of any decent size. Even if you adhere to a fixed set of style guidelines, others may still have trouble reading and understanding your code. Equally important to following a set of style guidelines is that you following a generally accepted set of style guidelines; guidelines that others are familiar and agree with. The purpose of this appendix, written by the designer of the HLA language, is to provide a consistent set of guidelines that HLA programmers can use consistently. Unless you can show a good reason to violate these rules, you should following them carefully when writing HLA programs; other HLA programmers will thank you for this.

C.1.1 Intended Audience

Of course, an assembly language program is going to be nearly unreadable to someone who doesn't know assembly language. This is true for almost any programming language. Other than burying a tutorial on 80x86 assembly language in a program's comments, there is no way to address this problem[1] other than to assume that the reader is familiar with assembly language programming and specifically HLA.

In view of the above, it makes sense to define an "intended audience" that we intend to have read our assembly language programs. Such a person should:

- Be a reasonably competent 80x86 assembly language/HLA programmer.
- Be reasonably familiar with the problem the assembly language program is attempting to solve.
- Fluently read English[2].
- Have a good grasp of high level language concepts.
- Possess appropriate knowledge for someone working in the field of Computer Science (e.g., understands standard algorithms and data structures, understands basic machine architecture, and understands basic discrete mathematics).

C.1.2 Readability Metrics

One has to ask "What is it that makes one program more readable than another?" In other words, how do we measure the "readability" of a program? The usual metric, "I know a well-written program when I see one" is inappropriate; for most people, this translates to "If your programs look like my better programs then they are readable, otherwise they are not." Obviously, such a metric is of little value since it changes with every person.

To develop a metric for measuring the readability of an assembly language program, the first thing we must ask is "Why is readability important?" This question has a simple (though somewhat flippant) answer:

1. Doing so (inserting an 80x86 tutorial into your comments) would wind up making the program *less* readable to those who already know assembly language since, at the very least, they'd have to skip over this material; at the worst they'd have to read it (wasting their time).
2. Or whatever other natural language is in use at the site(s) where you develop, maintain, and use the software.

Readability is important because programs are *read* (furthermore, a line of code is typically read ten times more often than it is written). To expand on this, consider the fact that most programs are read and maintained by other programmers (Steve McConnell claims that up to ten generations of maintenance programmers work on a typical real world program before it is rewritten from scratch; furthermore, they spend up to 60% of their effort on that code simply figuring out how it works). The more readable your programs are, the less time these other people will have to spend figuring out what your program does. Instead, they can concentrate on adding features or correcting defects in the code.

For the purposes of this document, we will define a "readable" program as one that has the following trait:

- A "readable" program is one that a competent programmer (one who is familiar with the problem the program is attempting to solve) can pick up, without ever having seen the program before, and fully comprehend the entire program in a minimal amount of time.

That's a tall order! This definition doesn't sound very difficult to achieve, but few non-trivial programs ever really achieve this status. This definition suggests that an appropriate programmer (i.e., one who is familiar with the problem the program is trying to solve) can pick up a program, read it at their normal reading pace (just once), and fully comprehend the program. Anything less is not a "readable" program.

Of course, in practice, this definition is unusable since very few programs reach this goal. Part of the problem is that programs tend to be quite long and few human beings are capable of managing a large number of details in their head at one time. Furthermore, no matter how well-written a program may be, "a competent programmer" does not suggest that the programmer's IQ is so high they can read a statement a fully comprehend its meaning without expending much thought. Therefore, we must define readability, not as a boolean entity, but as a scale. Although truly unreadable programs exist, there are many "readable" programs that are less readable than other programs. Therefore, perhaps the following definition is more realistic:

- A readable program is one that consists of one or more *modules.* A competent program should be able to pick a given module in that program and achieve an 80% comprehension level by expending no more than an average of one minute for each statement in the program.

An 80% comprehension level means that the programmer can correct bugs in the program and add new features to the program without making mistakes due to a misunderstanding of the code at hand.

C.1.3 How to Achieve Readability

The "I'll know one when I see one" metric for readable programs provides a big hint concerning how one should write programs that are readable. As pointed out early, the "I'll know it when I see it" metric suggests that an individual will consider a program to be readable if it is very similar to (good) programs that this particular person has written. This suggests an important trait that readable programs must possess: consistency. If all programmers were to write programs using a consistent style, they'd find programs written by others to be similar to their own, and, therefore, easier to read. This single goal is the primary purpose of this appendix - to suggest a consistent standard that everyone will follow.

Of course, consistency by itself is not good enough. Consistently bad programs are not particularly easy to read. Therefore, one must carefully consider the guidelines to use when defining an all-encompassing standard. The purpose of this paper is to create such a standard. However, don't get the impression that the material appearing in this document appears simply because it sounded good at the time or because of some personal preferences. The material in this paper comes from several software engineering texts on the subject (including Elements of Programming Style, Code Complete, and Writing Solid Code), nearly 20 years of personal assembly language programming experience, and research that led to the development of a set of generic programming guidelines for industrial use.

This document assumes consistent usage by its readers. Therefore, it concentrates on a lot of mechanical and psychological issues that affect the readability of a program. For example, uppercase letters are harder to read than lower case letters (this is a well-known result from psychology research). It takes longer for a human being to recognize uppercase characters, therefore, an average human being will take more time to read text written all in upper case. Hence, this document suggests that one should avoid the use of uppercase sequences in a program. Many of the other issues appearing in this document are in a similar vein; they suggest minor changes to the way you might write your programs that make it easier for someone to recognize some pattern in your code, thus aiding in comprehension.

C.1.4 How This Document is Organized

This document follows a top-down discussion of readability. It starts with the concept of a program. Then it discusses modules. From there it works its way down to procedures. Then it talks about individual statements. Beyond that, it talks about components that make up statements (e.g., instructions, names, and operators). Finally, this paper concludes by discussing some orthogonal issues.

Section Two discusses programs in general. It primarily discusses documentation that must accompany a program and the organization of source files. It also discusses, briefly, configuration management and source code control issues. Keep in mind that figuring out how to build a program (make, assemble, link, test, debug, etc.) is important. If your reader fully understands the "heapsort" algorithm you are using, but cannot build an executable module to run, they still do not fully understand your program.

Section Three discusses how to organize modules in your program in a logical fashion. This makes it easier for others to locate sections of code and organizes related sections of code together so someone can easily find important code and ignore unimportant or unrelated code while attempting to understand what your program does.

Section Four discusses the use of procedures within a program. This is a continuation of the theme in Section Three, although at a lower, more detailed, level.

Section Five discusses the program at the level of the statement. This (large) section provides the meat of this proposal. Most of the rules this paper presents appear in this section.

Section Six discusses comments and other documentation appearing within the source code.

Section Seven discusses those items that make up a statement (labels, names, instructions, operands, operators, etc.) This is another large section that presents a large number of rules one should follow when writing readable programs. This section discusses naming conventions, appropriateness of operators, and so on.

Section Eight discusses data types and other related topics.

C.1.5 Guidelines, Rules, Enforced Rules, and Exceptions

Not all rules are equally important. For example, a rule that you check the spelling of all the words in your comments is probably less important than suggesting that the comments all be in English[3]. Therefore, this paper uses three designations to keep things straight: Guidelines, Rules, and Enforced Rules.

A Guideline is a suggestion. It is a rule you should follow unless you can verbally defend why you should break the rule. As long as there is a good, defensible, reason, you should feel no apprehension violated a guideline. Guidelines exist in order to encourage consistency in areas where there are no good reasons for choosing one methodology over another. You shouldn't violate a Guideline just because you don't like it -- doing so will make your programs inconsistent with respect to other programs that do follow the Guideline (and, therefore, harder to read), however, you shouldn't lose any sleep because you violated a Guideline.

Rules are much stronger than Guidelines. You should never break a rule unless there is some external reason for doing so (e.g., making a call to a library routine forces you to use a bad naming convention). Whenever you feel you must violate a rule, you should verify that it is reasonable to do so in a peer review with at least two peers. Furthermore, you should explain in the program's comments why it was necessary

3. You may substitute the local language in your area if it is not English.

to violate the rule. Rules are just that -- rules to be followed. However, there are certain situations where it may be necessary to violate the rule in order to satisfy external requirements or even make the program more readable.

Enforced Rules are the toughest of the lot. You should *never* violate an enforced rule. If there is ever a true need to do this, then you should consider demoting the Enforced Rule to a simple Rule rather than treating the violation as a reasonable alternative.

An Exception is exactly that, a known example where one would commonly violate a Guideline, Rule, or (very rarely) Enforced Rule. Although exceptions are rare, the old adage "Every rule has its exceptions..." certainly applies to this document. The Exceptions point out some of the common violations one might expect.

Of course, the categorization of Guidelines, Rules, Enforced Rules, and Exceptions herein is one man's opinion. At some organizations, this categorization may require reworking depending on the needs of that organization.

C.1.6 Source Language Concerns

This document will assume that the entire program is written in 80x86 assembly language using the HLA assembler/compiler. Although this organization is rare in commercial applications, this assumption will, in no way, invalidate these guidelines. Other guidelines exist for various high level languages (including a set written by this paper's author). You should adopt a reasonable set of guidelines for the other languages you use and apply these guidelines to the 80x86 assembly language modules in the program.

C.2 Program Organization

A source program generally consists of one or more source, object, and library files. As a project gets larger and the number of files increases, it becomes difficult to keep track of the files in a project. This is especially true if a number of different projects share a common set of source modules. This section will address these concerns.

C.2.1 Library Functions

A library, by its very nature, suggests stability. Ignoring the possibility of software defects, one would rarely expect the number or function of routines in a library to vary from project to project. A good example is the "HLA Standard Library." One would expect "stdout.put" to behave identically in two different programs that use the Standard Library. Contrast this against two programs, each of which implement their own version of *stdout.put*. One could not reasonably assume both programs have identical implementations[4]. This leads to the following rule:

Rule:	Library functions are those routines intended for common reuse in many different assembly language programs. All assembly language (callable) libraries on a system should exist as ".lib" files and should appear in a "\lib" or "\hlalib" subdirectory.
Guideline:	"\hlalib" is probably a better choice if you're using multiple languages since those other languages may need to put files in a "\lib" directory.
Exception:	It's probably reasonable to leave the HLA Standard Library's "hlalib.lib" file in the "\hla\hlalib" directory since most people expect it there.

4. In fact, just the opposite is true. One should get concerned if both implementations *are* identical. This would suggest poor planning on the part of the program's author(s) since the same routine must now be maintained in two different programs.

The rule above ensures that the library files are all in one location so they are easy to find, modify, and review. By putting all your library modules into a single directory, you avoid configuration management problems such as having outdated versions of a library linking with one program and up-to-date versions linking with other programs.

C.2.2 Common Object Modules

This document defines a *library* as a collection of object modules that have wide application in many different programs. The HLA Standard Library is a typical example of a library. Some object modules are not so general purpose, but still find application in two or more different programs. Two major configuration management problems exist in this situation: (1) making sure the ".obj" file is up-to-date when linking it with a program; (2) Knowing which modules use the module so one can verify that changes to the module won't break existing code.

The following rules takes care of case one:

Rule: If two different program share an object module, then the associated source, object, and makefiles for that module should appear in a subdirectory that is specific to that module (i.e., no other files in the subdirectory). The subdirectory name should be the same as the module name. If possible, you should create a set of link/alias/shortcuts to this subdirectory and place these links in the main directory of each of the projects that utilize the module. If links are not possible, you should place the module's subdirectory in a "\common" subdirectory.

Enforced Rule: Every subdirectory containing one or more modules should have a make file that will automatically generate the appropriate, up-to-date, ".obj" files. An individual, a batch file, or another make file should be able to automatically generate new object modules (if necessary) by simply executing the make program.

Guideline: Use Microsoft's nmake program. At the very least, use nmake acceptable syntax in your makefiles.

The other problem, noting which projects use a given module is much more difficult. The obvious solution, commenting the source code associated with the module to tell the reader which programs use the module, is impractical. Maintaining these comments is too error-prone and the comments will quickly get out of phase and be worse than useless -- they would be incorrect. A better solution is to create alias and place this alias in the main subdirectory of each program that links the module.

Guideline: If a project uses a module that is not local to the project's subdirectory, create an alias to the file in the project's subdirectory. This makes locating the file very easy.

C.2.3 Local Modules

Local modules are those that a single program/project uses. Typically, the source and object code for each module appears in the same directory as the other files associated with the project. This is a reasonable arrangement until the number of files increases to the point that it is difficult to find a file in a directory listing. At that point, most programmers begin reorganizing their directory by creating subdirectories to hold many of these source modules. However, the placement, name, and contents of these new subdirectories can have a big impact on the overall readability of the program. This section will address these issues.

The first issue to consider is the contents of these new subdirectories. Since programmers rummaging through this project in the future will need to easily locate source files in a project, it is important that you organize these new subdirectories so that it is easy to find the source files you are moving into them. The best organization is to put each source module (or a small group of *strongly related* modules) into its own subdirectory. The subdirectory should bear the name of the source module minus its suffix (or the main module if there is more than one present in the subdirectory). If you place two or more source files in the same directory, ensure this set of source files forms a *cohesive* set (meaning the source files contain code that solve a single problem). A discussion of cohesiveness appears later in this document.

Rule: If a project directory contains too many files, try to move some of the modules to subdirectories within the project directory; give the subdirectory the same name as the source file without the suffix. This will nearly reduce the number of files in half. If this reduction is insufficient, try categorizing the source modules (e.g., FileIO, Graphics, Rendering, and Sound) and move these modules to a subdirectory bearing the name of the category.

Enforced Rule: Each new subdirectory you create should have its own make file that will automatically assemble all source modules within that subdirectory, as appropriate.

Enforced Rule: Any new subdirectories you create for these source modules should appear within the directory containing the project. The only excepts are those modules that are, or you anticipate, sharing with other projects. See “Common Object Modules” on page 1415 for more details.

Stand-alone assembly language programs generally contain a "main" procedure – the first program unit that executes when the operating system loads the program into memory. For any programmer new to a project, this procedure is the *anchor* where one first begins reading the code and the point where the reader will continually refer. Therefore, the reader should be able to easily locate this source file. The following rule helps ensure this is the case:

Rule: The source module containing the *main program* should have the same name as the executable (obviously the suffix will be different). For example, if the "Simulate 886" program’s executable name is "Sim886.exe" then you should find the main program in the "Sim886.hla" source file.

Finding the source file that contains the main program is one thing. Finding the main program itself can be almost as hard. Assembly language lets you give the main program any name you want. However, to make the main procedure easy to find (both in the source code and at the O/S level), you should actually name this program "main". See “Module Organization” on page 1417 for more details about the placement of the main program. An alternative is to give the main program’s source file the name of the project.

Guideline: The name of the main procedure in an assembly language program should be "main" or the name of the entire project.

C.2.4 Program Make Files

Every project, even if it contains only a single source module, should have an associated make file. If someone want to assemble your program, they should not have to worry about what program (e.g., HLA) to use to compile the program, what command line options to use, what library modules to use, etc. They should be able to type "nmake"[5] and wind up with an executable program. Even if assembling the program consists of nothing more than typing the name of the assembler and the source file, you should still have a make file. Someone else may not realize that’s all that is necessary.

Enforced Rule: The main project directory should contain a make file that will automatically generate an executable (or other expected object module) in response to a simple make/nmake command.

Rule: If your project uses object modules that are not in the same subdirectory as the main program’s module, you should test the ".obj" files for those modules and execute the corresponding make files in their directories if the object code is out of date. You can assume that library files are up to date.

Guideline: Avoid using fancy "make" features. Most programmers only learn the basics about make and will not be able to understand what your make file is doing if you fully exploit the make language. Especially avoid the use of default rules since this can create havoc if someone arbitrarily adds or removes files from the directory containing the make file.

5. Or whatever make program you normally use.

C.3 Module Organization

A module is a collection of objects that are logically related. Those objects may include constants, data types, variables, and program units (e.g., functions, procedures, etc.). Note that objects in a module need not be *physically* related. For example, it is quite possible to construct a module using several different source files. Likewise, it is quite possible to have several different modules in the same source file. However, the best modules are physically related as well as logically related; that is, all the objects associated with a module exist in a single source file (or directory if the source file would be too large) and nothing else is present.

Modules contain several different objects including constants, types, variables, and program units (routines). Modules shares many of the attributes with routines (program units); this is not surprising since routines are the major component of a typical module. However, modules have some additional attributes of their own. The following sections describe the attributes of a well-written module.

Note: *Unit* and *package* are both synonyms for the term *module.*

C.3.1 Module Attributes

A module is a generic term that describes a set of program related objects (program units as well as data and type objects) that are somehow coupled. Good modules share many of the same attributes as good program units as well as the ability to hide certain details from code outside the module.

C.3.1.1 Module Cohesion

Modules exhibit the following different kinds of *cohesion* (listed from good to bad):

- Functional or logical cohesion exists if the module accomplishes exactly one (simple) task.
- Sequential or *pipelined* cohesion exists when a module does several sequential operations that must be performed in a certain order with the data from one operation being fed to the next in a “filter-like” fashion.
- Global or *communicational* cohesion exists when a module performs a set of operations that make use of a common set of data, but are otherwise unrelated.
- Temporal cohesion exists when a module performs a set of operations that need to be done at the same time (though not necessarily in the same order). A typical initialization module is an example of such code.
- Procedural cohesion exists when a module performs a sequence of operations in a specific order, but the only thing that binds them together is the order in which they must be done. Unlike sequential cohesion, the operations do not share data.
- State cohesion occurs when several different (unrelated) operations appear in the same module and a state variable (e.g., a parameter) selects the operation to execute. Typically such modules contain a case (switch) or **if..elseif..elseif...** statement.
- No cohesion exists if the operations in a module have no apparent relationship with one another.

The first three forms of cohesion above are generally acceptable in a program. The fourth (temporal) is probably okay, but you should rarely use it. The last three forms should almost never appear in a program. For some reasonable examples of module cohesion, you should consult “Code Complete”.

Guideline: Design good modules! *Good modules exhibit strong cohesion.* That is, a module should offer a (small) group of services that are logically related. For example, a “printer” module might provide all the services one would expect from a printer. The individual routines within the module would provide the individual services.

C.3.1.2 Module Coupling

Coupling refers to the way that two modules communicate with one another. There are several criteria that define the level of coupling between two modules:

- Cardinality- the number of objects communicated between two modules. The fewer objects the better (i.e., fewer parameters).
- Intimacy- how "private" is the communication? Parameter lists are the most private form; private data fields in a class or object are next level; public data fields in a class or object are next, global variables are even less intimate, and passing data in a file or database is the least intimate connection. Well-written modules exhibit a high degree of intimacy.
- Visibility- this is somewhat related to intimacy above. This refers to how visible the data is to the entire system that you pass between two modules. For example, passing data in a parameter list is direct and very visible (you always see the data the caller is passing in the call to the routine); passing data in global variables makes the transfer less visible (you could have set up the global variable long before the call to the routine). Another example is passing simple (scalar) variables rather than loading up a bunch of values into a structure/record and passing that structure/record to the callee.
- Flexibility- This refers to how easy it is to make the connection between two routines that may not have been originally intended to call one another. For example, suppose you pass a structure containing three fields into a function. If you want to call that function but you only have three data objects, not the structure, you would have to create a dummy structure, copy the three values into the field of that structure, and then call the function. On the other hand, had you simply passed the three values as separate parameters, you could still pass in structures (by specifying each field) as well as call the function with separate values. The module containing this later function is more flexible.

A module is *loosely coupled* if its functions exhibit low cardinality, high intimacy, high visibility, and high flexibility. Often, these features are in conflict with one another (e.g., increasing the flexibility by breaking out the fields from a structures [a good thing] will also increase the cardinality [a bad thing]). It is the traditional goal of any engineer to choose the appropriate compromises for each individual circumstance; therefore, you will need to carefully balance each of the four attributes above.

A module that uses loose coupling generally contains fewer errors per KLOC (thousands of lines of code). Furthermore, modules that exhibit loose coupling are easier to reuse (both in the current and future projects). For more information on coupling, see the appropriate chapter in "Code Complete".

Guideline: Design good modules! *Good modules exhibit loose coupling*. That is, there are only a few, well-defined (visible) interfaces between the module and the outside world. Most data is private, accessible only through accessor functions (see information hiding below). Furthermore, the interface should be flexible.

Guideline: Design good modules! *Good modules exhibit information hiding*. Code outside the module should only have access to the module through a small set of public routines. All data should be private to that module. A module should implement an *abstract data type.* All interface to the module should be through a well-defined set of operations.

C.3.1.3 Physical Organization of Modules

Many languages provide direct support for modules (e.g., units in HLA, packages in Ada, modules in Modula-2, and units in Delphi/Pascal). Some languages provide only indirect support for modules (e.g., a source file in C/C++). Others, like BASIC, don't really support modules, so you would have to simulate them by physically grouping objects together and exercising some discipline. The primary mechanism in HLA for hiding names from other modules is to implement a module as an individual source file and publish only those names that are part of the module's interface to the outside world (i.e., EXTERNAL directives in a header file.

Rule: Each module should completely reside in a single source file. If size considerations prevent this, then all the source files for a given module should reside in a subdirectory specifically designated for that module.

Some people have the crazy idea that modularization means putting each function in a separate source file. Such *physical modularization* generally impairs the readability of a program more than it helps. Strive instead for *logical modularization*, that is, defining a module by its actions rather than by source code syntax (e.g., separating out functions).

This document does not address the decomposition of a problem into its modular components. Presumably, you can already handle that part of the task. There are a wide variety of texts on this subject if you feel weak in this area.

C.3.1.4 Module Interface

In any language system that supports modules, there are two primary components of a module: the interface component that publicizes the module visible names and the implementation component that contains the actual code, data, and private objects. HLA (like most assemblers) uses a scheme that is very similar to the one C/C++ uses. There are directives that let you import and export names. Like C/C++, you could place these directives directly in the related source modules. However, such code is difficult to maintain (since you need to change the directives in every file whenever you modify a public name). The solution, as adopted in the HLA programming language, is to use *header files*. Header files contain all the public definitions and exports (as well as common data type definitions and constant definitions). The header file provides the *interface* to the other modules that want to use the code present in the implementation module.

The HLA EXTERNAL attribute is perfect for creating interface/header files. When you use EXTERNAL within a source module that defines a symbol, EXTERNAL behaves like a *public* directive, exporting the name to other modules. When you use EXTERNAL within a source modules that refers to an external name, EXTERNAL declares the object to be supplied in a different module. This lets you place an EXTERNAL declaration of an object in a single header file and include this file into both the modules that import and export the public names.

Rule: Keep all module interface directives (EXTERNAL) in a single header file for a given module. Place any other common data type definitions and constant definitions in this header file as well.

Guideline: There should only be a single header file associated with any one module (even if the module has multiple source files associated with it). If, for some reason, you feel it is necessary to have multiple header files associated with a module, you should create a single file that includes all of the other interface files. That way a program that wants to use all the header files need only include the single file.

When designing header files, make sure you can include a file more than once without ill effects (e.g., duplicate symbol errors). The traditional way to do this is to put a #IF statement like the following around all the statements in a header file:

```
; Module: MyHeader.hhf

#if( @defined( MyHeader_hhf ) )
?MyHeader_hhf:=true; // Actual type and value doesn’t really matter.
                .
                .       ;Statements in this header file.
                .
#endif
```

The first time a source file includes "MyHeader.hhf" the symbol "MyHeader_hhf" is undefined. Therefore, the assembler will process all the statements in the header file. In successive include operations (during the same assembly) the symbol "MyHeader_hhf" is already defined, so the assembler ignores the body of the include file.

My would you ever include a file twice? Easy. Some header files may include other header files. By including the file "YourHeader.hhf" a module might also be including "MyHeader.hhf" (assuming "Your-

Header.hhf" contains the appropriate include directive). Your main program, that includes "YourHeader.hhf" might also need "MyHeader.hhf" so it explicitly includes this file not realizing "YourHeader.hhf" has already processed "MyHeader.hhf" thereby causing symbol redefinitions.

Rule:	Always put an appropriate #IF statement around all the definitions in a header file to allow multiple inclusion of the header file without ill effect.
Guideline:	Use the ".hhf" suffix for HLA header/interface files.
Rule:	Include files for library functions on a system should exist as ".hhf" files and should appear in the "\include" or "\hla\include" subdirectory.
Guideline:	"\hla\include" is probably a better choice if you're using multiple languages since those other languages may need to put files in a "\include" directory.
Exception:	It's probably reasonable to leave the HLA Standard Library's "stdlib.hhf" file in the "\hla\include" directory since most people expect it there.

You can also prevent multiple inclusion of a file by using the #INCLUDEONCE directive. However, it's safer to use the #IF..#ENDIF approach since that doesn't rely on the user of your include file to use the right directive.

C.4 Program Unit Organization

A program unit is any procedure, function, coroutine, iterator, subroutine, subprogram, routine, or other term that describes a section of code that abstracts a set of common operations on the computer. This text will simply use the term *procedure* or *routine* to describe these concepts.

Routines are closely related to modules, since they tend to be the major component of a module (along with data, constants, and types). Hence, many of the attributes that apply to a module also apply to routines. The following paragraphs, at the expense of being redundant, repeat the earlier definitions so you don't have to flip back to the previous sections.

C.4.1 Routine Cohesion

Routines exhibit the following kinds of *cohesion* (listed from good to bad and are mostly identical to the kinds of cohesion that modules exhibit):

- Functional or logical cohesion exists if the routine accomplishes exactly one (simple) task.
- Sequential or *pipelined* cohesion exists when a routine does several sequential operations that must be performed in a certain order with the data from one operation being fed to the next in a "filter-like" fashion.
- Global or *communicational* cohesion exists when a routine performs a set of operations that make use of a common set of data, but are otherwise unrelated.
- Temporal cohesion exists when a routine performs a set of operations that need to be done at the same time (though not necessarily in the same order). A typical initialization routine is an example of such code.
- Procedural cohesion exists when a routine performs a sequence of operations in a specific order, but the only thing that binds them together is the order in which they must be done. Unlike sequential cohesion, the operations do not share data.
- State cohesion occurs when several different (unrelated) operations appear in the same routine and a state variable (e.g., a parameter) selects the operation to execute. Typically such routines contain a case (switch) or **if..elseif..elseif...** statement.
- No cohesion exists if the operations in a routine have no apparent relationship with one another.

The first three forms of cohesion above are generally acceptable in a program. The fourth (temporal) is probably okay, but you should rarely use it. The last three forms should almost never appear in a program. For some reasonable examples of routine cohesion, you should consult "Code Complete".

Guideline: All routines should exhibit good cohesiveness. Functional cohesiveness is best, followed by sequential and global cohesiveness. Temporal cohesiveness is okay on occasion. You should avoid the other forms.

C.4.2 Routine Coupling

Coupling refers to the way that two routines communicate with one another. There are several criteria that define the level of coupling between two routines; again these are identical to the types of coupling that modules exhibit:

- Cardinality- the number of objects communicated between two routines. The fewer objects the better (i.e., fewer parameters).
- Intimacy- how "private" is the communication? Parameter lists are the most private form; private data fields in a class or object are next level; public data fields in a class or object are next, global variables are even less intimate, and passing data in a file or database is the least intimate connection. Well-written routines exhibit a high degree of intimacy.
- Visibility- this is somewhat related to intimacy above. This refers to how visible the data is to the entire system that you pass between two routines. For example, passing data in a parameter list is direct and very visible (you always see the data the caller is passing in the call to the routine); passing data in global variables makes the transfer less visible (you could have set up the global variable long before the call to the routine). Another example is passing simple (scalar) variables rather than loading up a bunch of values into a structure/record and passing that structure/record to the callee.
- Flexibility- This refers to how easy it is to make the connection between two routines that may not have been originally intended to call one another. For example, suppose you pass a structure containing three fields into a function. If you want to call that function but you only have three data objects, not the structure, you would have to create a dummy structure, copy the three values into the field of that structure, and then call the routine. On the other hand, had you simply passed the three values as separate parameters, you could still pass in structures (by specifying each field) as well as call the routine with separate values.

A function is *loosely coupled* if it exhibits low cardinality, high intimacy, high visibility, and high flexibility. Often, these features are in conflict with one another (e.g., increasing the flexibility by breaking out the fields from a structures [a good thing] will also increase the cardinality [a bad thing]). It is the traditional goal of any engineer to choose the appropriate compromises for each individual circumstance; therefore, you will need to carefully balance each of the four attributes above.

A program that uses loose coupling generally contains fewer errors per KLOC (thousands of lines of code). Furthermore, routines that exhibit loose coupling are easier to reuse (both in the current and future projects). For more information on coupling, see the appropriate chapter in "Code Complete".

Guideline: Coupling between routines in source code should be loose.

C.4.3 Routine Size

Sometime in the 1960's, someone decided that programmers could only look at one page in a listing at a time, therefore routines should be a maximum of one page long (66 lines, at the time). In the 1970's, when interactive computing became popular, this was adjusted to 24 lines -- the size of a terminal screen. In fact, there is very little empirical evidence to suggest that small routine size is a good attribute. In fact, several studies on code containing artificial constraints on routine size indicate just the opposite -- shorter routines often contain more bugs per KLOC[6].

6. This happens because shorter functions invariably have stronger coupling, leading to integration errors.

A routine that exhibits functional cohesiveness is the right size, almost regardless of the number of lines of code it contains. You shouldn't artificially break up a routine into two or more subroutines (e.g., sub_partI and sub_partII) just because you feel a routine is getting to be too long. First, verify that your routine exhibits strong cohesion and loose coupling. If this is the case, the routine is not too long. Do keep in mind, however, that a long routine is probably a good indication that it is performing several actions and, therefore, does not exhibit strong cohesion.

Of course, you can take this too far. Most studies on the subject indicate that routines in excess of 150-200 lines of code tend to contain more bugs and are more costly to fix than shorter routines. Note, by the way, that you do not count blank lines or lines containing only comments when counting the lines of code in a program.

Also note that most studies involving routine size deal with HLLs. A comparable HLA routine will contain more lines of code than the corresponding HLL routine. Therefore, you can expect your routines in assembly language to be a little longer.

Guideline: Do not let artificial constraints affect the size of your routines. If a routine exceeds about 200-250 lines of code, make sure the routine exhibits functional or sequential cohesion. Also look to see if there aren't some generic subsequences in your code that you can turn into stand alone routines.

Rule: Never shorten a routine by dividing it into *n* parts that you would always call in the appropriate sequence as a way of shortening the original routine.

C.5 Statement Organization

In an assembly language program, the author must work extra hard to make a program readable. By following a large number of rules, you can produce a program that is readable. However, by breaking a single rule *no matter how many other rules you've followed*, you can render a program unreadable. Nowhere is this more true than how you organize the statements within your program.

C.5.1 Writing "Pure" Assembly Code

Consider the following example taken from "The Art of Assembly Language Programming/DOS Edition" and converted to HLA:

> The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order). Any number of spaces or tabs can separate the various fields in the statement. To the assembler, the following two code sequences are identical:

```
                    mov( 0, ax );
                    mov( ax, bx );
                    add( dx, ax );
                    mov( ax, cx );
```

```
mov( 0,                 ax);
           mov( ax,                   bx);
    add( ad,               ax);
                   mov(                      ax, cx );
```

The first code sequence is much easier to read than the second (if you don't think so, perhaps you should go see a doctor!). With respect to readability, the judicial use of spacing within your program can make all the difference in the world.

While this is an extreme example, do note that it only takes a few mistakes to have a large impact on the readability of a program.

HLA is a free-form assembler insofar as it does not place stringent formatting requirements on its statements. For example, you can put multiple statements on a single line as well as spread a single statement across multiple lines. However, the freedom to arrange these statements in any manner is one of the primary contributors to hard to read assembly language programs. Although HLA lets you enter your programs in free-form, there is absolutely no reason you cannot adopt a fixed format. Doing so generally helps make an assembly language program much easier to read. Here are the rules you should use:

Guideline: Only place one statement per source line.

Rule: Within a given block of code, all mnemonics should start in the same column.

Exception: See the indentation rules appearing later in this documentation.

Guideline: Try to always start the comment fields on adjacent source lines in the same column (note that it is impractical to always start the comment field in the same column throughout a program).

Most people learn a high level language prior to learning assembly language. They have been firmly taught that readable (HLL) programs have their control structures properly indented to show the structure of the program. Indentation works great when you have a *block structured* language. In old-fashioned assembly language this scheme doesn't work; one of the principle benefits to HLA is that it lets you continue to use the indentation schemes you're familiar with in HLLs like C/C++ and Pascal. However, this assumes that you're using the HLA high level control structures. If you choose to work in "pure" assembly language, then these rules don't apply. The following discussion assumes the use of "pure" assembly language code; we'll address HLA's high level control statements later.

If you need to set off a sequence of statements from surrounding code, the best thing you can do is use blank lines in your source code. For a small amount of detachment, to separate one computation from another for example, a single blank line is sufficient. To really show that one section of code is special, use two, three, or even four blank lines to separate one block of statements from the surrounding code. To separate two totally unrelated sections of code, you might use several blank lines and a row of dashes or asterisks to separate the statements. E.g.,

```
mov( FileSpec, eax );
mov( 0, cl );
call MyFunction;
jc Error;

//**********************************************

mov( &fileRecords, edi );
mov( &files, ebx );
sub( 2, ebx );
```

Guideline: Use blank lines to separate special blocks of code from the surrounding code. Use an aesthetic looking row of asterisks or dashes if you need a stronger separation between two blocks of code (do not overdo this, however).

If two sequences of assembly language statements correspond to roughly two HLL statements, it's generally a good idea to put a blank line between the two sequences. This helps clarify the two segments of code in the reader's mind. Of course, it is easy to get carried away and insert too much white space in a program, so use some common sense here.

Guideline: If two sequences of code in assembly language correspond to two adjacent statements in a HLL, then use a blank line to separate those two assembly sequences (assuming the sequences are real short).

A common problem in any language (not just assembly language) is a line containing a comment that is adjacent to one or two lines containing code. Such a program is very difficult read because it is hard to determine where the code ends and the comment begins (or vice-versa). This is especially true when the comments contain sample code. It is often quite difficult to determine if what you're looking at is code or comments; hence the following enforced rule:

Enforced Rule: Always put at least one blank line between code and comments (assuming, of course, the comment is sitting only a line by itself; that is, it is not an endline comment[7]).

C.5.2 Using HLA's High Level Control Statements

Since HLA's high level control statements are so similar to high level language control statements, it's not surprising to discover that you'll use the same formatting for HLA's statements as you would with those other HLLs. Most of these statements compile to very efficient machine code (usually matching what you'd write yourself if you were writing "pure" assembly code). Since their use can make your programs more readable, you should use them whenever practical.

Guideline: Use the HLA high level control structures when they are appropriate in your programs.

There are two problems advanced assembly programmers have with high level control structures: (1) the compiler for such statements (e.g., HLA) doesn't always generate the best code, and (2) the use of such statements encourages inefficient coding on the programmer's part.

HLA's control structures are relatively limited, so point (1) above isn't as big a problem as you might expect. Nevertheless, there will certainly be situations where HLA does not generate the same exact instruction sequence you would for a given control construct. Therefore, it's a good idea to become familiar with the low-level code that HLA emits for each of the control structures so that you can intelligently choose whether to use a high level or low level control structure in a given situation. A later appendix explains how HLA generates code for the high level control structures; you should study this material. Also note that HLA emits MASM compatible assembly code, so you can certainly study HLA's output if you've got any questions about the code HLA generates.

Point (2) above is something that HLA has no control over. It is quite true that if you write "C code with MOV instructions" in HLA, the code probably isn't going to be as efficient as pure assembly code. However, with a little discipline you can prevent this problem from occurring.

One of the benefits to using the high level control structures HLA provides is that you can now use indentation of your statements to better show the structure of the program. Since HLA's high level control structures are very similar to those found in traditional high level languages, you can use well-established programming conventions when indenting statements in your HLA programs. Here are some suggestions:

Rule: Indent statements within a high-level control block four space. The ENDxxxx clause that matches the statement should begin in the same column as the statement that starts a block.

```
// Example of nesting an IF..THEN..ENDIF statement:

        if( eax = 0 ) then

            << Indent these statements four spaces >>

        endif;  // endif should be at the same level as the if statement.
```

Guideline: Avoid putting multiple statements on the same line.

7. See the next section concerning comments for more information.

The HLA programming language contains eight flow-of-control statements: two conditional selection statements (IF..THEN..ELSEIF..ELSE and SWITCH..CASE..DEFAULT..ENDSWITCH), five loops (WHILE..ENDWHILE, REPEAT..UNTIL, FOR..ENDFOR, FOREACH..ENDFOR, and FOREVER..ENDFOR), a program unit invocation (i.e., procedure call), and the statement sequence.

Rule: If your code contains a chain of if..elseif..elseif.......elseif..... statements, do not use the final else clause to handle a remaining case. Only use the final else to catch an error condition. If you need to test for some value in an if..elseif..elseif.... chain, always test the value in an if or elseif statement.

The HLA Standard Library implements the multi-way selection statements (SWITCH) using a jump table. This means that the order of the cases within the selection statement is usually irrelevant. Placing the statements in a particular order rarely improves performance. Since the order is usually irrelevant to the compiler, you should organize the cases so that they are easy to read. There are two common organizations that make sense: sorted (numerically or alphabetically) or by frequency (the most common cases first). Either organization is readable; one drawback to this approach is that it is often difficult to predict which cases the program will execute most often.

Guideline: When using multi-way selection statements (case/switch) sort the cases numerically (alphabetically) or by frequency of expected occurrence.

There are three general categories of looping constructs available in common high-level languages-loops that test for termination at the beginning of the loop (e.g., WHILE), loops that test for loop termination at the bottom of the loop (e.g., REPEAT..UNTIL), and those that test for loop termination in the middle of the loop (e.g., FOREVER..ENDFOR). It is possible simulate any one of these loops using any of the others. This is particularly trivial with the FOREVER..ENDFOR construct:

```
/* Test for loop termination at beginning of FOREVER..ENDFOR */

   forever
       breakif( ax = y );
         .
         .
         .
   endfor;


/* Test for loop termination in the middle of FOREVER..ENDFOR */

   forever
         .
         .
         .
       breakif( ax = y );
         .
         .
         .
    endfor;

/* Test for loop termination at the end of FOREVER..ENDFOR */

    forever
         .
         .
         .
       breakif( x = y );
    endfor;
```

Given the flexibility of the FOREVER..ENDFOR control structure, you might question why one would even burden a compiler with the other loop statements. However, using the appropriate looping structure makes a program far more readable, therefore, you should never use one type of loop when the situation demands another. If someone reading your code sees a FOREVER..ENDFOR construct, they may think it's okay to insert statements before or after the exit statement in the loop. If your algorithm truly depends on WHILE..ENDWHILE or REPEAT..UNTIL semantics, the program may now malfunction.

Rule: Always use the most appropriate type of loop (categorized by termination test position). Never force one type of loop to behave like another.

Many languages provide a special case of the while loop that executes some number of times specified upon first encountering the loop (a *definite* loop rather than an *indefinite* loop). This is the "for" loop in most languages. The vast majority of the time a for loop sequences through a fixed range of value incrementing or decrementing the loop control variable by one. Therefore, most programmers automatically assume this is the way a for loop will operate until they take a closer look at the code. Since most programmers immediately expect this behavior, it makes sense to limit FOR loops to these semantics. If some other looping mechanism is desirable, you should use a WHILE loop to implement it (since the for loop is just a special case of the while loop). There are other reasons behind this decision as well.

Rule: "FOR" loops should always use an ordinal loop control variable (e.g., integer, char, boolean, enumerated type) and should always increment or decrement the loop control variable by one.

Most people expect the execution of a loop to begin with the first statement at the top of the loop, therefore,

Rule: All loops should have one entry point. The program should enter the loop with the instruction at the top of the loop.

Likewise, most people expect a loop to have a single exit point, especially if it's a WHILE or REPEAT..UNTIL loop. They will rarely look closely inside a loop body to determine if there are "break" statements within the loop once they find one exit point. Therefore,

Guideline: Loops with a single exit point are more easily understood.

Whenever a programmer sees an empty loop, the first thought is that something is missing. Therefore,

Guideline: Avoid empty loops. If testing the loop termination condition produces some side effect that is the whole purpose of the loop, move that side effect into the body of the loop. If a loop truly has an empty body, place a comment like "/* nothing */" within your code.

Even if the loop body is not empty, you should avoid side effects in a loop termination expression. When someone else reads your code and sees a loop body, they may skim right over the loop termination expression and start reading the code in the body of the loop. If the (correct) execution of the loop body depends upon the side effect, the reader may become confused since s/he did not notice the side effect earlier. The presence of side effects (that is, having the loop termination expression compute some other value beyond whether the loop should terminate or repeat) indicates that you're probably using the wrong control structure. Consider the following WHILE loop in HLA that is easily corrected:

```
while( mov( stdin.geti32(), ecx ) != 0 ) do

     << statements >>

endwhile;
```

A better implementation of this code fragment would be to use a FOREVER..ENDFOR construct:

```
forever
```

```
        stdin.geti32();
        mov( eax, ecx );
        breakif( eax = 0 );
            .
            .
            .
    endfor;
```

Rule: Avoid side-effects in the computation of the loop termination expression (others may not be expecting such side effects). Also see the guideline about empty loops.

Like functions, loops should exhibit functional cohesion. That is, the loop should accomplish exactly one thing. It's very tempting to initialize two separate arrays in the same loop. You have to ask yourself, though, "what do you really accomplish by this?" You save about four machine instructions on each loop iteration, that's what. That rarely accounts for much. Furthermore, now the operations on those two arrays are tied together, you cannot change the size of one without changing the size of the other. Finally, someone reading your code has to remember two things the loop is doing rather than one.

Guideline: Make each loop perform only one function.

Programs are much easier to read if you read them from left to right, top to bottom (beginning to end). Programs that jump around quite a bit are much harder to read. Of course, the *jmp (goto)* statement is well-known for its ability to scramble the logical flow of a program, but you can produce equally hard to read code using other, structured, statements in a language. For example, a deeply nested set of if statements, some with and some without ELSE clauses, can be very difficult to follow because of the number of possible places the code can transfer depending upon the result of several different boolean expressions.

Rule: Code, as much as possible, should read from top to bottom.

Rule: Related statements should be grouped together and separated from unrelated statements with whitespace or comments.

In theory, a line of source code can be arbitrarily long. In practice, there are several practical limitations on source code lines. Paramount is the amount of text that will fit on a given terminal display device (we don't all have 21" high resolution monitors!) and what can be printed on a typical sheet of paper. Even with small fonts and wide carriage printers, keep in mind that many people like to print listings two-up or three-up in order to save paper. If this isn't enough to suggest an 80 character limit on source lines, McConnell suggests that longer lines are harder to read (remember, people tend to look at only the left side of the page while skimming through a listing).

Enforced Rule: Source code lines will not exceed 80 characters in length.

If a statement approaches the maximum limit of 80 characters, it should be broken up at a reasonable point and split across two lines. If the line is a control statement that involves a particularly long logical expression, the expression should be broken up at a logical point (e.g., at the point of a low-precedence operator outside any parentheses) and the remainder of the expression placed underneath the first part of the expression. E.g., (note that the following involves constant expressions, run-time expressions generally aren't very long):

```
#if
(
        ( ( x + y * z) < ( ComputeProfits(1980,1990) / 1.0775 ) )
    &&  ( ValueOfStock[ ThisYear ] >= ValueOfStock[ LastYear ] )
)
```

```
        << statements >>

#endif
```

Many statements (e.g., IF, WHILE, FOR, and function or procedure calls) contain a keyword followed by a parenthesis. If the expression appearing between the parentheses is too long to fit on one line, consider putting the opening and closing parentheses in the same column as the first character of the start of the statement and indenting the remaining expression elements. The example above demonstrates this for the "IF" statement. The following examples demonstrate this technique for other statements:

```
while
(
       SomeFunctionReturningAValueInEAX( with, lots, of, parameters )
   <=  AFunctionReturningAValueInEBX( also, has, lots, of, parameters )
) do

    << Statements to execute >>

endwhile;

fileio.put
(
   outputFileHandle,
   "Error in module “,
   ModuleName,
   “ at line #”,
   LineNumber,
   “, encountered illegal value",
   nl
);
```

Guideline: For statements that are too long to fit on one physical 80-column line, you should break the statement into two (or more) lines at points in the statement that will have the least impact on the readability of the statement. This situation usually occurs immediately after low-precedence operators or after commas.

If a procedure, function, or other program unit has a particularly long actual or formal parameter list, each parameter should be placed on a separate line. The following examples demonstrate a procedure declaration and call using this technique:

```
procedure MyFunction
(
    NumberOfDataPoints: int32,
    X1Root: real32,
    X2Root: real32,
    var YIntercept: real32
);


MyFunction
(
   GetNumberOfPoints(RootArray),  // Assume “RETURNS” value is EAX.
   RootArray[ EBX*4 ],
   RootArray[ ECX*4 ],
   Solution
);
```

Rule: If an actual or formal parameter list is too long to fit a function call or definition on a single line, then place each parameter on a separate line and align them so they are easy to read.

Guideline: If a boolean expression exceeds the length of the source line (usually 80 characters), then break the source line into pieces and align the parentheses associated with the statement underneath the start of the statement.

This usually isn't a problem in HLA since expressions are very limited. However, if you call a function with a long parameter list you could run into this problem. One area where this problem does occur is when you're using HLA's hybrid control structures. For such sequences you should always place the statements associated with the boolean expression on separate lines and align the braces with the high level control structure, e.g.,

```
if
{
    cmp( ax, bx );;
    jne true;
    cmp( ax, 5 );
    jl false;
    cmp( bx, 0 );
    je false;
}

    << statements to execute on TRUE >>

endif;
```

Rule: Always put a blank line between a high level control statement and the nested statements associated with that statement. Likewise, put a blank line between the end of the nested statements and the corresponding ENDxxx clause of the statement. E.g.,

```
if( ax = 0 ) then
                                    <-- Blank line.
    << Nested Statements >>
                                    <-- Blank line.
endif;
```

HLA provides special symbols like "@c" and "@s" to denote flag bits within boolean expressions. Using statements like "if(@c) then .... endif;" is semantically equivalent to using a conditional jump (JNC in this case) to jump around the THEN code. Not only is this statement semantically equivalent, it is exactly equivalent since it simply generates the JNC (or whatever) instruction to transfer control to the statement following the ENDIF. The difference between the two, from a readability point of view, is that JNC requires a statement label. As it turns out, the large number of statement labels that appear in an assembly language program contribute to the lack of readability. Hence, anything you can do to legitimately reduce the number of statement labels will improve the readability of your program. So,

Guideline: Try to use statements like "if(@c) then...endif;" rather than "jnc label; ... label:" in your programs to reduce the number of statement labels in the code. Combined with indentation, this will make your programs easier to read since the user doesn't have to search for a specific label associated with the branch (searching for the end of indentation is a much easier task).

C.6 Comments

Comments in an assembly language program generally come in two forms: *endline* comments and *standalone* comments[8]. As their names suggest, endline lines comments always occur at the end of a source statement and standalone comments sit on a line by themselves[9]. These two types of comments have distinct purposes, this section will explore their use and describe the attributes of a well-commented program.

C.6.1 What is a Bad Comment?

It is amazing how many programmers claim their code is well-commented. Were you to count characters between (or after) the comment delimiters, they might have a point. Consider, however, the following comment:

```
mov( 0, ax );    //Set AX to zero.
```

Quite frankly, this comment is worse than no comment at all. It doesn't tell the reader anything the instruction itself doesn't tell and it requires the reader to take some of his or her precious time to figure out that the comment is worthless. If someone cannot tell that this instruction is setting AX to zero, they have no business reading an assembly language program. This brings up the first guideline of this section:

Guideline: Choose an intended audience for your source code and write the comments to that audience. For HLA source code, you can usually assume that the target audience are those who know a reasonable amount of HLA and assembly language.

Don't explain the actions of an assembly language instruction in your code unless that instruction is doing something that isn't obvious (and most of the time you should consider changing the code sequence if it isn't obvious what is going on). Instead, explain how that instruction is helping to solve the problem at hand. The following is a much better comment for the instruction above:

```
mov( 0, ax );    //AX is the resulting sum.  Initialize it.
```

Note that the comment does not say "Initialize it to zero." Although there would be nothing intrinsically wrong with saying this, the phrase "Initialize it" remains true no matter what value you assign to AX. This makes maintaining the code (and comment) much easier since you don't have to change the comment whenever you change the constant associated with the instruction.

Guideline: Write your comments in such a way that minor changes to the instruction do not require that you change the corresponding comment.

Note: Although a trivial comment is bad (indeed, worse than no comment at all), the worst comment a program can have is one that is wrong. Consider the following statement:

```
mov( 1, ax );    //Set AX to zero.
```

It is amazing how long a typical person will look at this code trying to figure out how on earth the program sets AX to zero when it's obvious it does not do this. *People will always believe comments over code*. If there is some ambiguity between the comments and the code, they will assume that the code is tricky and that the comments are correct. Only after exhausting all possible options is the average person likely to concede that the comment must be incorrect.

Enforced Rule: Never allow incorrect comments in your program.

This is another reason not to put trivial comments like "Set AX to zero" in your code. As you modify the program, these are the comments most likely to become incorrect as you change the code and fail to keep the comments in sync. However, even some non-trivial comments can become incorrect via changes to the code. Therefore, always follow this rule:

8. This document will simply use the term *comments* when referring to standalone comments.
9. Since the label, mnemonic, and operand fields are all optional, it is legal to have a comment on a line by itself.

Enforced Rule: Always update *all* comments affected by a code change immediately after making the code change.

Undoubtedly you've heard the phrase "make sure you comment your code as though someone else wrote it for you; otherwise in six months you'll wish you had." This statement encompasses two concepts. First, don't ever think that your understanding of the current code will last. While working on a given section of a program you're probably investing considerable thought and study to figure out what's going on. Six months down the road, however, you will have forgotten much of what you figured out and the comments can go a long way to getting you back up to speed quickly. The second point this code makes is the implication that others read and write code too. You will have to read someone else's code, they will have to read yours. If you write the comments the way you would expect others to write it for you, chances are pretty good that your comments will work for them as well.

Rule: Never use racist, sexist, obscene, or other exceptionally politically incorrect language in your comments. Undoubtedly such language in your comments will come back to embarrass you in the future. Furthermore, it's doubtful that such language would help someone better understand the program.

It's much easier to give examples of bad comments than it is to discuss good comments. The following list describes some of the worst possible comments you can put in a program (from worst up to barely tolerable):

- The absolute worst comment you can put into a program is an incorrect comment. Consider the following assembly statement:
  ```
  mov( 10, ax );  // Set AX to 11
  ```
  It is amazing how many programmers will automatically assume the comment is correct and try to figure out how this code manages to set the variable "A" to the value 11 when the code so obviously sets it to 10.
- The second worst comment you can place in a program is a comment that explains what a statement is doing. The typical example is something like "mov(10, ax); // Set 'A' to 10". Unlike the previous example, this comment is correct. But it is still worse than no comment at all because it is redundant and forces the reader to spend additional time reading the code (reading time is directly proportional to reading difficulty). This also makes it harder to maintain since slight changes to the code (e.g., "mov(9, ax);") requires modifications to the comment that would not otherwise be required.
- The third worst comment in a program is an irrelevant one. Telling a joke, for example, may seem cute, but it does little to improve the readability of a program; indeed, it offers a distraction that breaks concentration.
- The fourth worst comment is no comment at all.
- The fifth worst comment is a comment that is obsolete or out of date (though not incorrect). For example, comments at the beginning of the file may describe the current version of a module and who last worked on it. If the last programmer to modify the file did not update the comments, the comments are now out of date.

C.6.2 What is a Good Comment?

Steve McConnell provides a long list of suggestions for high-quality code. These suggestions include:

- **Use commenting styles that don't break down or discourage modification.** Essentially, he's saying pick a commenting style that isn't so much work people refuse to use it. He gives an example of a block of comments surrounded by asterisks as being hard to maintain. This is a poor example since modern text editors will automatically "outline" the comments for you. Nevertheless, the basic idea is sound.
- **Comment as you go along**. If you put commenting off until the last moment, then it seems like another task in the software development process always comes along and management is likely to discourage the completion of the commenting task in hopes of meeting new deadlines.

- **Avoid self-indulgent comments.** Also, you should avoid sexist, profane, or other insulting remarks in your comments. Always remember, someone else will eventually read your code.
- **Avoid putting comments on the same physical line as the statement they describe.** Such comments are very hard to maintain since there is very little room. McConnell suggests that endline comments are okay for variable declarations. For some this might be true but many variable declarations may require considerable explanation that simply won't fit at the end of a line. One exception to this rule is "maintenance notes." Comments that refer to a defect tracking entry in the defect database are okay (note that the CodeWright text editor provides a much better solution for this -- buttons that can bring up an external file). Of course, endline comments are marginally more useful in assembly language than in the HLLs that McConnell addresses, but the basic idea is sound.
- **Write comments that describe blocks of statements rather than individual statements.** Comments covering single statements tend to discuss the mechanics of that statement rather than discussing what the program is doing.
- **Focus paragraph comments on the *why* rather than the *how.*** Code should explain what the program is doing and why the programmer chose to do it that way rather than explain what each individual statement is doing.
- **Use comments to prepare the reader for what is to follow.** Someone reading the comments should be able to have a good idea of what the following code does without actually looking at the code. Note that this rule also suggests that comments should always precede the code to which they apply.
- **Make every comment count.** If the reader wastes time reading a comment of little value, the program is harder to read; period.
- **Document surprises and tricky code.** Of course, the best solution is not to have any tricky code. In practice, you can't always achieve this goal. When you do need to restore to some tricky code, make sure you fully document what you've done.
- **Avoid abbreviations.** While there may be an argument for abbreviating identifiers that appear in a program, no way does this apply to comments.
- **Keep comments close to the code they describe.** The prologue to a program unit should give its name, describe the parameters, and provide a short description of the program. It should not go into details about the operation of the module itself. Internal comments should to that.
- **Comments should explain the parameters to a function**, assertions about these parameters, whether they are input, output, or in/out parameters.
- **Comments should describe a routine's limitations, assumptions, and any side effects**.

Rule: All comments will be high-quality comments that describe the actions of the surrounding code in a concise manner

C.6.3 Endline vs. Standalone Comments

Guideline: Adjacent lines of comments should not have any interspersed blank lines. At least lead off the comment with the HLA comment character sequence (e.g., "//").

The guideline above suggests that your code should look like this:

```
// This is a comment with a blank line between it and the next comment.
//
// This is another line with a comment on it.
```

Rather than like this:

```
// This is a comment with a blank line between it and the next comment.

// This is another line with a comment on it.
```

The “//” appearing between the two statements suggest continuity that is not present when you remove the “//”. If two blocks of comments are truly separate and whitespace between them is appropriate, you should consider separating them by a large number of blank lines to completely eliminate any possible association between the two.

Standalone comments are great for describing the actions of the code that immediately follows. So what are endline comments useful for? Endline comments can explain how a sequence of instructions are implementing the algorithm described in a previous set of standalone comments. Consider the following code:

```
// Compute the transpose of a matrix using the algorithm:
//
//      for i := 0 to 3 do
//              for j := 0 to 3 do
//                      swap( a[i][j], b[j][i] );

    for( mov( 0, i); i < 3; inc( i )) do

        for( mov( 0, j ); j < 3; inc( j )) do

            mov( i, ebx );      // Compute address of a[i][j] using
            shl( 2, ebx );      // row major ordering (i*4 + j)*4.
            add( j, ebx );
            lea( ebx, a[ebx*4] );
            push( ebx );        // Push address of a[i][j] onto stack.

            mov( j, ebx );      // Compute address of b[j][i] using
            shl( 2, ebx );      // row major ordering (j*4 + i)*4.
            add( i, ebx );
            lea( ebx, b[ebx*4] );
            push( ebx );        // Push address of b[j][i] onto stack.
            call swap;          // Swap objects pointed at by [esp] and [esp+4].

        endfor;

    endfor;
```

Note that the block comments before this sequence explain, in high level terms, what the code is doing. The endline comments explain how the statement sequence implements the general algorithm. Note, however, that the endline comments do not explain what each statement is doing (at least at the machine level). Rather than claiming "`lea( ebx, b[ebx*4] )`" also multiplies the quantity in EBX by four, this code assumes the reader can figure that out for themselves (any reasonable assembly programmer would know this). Once again, keep in mind your audience and write your comments for them.

C.6.4 Unfinished Code

Often it is the case that a programmer will write a section of code that (partially) accomplishes some task but needs further work to complete a feature set, make it more robust, or remove some known defect in the code. It is common for such programmers to place comments into the code like "This needs more work," "Kludge ahead," etc. The problem with these comments is that they are often forgotten. It isn’t until the code fails in the field that the section of code associated with these comments is found and their problems corrected.

Ideally, one should never have to put such code into a program. Of course, ideally, programs never have any defects in them, either. Since such code inevitably finds its way into a program, it’s best to have a policy in place to deal with it, hence this section.

Unfinished code comes in five general categories: non-functional code, partially functioning code, suspect code, code in need of enhancement, and code documentation. Non-functional code might be a stub or driver that needs to be replaced in the future with actual code or some code that has severe enough defects that it is useless except for some small special cases. This code is really bad, fortunately its severity prevents you from ignoring it. It is unlikely anyone would miss such a poorly constructed piece of code in early testing prior to release.

Partially functioning code is, perhaps, the biggest problem. This code works well enough to pass some simple tests yet contains serious defects that should be corrected. Moreover, these defects are known. Software often contains a large number of unknown defects; it's a shame to let some (prior) known defects ship with the product simply because a programmer forgot about a defect or couldn't find the defect later.

Suspect code is exactly that- code that is suspicious. The programmer may not be aware of a quantifiable problem but may suspect that a problem exists. Such code will need a later review in order to verify whether it is correct.

The fourth category, code in need of enhancement, is the least serious. For example, to expedite a release, a programmer might choose to use a simple algorithm rather than a complex, faster algorithm. S/he could make a comment in the code like "This linear search should be replaced by a hash table lookup in a future version of the software." Although it might not be absolutely necessary to correct such a problem, it would be nice to know about such problems so they can be dealt with in the future.

The fifth category, documentation, refers to changes made to software that will affect the corresponding documentation (user guide, design document, etc.). The documentation department can search for these defects to bring existing documentation in line with the current code.

This standard defines a mechanism for dealing with these five classes of problems. Any occurrence of unfinished code will be preceded by a comment that takes one of the following forms (where "_" denotes a single space):

```
//_#defect#severe_//
//_#defect#functional_//
//_#defect#suspect_//
//_#defect#enhancement_//
//_#defect#documentation_//
```

It is important to use all lower case and verify the correct spelling so it is easy to find these comments using a text editor search or a tool like grep. Obviously, a separate comment explaining the situation must follow these comments in the source code.

Examples:

```
// #defect#suspect //
// #defect#enhancement //
// #defect#documentation //
```

Notice the use of comment delimiters (the "//") on both sides even though HLA doesn't require them.

Enforced Rule: If a module contains some defects that cannot be immediately removed because of time or other constraints, the program will insert a standardized comment before the code so that it is easy to locate such problems in the future. The five standardized comments are "//_#defect#severe_//", "//_#defect#functional_//", "//_#defect#suspect_//", "//_#defect#enhancement_//", and "//_#defect#documentation_//" where "_" denotes a single space. The spelling and spacing should be exact so it is easy to search for these strings in the source tree.

C.6.5 Cross References in Code to Other Documents

In many instances a section of code might be intrinsically tied to some other document. For example, you might refer the reader to the user document or the design document within your comments in a program. This document proposes a standard way to do this so that it is relatively easy to locate cross references appearing in source code. The technique is similar to that for defect reporting, except the comments take the form:

```
        //  text #link#location text //
```

"*Text*" is optional and represents arbitrary text (although it is really intended for embedding html commands to provide hyperlinks to the specified document). "*Location*" describes the document and section where the associated information can be found.

```
Examples:

// #link#User's Guide Section 3.1 //
// #link#Program Design Document, Page 5 //
// #link#Funcs.pas module, "xyz" function //
// <A HREF="DesignDoc.html#xyzfunc"> #link#xyzfunc </a> //
```

Guideline: If a module contains some cross references to other documents, there should be a comment that takes the form "// *text* #link#*location text* //" that provides the reference to that other document. In this comment "*text*" represents some optional text (typically reserved for html tags) and "location" is some descriptive text that describes the document (and a position in that document) related to the current section of code in the program.

C.7 Names, Instructions, Operators, and Operands

Although program features like good comments, proper spacing of statements, and good modularization can help yield programs that are more readable; ultimately, a programmer must read the instructions in a program to understand what it does. Therefore, do not underestimate the importance of making your statements as readable as possible. This section deals with this issue.

C.7.1 Names

According to studies done at IBM, the use of high-quality identifiers in a program contributes more to the readability of that program than any other single factor, including high-quality comments. The quality of your identifiers can make or break your program; program with high-quality identifiers can be very easy to read, programs with poor quality identifiers will be very difficult to read. There are very few “tricks” to developing high-quality names; most of the rules are nothing more than plain old-fashion common sense. Unfortunately, programmers (especially C/C++ programmers) have developed many arcane naming conventions that ignore common sense. The biggest obstacle most programmers have to learning how to create good names is an unwillingness to abandon existing conventions. Yet their only defense when quizzed on why they adhere to (existing) bad conventions seems to be “because that’s the way I’ve always done it and that’s the way everybody else does it.”

The aforementioned researchers at IBM developed several programs with the following set of attributes:

- Bad comments, bad names
- Bad comments, good names
- Good comments, bad names
- Good comments, good names

As should be obvious, the programs that had bad comments and names were the hardest to read; likewise, those programs with good comments and names were the easiest to read. The surprising results concerned the other two cases. Most people assume good comments are more important than good names in a program. Not only did IBM find this to be false, they found it to be *really* false.

As it turns out, good names are even more important that good comments in a program. This is not to say that comments are unimportant, they are extremely important; however, it is worth pointing out that if you spend the time to write good comments and then choose poor names for your program's identifiers, you've damaged the readability of your program despite the work you've put into your comments. Quickly read over the following code:

```
mov( SignedValue, ax );
cwd();
add( -1, ax );
rcl( 1, dx );
mov( dx, AbsoluteValue );
```

Question: What does this code compute and store in the AbsoluteValue variable?

- The sign extension of SignedValue.
- The negation of SignedValue.
- The absolute value of SignedValue.
- A boolean value indicating that the result is positive or negative.
- Signum(SignedValue) (-1, 0, +1 if neg, zero, pos).
- Ceil(SignedValue)
- Floor(SignedValue)

The obvious answer is the absolute value of SignedValue. This is also incorrect. The correct answer is signum:

```
mov( SignedValue, ax ); // Get value to check.
cwd();                  // DX = FFFF if neg, 0000 otherwise.
add( $ffff, ax );       // Carry=0 if ax is zero, one otherwise.
rcl( 1, dx );           // DX = FFFF if AX is neg, 0 if ax=0,
mov( dx, Signum );      //  1 if ax>0.
```

Granted, this is a tricky piece of code[10]. Nonetheless, even without the comments you can probably figure out what the code sequence does even if you can't figure out how it does it:

```
mov( SignedValue, ax );
cwd();
add( $ffff, ax );
rcl( 1, dx );
mov( dx, Signum );
```

Based on the names alone you can probably figure out that this code computes the signum function (even if understanding *how* it does it remains a mystery). This is the "understanding 80% of the code" referred to earlier. Note that you don't need misleading names to make this code unfathomable. Consider the following code that doesn't trick you by using misleading names:

```
mov( x, ax );
cwd();
add( $ffff, ax );
rcl( 1, dx );
mov( dx, y );
```

This is a very simple example. Now imagine a large program that has many names. As the number of names increase in a program, it becomes harder to keep track of them all. If the names themselves do not provide a good clue to the meaning of the name, understanding the program becomes very difficult.

Enforced Rule: All identifiers appearing in an assembly language program must be descriptive names whose meaning and use are clear.

10. It could be worse, you should see what the "superoptimizer" outputs for the signum function. It's even shorter and harder to understand than this code.

Since labels (i.e., identifiers) are the target of jump and call instructions, a typical assembly language program may have a large number of identifiers, especially if you write in “pure” assembly and forego the HLA high level control structures. Therefore, it is tempting to begin using names like "label1, label2, label3, ..." Avoid this temptation! There is always a reason you are jumping to some spot in your code. Try to describe that reason and use that description for your label name.

Rule: Never use names like "Lbl0, Lbl1, Lbl2, ..." in your program. Always use meaningful names!

C.7.1.1 Naming Conventions

Naming conventions represent one area in Computer Science where there are far too many divergent views (program layout is the other principle area). The primary purpose of an object’s name in a programming language is to describe the use and/or contents of that object. A secondary consideration may be to describe the type of the object. Programmers use different mechanisms to handle these objectives. Unfortunately, there are far too many “conventions” in place, it would be asking too much to expect any one programmer to follow several different standards. Therefore, this standard will apply across all languages as much as possible.

The vast majority of programmers know only one language - English. Some programmers know English as a second language and may not be familiar with a common non-English phrase that is not in their own language (e.g., rendezvous). Since English is the common language of most programmers, all identifiers should use easily recognizable English words and phrases.

Rule: All identifiers that represent words or phrases must be English words or phrases.

C.7.1.2 Alphabetic Case Considerations

A case-neutral identifier will work properly whether you compile it with a compiler that has case sensitive identifiers or case insensitive identifiers. In practice, this means that all uses of the identifiers must be spelled exactly the same way (including case) *and* that no other identifier exists whose only difference is the case of the letters in the identifier. For example, if you declare an identifier “ProfitsThisYear” in Pascal (a case-insensitive language), you could legally refer to this variable as “profitsThisYear” and “PROFITSTHISYEAR”. However, this is not a case-neutral usage since a case sensitive language would treat these three identifiers as different names. Conversely, in case-sensitive languages like C/C++, it is possible to create two different identifiers with names like “PROFITS” and “profits” in the program. This is not case-neutral since attempting to use these two identifiers in a case insensitive language (like Pascal) would produce an error since the case-insensitive language would think they were the same name.

Enforced Rule: All identifiers must be “case-neutral.”

Fortunately, HLA enforces case neutrality in its identifiers; so HLA doesn’t allow you to violate this rule. However, if you are linking assembly and high level language code together, it’s a good idea to follow this rule in the HLL code to prevent problems when linking with the HLA code.

Different programmers (especially in different languages) use alphabetic case to denote different objects. For example, a common C/C++ coding convention is to use all upper case to denote a constant, macro, or type definition and to use all lower case to denote variable names or reserved words. Prolog programmers use an initial lower case alphabetic to denote a variable. Other comparable coding conventions exist. Unfortunately, there are so many different conventions that make use of alphabetic case, they are nearly worthless, hence the following rule:

Rule: You should never use alphabetic case to denote the type, classification, or any other program-related attribute of an identifier.

There are going to be some obvious exceptions to the above rule, this document will cover those exceptions a little later. Alphabetic case does have one very useful purpose in identifiers - it is useful for separating words in a multi-word identifier; more on that subject in a moment.

To produce readable identifiers often requires a multi-word phrase. Natural languages typically use spaces to separate words; we can not, however, use this technique in identifiers.

Unfortunatelywritingmultiwordidentifiers makesthemalmostimpossibletoreadifyoudonotdosomethingtodistiguishtheindividualwords (Unfortunately writing multiword identifiers makes them almost impossible to read if you do not do something to distinguish the individual words).

There are a couple of good conventions in place to solve this problem. This standard's convention is to capitalize the first alphabetic character of each word in the middle of an identifier.

Rule: Capitalize the first letter of interior words in all multi-word identifiers.

Note that the rule above does not specify whether the first letter of an identifier is upper or lower case. Subject to the other rules governing case, you can elect to use upper or lower case for the first symbol, although you should be consistent throughout your program. The second convention is to use an underscore to separate words in a multi-word document. This is also acceptable, though the capitalization rule probably produces identifiers that are easier to read and write.

Lower case characters are easier to read than upper case. Identifiers written completely in upper case take almost twice as long to recognize and, therefore, impair the readability of a program. Yes, all upper case does make an identifier stand out. Such emphasis is rarely necessary in real programs. Yes, common C/C++ coding conventions dictate the use of all upper case identifiers. Forget them. They not only make your programs harder to read, they also violate the first rule above.

Rule: Avoid using all upper case characters in an identifier.

Some programmers prefer to begin all identifiers with a lower case letter. Others prefer to begin them with an upper case alphabetic character. Either scheme is fine as long as you apply it consistently throughout your program. Under no circumstances should you use the presence of an upper or lower case character to denote different things in your code (see the earlier rule about this).

C.7.1.3 Abbreviations

The primary purpose of an identifier is to describe the use of, or value associated with, that identifier. The best way to create an identifier for an object is to describe that object in English and then create a variable name from that description. Variable names should be meaningful, concise, and non-ambiguous to an average programmer fluent in the English language. Avoid short names. Some research has shown that programs using identifiers whose average length is 10-20 characters are generally easier to debug than programs with substantially shorter or longer identifiers.

Avoid abbreviations as much as possible. What may seem like a perfectly reasonable abbreviation to you may totally confound someone else. Consider the following variable names that have actually appeared in commercial software:

NoEmployees, NoAccounts, pend

The "NoEmployees" and "NoAccounts" variables seem to be boolean variables indicating the presence or absence of employees and accounts. In fact, this particular programmer was using the (perfectly reasonable in the real world) abbreviation of "number" to indicate the number of employees and the number of accounts. The "pend" name referred to a procedure's end rather than any pending operation.

Programmers often use abbreviations in two situations: they're poor typists and they want to reduce the typing effort, or a good descriptive name for an object is simply too long. The former case is an unacceptable reason for using abbreviations. The second case, especially if care is taken, may warrant the occasional use of an abbreviation.

Guideline: Avoid all identifier abbreviations in your programs. When necessary, use standardized abbreviations or ask someone to review your abbreviations. Whenever you use abbreviations in your programs, create a "data dictionary" in the comments near the names' definition that provides a full name and description for your abbreviation.

The variable names you create should be pronounceable. "NumFiles" is a much better identifier than "NmFls". The first can be spoken, the second you must generally spell out. Avoid homonyms and long names that are identical except for a few syllables. If you choose good names for your identifiers, you should be able to read a program listing over the telephone to a peer without overly confusing that person.

Rule: All identifiers should be pronounceable (in English) without having to spell out more than one letter.

C.7.1.4 The Position of Components Within an Identifier

When scanning through a listing, most programmers only read the first few characters of an identifier. It is important, therefore, to place the most important information (that defines and makes this identifier unique) in the first few characters of the identifier. So, you should avoid creating several identifiers that all begin with the same phrase or sequence of characters since this will force the programmer to mentally process additional characters in the identifier while reading the listing. Since this slows the reader down, it makes the program harder to read.

Guideline: Try to make most identifiers unique in the first few character positions of the identifier. This makes the program easier to read.

Corollary: Never use a numeric suffix to differentiate two names.

Many C/C++ Programmers, especially Microsoft Windows programmers, have adopted a formal naming convention known as "Hungarian Notation." To quote Steve McConnell from Code Complete: "The term 'Hungarian' refers both to the fact that names that follow the convention look like words in a foreign language and to the fact that the creator of the convention, Charles Simonyi, is originally from Hungary." One of the first rules given concerning identifiers stated that all identifiers are to be English names. Do we really want to create "artificially foreign" identifiers? Hungarian notation actually violates another rule as well: names using the Hungarian notation generally have very common prefixes, thus making them harder to read.

Hungarian notation does have a few minor advantages, but the disadvantages far outweigh the advantages. The following list from Code Complete and other sources describes what's *wrong* with Hungarian notation:

- Hungarian notation generally defines objects in terms of basic machine types rather than in terms of abstract data types.
- Hungarian notation combines *meaning* with *representation*. One of the primary purposes of high level language is to abstract representation away. For example, if you declare a variable to be of type *integer*, you shouldn't have to change the variable's name just because you changed its type to *real*.
- Hungarian notation encourages lazy, uninformative variable names. Indeed, it is common to find variable names in Windows programs that contain *only* type prefix characters, without an descriptive name attached.
- Hungarian notation prefixes the descriptive name with some type information, thus making it harder for the programming to find the descriptive portion of the name.

Guideline: Avoid using Hungarian notation and any other formal naming convention that attaches low-level type information to the identifier.

Although attaching *machine* type information to an identifier is generally a bad idea, a well thought-out name can successfully associate some high-level type information with the identifier, especially if the name implies the type or the type information appears as a suffix. For example, names like "PencilCount" and "BytesAvailable" suggest integer values. Likewise, names like "IsReady" and "Busy" indicate boolean values. "KeyCode" and "MiddleInitial" suggest character variables. A name like "StopWatchTime" probably indicates a real value. Likewise, "CustomerName" is probably a string variable. Unfortunately, it isn't always possible to choose a great name that describes both the content and type of an object; this is particularly true when the object is an instance (or definition of) some abstract data type. In such instances, some additional text can improve the identifier. Hungarian notation is a raw attempt at this that, unfortunately, fails for a variety of reasons.

A better solution is to use a *suffix phrase* to denote the type or class of an identifier. A common UNIX/C convention, for example, is to apply a "_t" suffix to denote a type name (e.g., size_t, key_t, etc.). This convention succeeds over Hungarian notation for several reasons including (1) the "type phrase" is a suffix and doesn't interfere with reading the name, (2) this particular convention specifies the *class* of the object (const, var, type, function, etc.) rather than a low level *type*, and (3) It certainly makes sense to change the identifier if it's classification changes.

Guideline: If you *want* to differentiate identifiers that are constants, type definitions, and variable names, use the suffixes "_c", "_t", and "_v", respectively (generally, the lack of a suffix denotes a variable).

Rule: The classification suffix should not be the only component that differentiates two identifiers.

Can we apply this suffix idea to variables and avoid the pitfalls? Sometimes. Consider a high level data type "button" corresponding to a button on a Visual BASIC or Delphi form. A variable name like "CancelButton" makes perfect sense. Likewise, labels appearing on a form could use names like "ETWWLabel" and "EditPageLabel". Note that these suffixes still suffer from the fact that a change in type will require that you change the variable's name. However, changes in high level types are far less common than changes in low-level types, so this shouldn't present a big problem.

HLA provides a special operator, "‘" (grave accent) that separates an identifier name from an attached comment. For example, the legal HLA identifier "Hello‘world" is really just "Hello". The characters following the grave accent (to the end of the identifier) are treated as a comment by the compiler. So if you want to attach a comment concerning the variable's type or use to the identifier, you can use this feature in HLA.

GuideLine: If you must attach low level type information to an identifier, use the HLA identifier comment ("‘", grave accent) and append the information to the end of the identifier.

C.7.1.5 Names to Avoid

Avoid using symbols in an identifier that are easily mistaken for other symbols. This includes the sets {"1" (one), "I" (upper case "I"), and "l" (lower case "L")}, {"0" (zero) and "O" (upper case "O")}, {"2" (two) and "Z" (upper case "Z")}, {"5" (five) and "S" (upper case "S")}, and ("6" (six) and "G" (upper case "G")}.

Guideline: Avoid using symbols in identifiers that are easily mistaken for other symbols (see the list above).

Avoid misleading abbreviations and names. For example, FALSE shouldn't be an identifier that stands for "Failed As a Legitimate Software Engineer." Likewise, you shouldn't compute the amount of free memory available to a program and stuff it into the variable "Profits".

Rule: Avoid misleading abbreviations and names.

You should avoid names with similar meanings. For example, if you have two variables "InputLine" and "InputLn" that you use for two separate purposes, you will undoubtedly confuse the two when writing or reading the code. If you can swap the names of the two objects and the program still makes sense, you should rename those identifiers. Note that the names do not have to be similar, only their meanings. "Input-Line" and "LineBuffer" are obviously different but you can still easily confuse them in a program.

Rule: Do not use names with similar meanings for different objects in your programs.

In a similar vein, you should avoid using two or more variables that have different meanings but similar names. For example, if you are writing a teacher's grading program you probably wouldn't want to use the name "NumStudents" to indicate the number of students in the class along with the variable "StudentNum" to hold an individual student's ID number. "NumStudents" and "StudentNum" are too similar.

Rule: Do not use similar names that have different meanings.

Avoid names that sound similar when read aloud, especially out of context. This would include names like "hard" and "heart", "Knew" and "new", etc. Remember the discussion in the section above on abbreviations, you should be able to discuss your problem listing over the telephone with a peer. Names that sound alike make such discussions difficult.

Guideline: Avoid homonyms in identifiers.

Avoid misspelled words in names and avoid names that are commonly misspelled. Most programmers are notoriously bad spellers (look at some of the comments in our own code!). Spelling words correctly is hard enough, remembering how to spell an identifier *incorrectly* is even more difficult. Likewise, if a word is often spelled incorrectly, requiring a programer to spell it correctly on each use is probably asking too much.

Guideline: Avoid misspelled words and names that are often misspelled in identifiers.

If you redefine the name of some library routine in your code, another program will surely confuse your name with the library's version. This is especially true when dealing with standard library routines and APIs.

Enforced Rule: Do not reuse existing standard library routine names in your program unless you are specifically replacing that routine with one that has similar semantics (i.e., don't reuse the name for a different purpose).

Corollary: Use Namespaces to prevent name space pollution!

C.7.1.6 Special Identifers

By convention, HLA programmers use certain identifiers for special purposes. Any identifier beginning with an underscore falls into this category. HLA defines five such conventions. The HLA compiler does not enforce these conventions, but if you violate them you may run into problems. The following paragraphs describe each of these conventions.

HLA reserves for its own use (and the use of the HLA Standard Library) all identifiers that begin and end with a single underscore. You should never define any identifiers in your programs that take this form since your identifiers may conflict with HLA's use. These reserveration effectively reserves an "HLA namespace" of identifiers that the compiler and Standard Library can draw from without fear of breaking any existing code (that follows this convention).

Identifiers that begin and end with two underscores are reserved for use as local symbols in user-defined macros. To avoid conflicts with such symbols (especially in context-free/multi-part macros) you should never use symbols that begin and end with two underscores outside of a macro. Within a macro, you should use the convention for all symbols that are local to that macro.

By convention, HLA programmers reserve all identifiers beginning with two underscores for defining private data in classes, records, and other declaration sections. If you're using a class (or other structure) and some if its identifiers begin with two underscores, this is your hint that these fields are private to that class and subject to change. You should never directly access such fields. When you're defining your own classes, you should employ this convention to warn others when you're defining private data to that class.

Identifiers that begin with a single underscore have two uses. First, some languages and calling conventions (most notably, C) prepend an underscore to all external names. Therefore, it is common to use reserve symbols that begin with a single underscore for external linkage. The second use is closely related to the first – HLA programmers conventionally use identifiers beginning with a single underscore as a *shadow name*. Consider the following linkage to an external procedure written in C:

```
procedure _externalCFunc( parm2:int32; parm1:int32 ); external;
#macro externalCFunc( p1, p2 );

    _externalCFunc( p2, p1 );

#endmacro;
```

The C compiler exports the name "_externalCFunc" for the C function that is actually named "externalCFunc" inside the C code. Of course, inside our HLA code we would like to use the C name, not the exported name. We could easily achieve this using the following external definition:

```
procedure externalCFunc( parm2:int32; parm1:int32 ); external("_externalCFunc");
```

The only catch is that we'd have to always remember to put the parameters in the reverse order (to match C's calling convention). Savvy HLA programmers use a macro to swap the parameters as in the previous example. They use the exported C name as a shadow name of the function and then write the macro that swaps the function's parameters as the real name.

You can use shadow names for all sorts of different purposes, not just for linkage to C functions. For example, the chapter on macros in this text has given examples of function overloading that uses a macro with the overloaded function name and shadow names for the actual functions that implement each of the overloaded calls. The convention is to use a leading underscore on all the shadow function (and other object) names.

C.7.2 Instructions, Directives, and Pseudo-Opcodes

Your choice of assembly language sequences, the instructions themselves, and your choice of directives and pseudo-opcodes can have a big impact on the readability of your programs. The following subsections discuss these problems.

C.7.2.1 Choosing the Best Instruction Sequence

Like any language, you can solve a given problem using a wide variety of solutions involving different instruction sequences. As a continuing example, consider (again) the following code sequence:

```
mov( SignedValue, ax ); // Get value to check.
cwd();                  // DX = FFFF if neg, 0000 otherwise.
add( $ffff, ax );       // Carry=0 if ax is zero, one otherwise.
rcl( 1, dx );           // DX = FFFF if AX is neg, 0 if ax=0,
mov( dx, Signum );      //  1 if ax>0.
```

Now consider the following code sequence that also computes the signum function:

```
mov( SignedValue, ax ); // Get value to check.
cmp( ax, 0 );           // Check the sign.
je GotSigum;            // We're done if it's zero
mov( 1, ax );           // Assume it's positive.
```

```
                jns GotSignum;          // We're done if it was positive.
                neg( ax );              // 1 -> -1, we've got a negative value.
GotSignum:      mov( ax, Signum );
```

Yes, the second version is longer and slower. However, an average person can read the instruction sequence and figure out what it's doing; hence the second version is much easier to read than the first. Which sequence is best? Unless speed or space is an extremely critical factor and you can show that this routine is in the critical execution path, then the second version is obviously better. There is a time and a place for tricky assembly code; however, it's rare that you would need to pull tricks like this throughout your code.

So how does one choose appropriate instruction sequences when there are many possible ways to accomplish the same task? The best way is to ensure that you have a choice. Although there are many different ways to accomplish an operation, few people bother to consider any instruction sequence other than the first one that comes to their mind. Unfortunately, the "best" instruction sequence is rarely the first instruction sequence that comes to most people's minds[11]. In order to make a choice, you have to have a choice to make. That means you should create at least two different code sequences for a given operation if there is ever a question concerning the readability of your code. Once you have at least two versions, you can choose between them based on your needs at hand. While it is impractical to "write your program twice" so that you'll have a choice for every sequence of instructions in the program, you should apply this technique to particularly bothersome code sequences.

Guideline: For particularly difficult to understand sections of code, try solving the problem several different ways. Then choose the most easily understood solution for actual incorporation into your program.

One problem with the above suggestion is that you're often too close to your own work to make decisions like "this code isn't too hard to understand, I don't have to worry about it." It is often a good idea to have someone else review your code and point out those sections they find hard to understand[12].

Guideline: Take advantage of reviews to determine those sections of code in your program that may need to be rewritten to make them easier to understand.

C.7.2.2 Control Structures

Ralph Griswold[13] once said (roughly) the following about C, Pascal, and Icon: "C makes it easy to write hard to read programs[14], Pascal makes it hard to write hard to read programs, and Icon makes it easy to write easy to read programs." Assembly language can be summed up like this: "Assembly language makes it hard to write easy to read programs and easy to write hard to read programs." It takes considerable discipline to write readable assembly language programs; *but it can be done*. Sadly, most assembly code you find today is extremely poorly written. Indeed, that state of affairs is the whole reason for this document. Once you get past issues like comments and naming conventions, issues like program control flow and data structure design have among the largest impacts on program readability. One need look no farther than the public domain code on the Internet, or at Microsoft's sample code for that matter[15], to see abundant examples of poorly written assembly language code.

Fortunately, with a little discipline it is possible to write readable assembly language programs. Particularly in HLA which was designed from the beginning to allow the easy creation of readable code. How you design your control structures can have a big impact on the readability of your programs. The best way to do this can be summed up in two words: avoid spaghetti.

11. This is true regardless of what metric you use to determine the "best" code sequence.
12. Of course, if the program is a *class assignment*, you may want to check your instructor's cheating policy before showing your work to your classmates!
13. The designer of the SNOBOL4 and Icon programming languages.
14. Note that this does not infer that it is hard to write easy to read C programs. Only that if one is sloppy, one can easily write something that is near impossible to understand.
15. Okay, this is a cheap shot. In fact, most of the assembly code on this planet is poorly written.

Spaghetti code is the name given to a program that has a large number of intertwined branches and branch targets within a code sequence. Consider the following example:

```
                jmp L1;
L1:             mov( 0, ax ;
                jmp L2;
L3:             mov( 1, ax );
                jmp L2;
L4:             mov( -1, ax );
                jmp L2;
L0:             mov( x, ax );
                cmp( ax, 0 );
                je L1;
                jns L3;
                jmp L4;
L2:             mov( ax, y );
```

This code sequence, by the way, is our good friend the Signum function. It takes a few moments to figure this out because as you manually trace through the code you find yourself spending more time following jumps around than you do looking at code that computes useful results. Now this is a rather extreme example, but it is also fairly short. A longer code sequence code become just as obfuscated with even fewer branches all over the place.

Spaghetti code is given this name because it resembles a bowl of spaghetti. That is, if we consider a control path in the program a spaghetti noodle, spaghetti code contains lots of intertwined branches into and out of different sections of the program. Needless to say, most spaghetti programs are difficult to understand, generally contain lots of bugs, and are often inefficient (don't forget that branches are among the slowest executing instructions on most modern processors).

So how to we resolve this? Easy by physically adopting structured programming techniques in assembly language code. Of course, "pure" 80x86 assembly language doesn't provide IF..THEN..ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, and other such statements, but we can certainly simulate them if you insist on writing "pure" assembly code[16]. Consider the following high level sequence:

```
        if(expression) then

                << statements to execute if expression is true >>

        else

                << statements to execute if expression is false >>

        endif;
```

Almost any high level language programmer can figure out what this type of statement will do. Assembly language programmers should leverage this knowledge by attempting to organize their code so it takes this same form. Specifically, the assembly language version should look something like the following:

```
                << Assembly code to compute value of expression >>

                JNxx    ElsePart ;xx is the opposite condition we want to check.

                << Assembly code corresponding to the then portion >>

                jmp     AroundElsePart

ElsePart:
                << Assembly code corresponding to the else portion >>

AroundElsePart:
```

16. We'll consider the HLA high level control statements elsewhere in this appendix.

For an concrete example, consider the following:

```
        if( ax = y ) then

                write( 'ax = y' );

        else

                write( 'ax <> y' );

        endif;

; Corresponding Assembly Code:

                mov( x, ax );
                cmp( ax, y );
                jne ElsePart;

                stdout.put( "x = y",nl );
                jmp IfDone;

ElsePart:       stdout.put( "x<>y",nl );
IfDone:
```

While this may seem like the obvious way to organize an IF..THEN.ELSE..ENDIF statement, it is surprising how many people would naturally assume they've got to place the ELSE part somewhere else in the program as follows:

```
                mov( x, ax );
                cmp( ax, y );
                jne ElsePart;

                stdout.put( "x = y", nl );
IfDone:
                 .
                 .
                 .
ElsePart:       stdout.put( "x <> y", nl );
                jmp IfDone;
```

This code organization makes the program more difficult to follow. Most programmers have a HLL background and despite a current assignment, they still work mostly in HLLs. Assembly language programs will be more readable if they mimic the HLL control constructs[17].

For similar reasons, you should attempt to organize your assembly code that simulates WHILE loops, REPEAT..UNTIL loops, FOR loops, etc., so that the code resembles the HLL code (for example, a WHILE loop should physically test the condition at the beginning of the loop with a jump at the bottom of the loop).

Rule: Attempt to design your programs using HLL control structures. The organization of the assembly code that you write should physically resemble the organization of some corresponding HLL program.

Assembly language offers you the flexibility to design arbitrary control structures. This flexibility is one of the reasons good assembly language programmers can write better code than that produced by a compiler (that can only work with high level control structures). However, keep in mind that a fast program

17. Sometimes, for performance reasons, the code sequence above is justified since straight-line code executes faster than code with jumps. If the program rarely executes the ELSE portion of an if statement, always having to jump over it could be a waste of time. But if you're optimizing for speed, you will often need to sacrifice readability.

doesn't have to contain the tightest possible code in every sequence. Execution speed is nearly irrelevant in most parts of the program. Sacrificing readability for speed isn't a big win in most of the program.

Guideline: Avoid control structures that don't easily map to well-known high level language control structures in your assembly language programs. Deviant control structures should only appear in small sections of code when efficiency demands their use.

C.7.2.3 Instruction Synonyms

HLA defines several synonyms for common instructions. This is especially true for the conditional jump and "set on condition code" instructions. For example, JA and JNBE are synonyms for one another. Logically, one could use either instruction in the same context. However, the choice of synonym can have an impact on the readability of a code sequence. To see why, consider the following:

```
                if( x <= y ) then
                    << true statements>>
                else
                    << false statements>>
                endif

// Assembly code:

                mov( x, ax );
                cmp( ax, y );
                ja ElsePart;

                << true code >>

                jmp IfDone;

ElsePart:       << false code >>
IfDone:
```

When someone reads this program, the "JA" statement skips over the true portion. Unfortunately, the "JA" instruction gives the illusion we're checking to see if something is greater than something else; in actuality, we're testing to see if some condition is less than or equal, not greater than. As such, this code sequence hides some of the original intent of high level algorithm. One solution is to swap the false and true portions of the code:

```
                mov( x, ax );
                cmp( ax, y
                jbe ThenPart;

                << false code >>

                jmp IfDone;

ThenPart:       << true code >>
IfDone:
```

This code sequence uses the conditional jump that matches the high level algorithm's test (less than or equal). However, this code is now organized in a non-standard fashion (it's an IF..ELSE..THEN..ENDIF statement). This hurts the readability more than using the proper jump helped it. Now consider the following solution:

```
                mov( x, ax );
                cmp( ax, y );
                jnbe ElsePart;

                << true code >>
                jmp IfDone;

ElsePart:       << false code >>
IfDone:
```

This code is organized in the traditional IF..THEN..ELSE..ENDIF fashion. Instead of using JA to skip over the then portion, it uses JNBE to do so. This helps indicate, in a more readable fashion, that the code falls through on below or equal and branches if it is not below or equal. Since the instruction (JNBE) is easier to relate to the original test (<=) than JA, this makes this section of code a little more readable.

Rule: When skipping over some code because some condition has failed (e.g., you fall into the code because the condition is successful), always use a conditional jump of the form "JN*xx*" to skip over the code section. For example, to fall through to a section of code if one value is less than another, use the JNL or JNB instruction to skip over the code. Of course, if you are testing a negative condition (e.g., testing for equality) then use an instruction of the form J*x* to skip over the code.

C.8 Data Types

Prior to the arrival of MASM from Microsoft for the 80x86, most assemblers provided very little capability for declaring and allocated complex data types. Generally, you could allocate bytes, words, and other primitive machine structures. You could also set aside a block of bytes. As high level languages improved their ability to declare and use abstract data types, assembly language fell farther and farther behind. Then MASM came along and changed all that[18]. HLA expands the ability to declare abstract data types even farther than MASM. Unfortunately, man new assembly language programmers don't bother learning and using these data typing facilities because they're already overwhelmed by assembly language and want to minimize the number of things they've got to learn. This is really a shame because HLA's data typing is one of the biggest improvements to assembly language since using mnemonics rather than binary opcodes for machine level programming.

Note that HLA is a "high-level" assembler. It does things assemblers for other chips won't do like checking the types of operands and reporting errors if there are mismatches. Some people, who are used to assemblers on other machines find this annoying. However, it's a great idea in assembly language for the same reason it's a great idea in HLLs[19]. These features have one other beneficial side-effect: they help other understand what you're trying to do in your programs. It should come as no surprise, then, that this style guide will encourage the use of these features in your assembly language programs.

C.8.1 Declaring Structures in Assembly Language

HLA provides an excellent facility for declaring and using records and unions; for some reason, many assembly language programmers ignore them and manually compute offsets to fields within structures in their code. Not only does this produce hard to read code, the result is nearly unmaintainable as well.

Rule: When a structure data type is appropriate in an assembly language program, declare the corresponding structure in the program and use it. Do not compute the offsets to fields in the structure manually, use the standard structure "dot-notation" to access fields of the structure.

One problem with using structures occurs when you access structure fields indirectly (i.e., through a pointer). Indirect access always occurs through a register. Once you load a pointer value into a register, the program doesn't readily indicate what pointer you are using. This is especially true if you use the indirect

18. Okay, MASM wasn't the first, but such techniques were not popularized until MASM appeared.
19. Of course, MASM gives you the ability to override this behavior when necessary. Therefore, the complaints from "old-hand" assembly language programmers that this is insane are groundless.

access several times in a section of code without reloading the register(s). One solution is to use a text constant to create a special symbol that expands as appropriate. Consider the following code:

```
type
    s:record

        a: int32;
        b: int32;

    endrecord;
        .
        .
        .
static
    r: s;
    ptr2r: pointer to s;
            .
            .
            .
        mov( ptr2r, edi );
        mov( (type s [edi]).a, eax );   // No indication this is ptr2r!
            .
            .
            .
        mov( ebx, (type s [edi).b );    // Still no indication.
```

Now consider the following:

```
type
    s:record

        a: int32;
        b: int32;

    endrecord;

    sptr : pointer to s;
            .
            .
            .
static
    r: s;
    ptr2r: sptr := q;
    ?_r:text := (type s [edi])";
            .
            .
            .
    mov( ptr2r, edi );
    mov( _r.a, eax );      // Now it's a lot more clear that we're using r.
            .
            .
            .
    mov( ebx, _r.b );      // It's still clear that we're using r!
```

Note that the "_" symbol is a legal identifier character to HLA, hence "_r" is just another symbol. Of course, you must always make sure to load the pointer into EDI when using the text constant above. If you use several different registers to access the data that "r" points at, this trick may not make the code anymore readable since you will need several text constants that all mean the same thing.

The 80x86 Instruction Set Appendix D

The following three tables discuss the integer/control, floating point, and MMX instruction sets. This document uses the following abbreviations:

imm-	A constant value, must be appropriate for the operand size.
imm8-	An eight-bit immediate constant. Some instructions limit the range of this value to less than 0..255.
immL-	A 16- or 32-bit immediate constant.
immH-	A 16- or 32-bit immediate constant.
reg-	A general purpose integer register.
reg8-	A general purpose eight-bit register
reg16-	A general purpose 16-bit register.
reg32-	A general purpose 32-bit register.
mem-	An arbitrary memory location using any of the available addressing modes.
mem16-	A word variable using any legal addressing mode.
mem32-	A dword variable using any legal addressing mode.
mem64-	A qword variable using any legal addressing mode.
label-	A statement label in the program.
ProcedureName-	The name of a procedure in the program.

Instructions that have two source operands typically use the first operand as a source operand and the second operand as a destination operand. For exceptions and other formats, please see the description for the individual instruction.

Note that this appendix only lists those instructions that are generally useful for application programming. HLA actually supports some additional instructions that are useful for OS kernel developers; please see the HLA documentation for more details on those instructions.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
aaa()	ASCII Adjust after Addition. Adjusts value in AL after a decimal addition operation.
aad()	ASCII Adjust before Division. Adjusts two unpacked values in AX prior to a decimal division.
aam()	ASCII Adjust AX after Multiplication. Adjusts the result in AX for a decimal mulitply.
aas()	ASCII Adjust AL after Subtraction. Adjusts the result in AL for a decimal subtraction.

adc(imm, reg);
adc(imm, mem);
adc(reg, reg);
adc(reg, mem);
adc(mem, reg);
	Add with carry. Adds the source operand plus the carry flag to the destination operand.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

add(imm, reg);
add(imm, mem);
add(reg, reg);
add(reg, mem);
add(mem, reg);
	Add. Adds the source operand to the destination operand.

and(imm, reg);
and(imm, mem);
and(reg, reg);
and(reg, mem);
and(mem, reg);
	Bitwise AND. Logically ANDs the source operand into the destination operand. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

bound(reg, mem);
bound(reg, immL, immH);
	Bounds check. Reg and memory operands must be the same size and they must be 16 or 32-bit values. This instruction compares the register operand against the value at the specified memory location and raises an exception if the register's value is less than the value in the memory location. If greater or equal, then this instruction compares the register to the next word or dword in memory and raises an exception if the register's value is greater.
	
	The second form of this instruction is an HLA extended syntax instruction. HLA encodes the constants as two memory locations and then emits the first form of this instruction using these newly created memory locations.
	For the second form, the constant values must not exceed the 16-bit or 32-bit register size.

bsf(reg, reg);
bsr(mem, reg);
	Bit Scan Forward. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the first set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.

bsr(reg, reg);
bsr(mem, reg);
	Bit Scan Reverse. The two operands must be the same size and they must be 16-bit or 32-bit operands. This instruction locates the last set bit in the source operand and stores the bit number into the destination operand and clears the zero flag. If the source operand does not have any set bits, then this instruction sets the zero flag and the dest register value is undefined.

bswap(reg32);	Byte Swap. This instruction reverses the order of the bytes in a 32-bit register. It swaps bytes zero and three and it swaps bytes one and two. This effectively converts data between the little endian (used by Intel) and big endian (used by some other CPUs) formats.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

bt(reg, mem);
bt(reg, reg);
bt(imm8, reg);
bt(imm8, mem);
	Register and memory operands must be 16- or 32-bit values. Eight bit immediate values must be in the range 0..15 for 16-bit registers, 0..31 for 32-bit registers, and 0..255 for memory operands. Source register must be in the range 0..15 or 0..31 for registers. Any value is legal for the source register if the destination operand is a memory location. This instruction copies the bit in the second operand, whose bit position the first operand specifies, into the carry flag.

btc(reg, mem);
btc(reg, reg);
btc(imm8, reg);
btc(imm8, mem);
	Bit test and complement. As above, except this instruction also complements the value of the specified bit in the second operand. Note that this instruction first copies the bit to the carry flag, then complements it. To support atomic operations, the memory-based forms of this instruction are always "memory locked" and they always directly access main memory; the CPU does not use the cache for this result. Hence, this instruction always operates at memory speeds (i.e., slow).

btr(reg, mem);
btr(reg, reg);
btr(imm8, reg);
btr(imm8, mem);
	Bit test and reset. Same as BTC except this instruction tests and resets (clears) the bit.

bts(reg, mem);
bts(reg, reg);
bts(imm8, reg);
bts(imm8, mem);
	Bit test and set. Same as BTC except this instructions tests and sets the bit.

call label;
call(label);
call(reg32);
call(mem32);
	Pushes a return address onto the stack and calls the subroutine at the address specified. Note that the first two forms are the same instruction. The other two forms provide indirect calls via a register or a pointer in memory.

cbw();	Convert Byte to Word. Sign extends AL into AX.
cdq();	Convert double word to quadword. Sign extends EAX into EDX:EAX.
clc();	Clear Carry.
cld();	Clear direction flag. When the direction flag is clear the string instructions increment ESI and/or EDI after each operation.
cli();	Clear the interrupt enable flag.
cmc();	Complement (invert) Carry.

cmova(mem, reg);
cmova(reg, reg);
cmova(reg, mem);
	Conditional Move (if above). Copies the source operand to the destination operand if the previous comparison found the left operand to be greater than (unsigned) the right operand (c=0, z=0). Register and memory operands must be 16-bit or 32-bit values, eight-bit operands are illegal. Does not affect the destination operand if the condition is false.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovae(mem, reg);
cmovae(reg, reg);
cmovae(reg, mem);
	Conditional move if above or equal (see cmova for details).

cmovb(mem, reg);
cmovb(reg, reg);
cmovb(reg, mem);
	Conditional move if below (see cmova for details).

cmovbe(mem, reg);
cmovbe(reg, reg);
cmovbe(reg, mem);
	Conditional move if below or equal (see cmova for details).

cmovc(mem, reg);
cmovc(reg, reg);
cmovc(reg, mem);
	Conditional move if carry set (see cmova for details).

cmove(mem, reg);
cmove(reg, reg);
cmove(reg, mem);
	Conditional move if equal (see cmova for details).

cmovg(mem, reg);
cmovg(reg, reg);
cmovg(reg, mem);
	Conditional move if (signed) greater (see cmova for details).

cmovge(mem, reg);
cmovge(reg, reg);
cmovge(reg, mem);
	Conditional move if (signed) greater or equal (see cmova for details).

cmovl(mem, reg);
cmovl(reg, reg);
cmovl(reg, mem);
	Conditional move if (signed) less than (see cmova for details).

cmovle(mem, reg);
cmovle(reg, reg);
cmovle(reg, mem);
	Conditional move if (signed) less than or equal (see cmova for details).

cmovna(mem, reg);
cmovna(reg, reg);
cmovna(reg, mem);
	Conditional move if (unsigned) not greater (see cmova for details).

cmovnae(mem, reg);
cmovnae(reg, reg);
cmovnae(reg, mem);
	Conditional move if (unsigned) not greater or equal (see cmova for details).

cmovnb(mem, reg);
cmovnb(reg, reg);
cmovnb(reg, mem);
	Conditional move if (unsigned) not less than (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovnbe(mem, reg);
cmovnbe(reg, reg);
cmovnbe(reg, mem);
	Conditional move if (unsigned) not less than or equal (see cmova for details).

cmovnc(mem, reg);
cmovnc(reg, reg);
cmovnc(reg, mem);
	Conditional move if no carry/carry clear (see cmova for details).

cmovne(mem, reg);
cmovne(reg, reg);
cmovne(reg, mem);
	Conditional move if not equal (see cmova for details).

cmovng(mem, reg);
cmovng(reg, reg);
cmovng(reg, mem);
	Conditional move if (signed) not greater (see cmova for details).

cmovnge(mem, reg);
cmovnge(reg, reg);
cmovnge(reg, mem);
	Conditional move if (signed) not greater or equal (see cmova for details).

cmovnl(mem, reg);
cmovnl(reg, reg);
cmovnl(reg, mem);
	Conditional move if (signed) not less than (see cmova for details).

cmovnle(mem, reg);
cmovnle(reg, reg);
cmovnle(reg, mem);
	Conditional move if (signed) not less than or equal (see cmova for details).

cmovno(mem, reg);
cmovno(reg, reg);
cmovno(reg, mem);
	Conditional move if no overflow / overflow flag = 0 (see cmova for details).

cmovnp(mem, reg);
cmovnp(reg, reg);
cmovnp(reg, mem);
	Conditional move if no parity / parity flag = 0 / odd parity (see cmova for details).

cmovns(mem, reg);
cmovns(reg, reg);
cmovns(reg, mem);
	Conditional move if no sign / sign flag = 0 (see cmova for details).

cmovnz(mem, reg);
cmovnz(reg, reg);
cmovnz(reg, mem);
	Conditional move if not zero (see cmova for details).

cmovo(mem, reg);
cmovo(reg, reg);
cmovo(reg, mem);
	Conditional move if overflow / overflow flag = 1 (see cmova for details).

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

cmovp(mem, reg);
cmovp(reg, reg);
cmovp(reg, mem);
	Conditional move if parity flag = 1 (see cmova for details).

cmovpe(mem, reg);
cmovpe(reg, reg);
cmovpe(reg, mem);
	Conditional move if even parity / parity flag = 1(see cmova for details).

cmovpo(mem, reg);
cmovpo(reg, reg);
cmovpo(reg, mem);
	Conditional move if odd parity / parity flag = 0 (see cmova for details).

cmovs(mem, reg);
cmovs(reg, reg);
cmovs(reg, mem);
	Conditional move if sign flag = 1 (see cmova for details).

cmovz(mem, reg);
cmovz(reg, reg);
cmovz(reg, mem);
	Conditional move if zero flag = 1 (see cmova for details).

cmp(imm, reg);
cmp(imm, mem);
cmp(reg, reg);
cmp(reg, mem);
cmp(mem, reg);
	Compare. Compares the first operand against the second operand. The two operands must be the same size. This instruction sets the condition code flags as appropriate for the condition jump and set instructions. This instruction does not change the value of either operand.

cmpsb();
repe.cmpsb();
repne.cmpsb();
	Compare string of bytes. Compares the byte pointed at byte ESI with the byte pointed at by EDI and then adjusts ESI and EDI by ±1 depending on the value of the direction flag. Sets the flags according to the result. With the REPNE (repeat while not equal) flag, this instruction compares up to ECX bytes until all the first byte it finds in the two string that are equal. With the REPE (repeat while equal) prefix, this instruction compares two strings up to the first byte that is different. See the chapter on the String Instructions for more details.

cmpsw()
repe.cmpsw();
repne.cmpsw();
	Compare a string of words. Like cmpsb except this instruction compares words rather than bytes and adjusts ESI/EDI by ±2.

cmpsd()
repe.cmpsd();
repne.cmpsd();
	Compare a string of double words. Like cmpsb except this instruction compares double words rather than bytes and adjusts ESI/EDI by ±4.

cmpxchg(reg, mem);
cmpxchg(reg, reg);
	Reg and mem must be the same size. They can be eight, 16, or 32 bit objects. This instruction compares the value in the accumulator (al, ax, or eax) against the second operand. If the two values are equal, this instruction copies the source (first) operand to the destination (second) operand. Otherwise, this instruction copies the second operand into the accumulator.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
cmpxchg8b(mem64);	Compares the 64-bit value in EDX:EAX with the memory operand. If the values are equal, then this instruction stores the 64-bit value in ECX:EBX into the memory operand and sets the zero flag. Otherwise, this instruction copies the 64-bit memory operand into the EDX:EAX registers and clears the zero flag.
cpuid();	CPU Identification. This instruction identifies various features found on the different Pentium processors. See the Intel documentation on this instruction for more details.
cwd();	Convert Word to Double. Sign extends AX to DX:AX.
cwde();	Convert Word to Double Word Extended. Sign extends AX to EAX.
daa();	Decimal Adjust after Addition. Adjusts value in AL after a decimal addition.
das();	Decimal Adjust after Subtraction. Adjusts value in AL after a decimal subtraction.

dec(reg);
dec(mem);
	Decrement. Subtracts one from the destination memory location or register.

div(reg);
div(reg8, ax);
div(reg16, dx:ax);
div(reg32, edx:eax);
div(mem);
div(mem8, ax);
div(mem16, dx:ax);
div(mem32, edx:eax);
div(imm8, ax);
div(imm16, dx:ax);
div(imm32, edx:eax);
	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX).
	
	This instruction performs an unsigned division.

enter(imm16, imm8);	Enter a procedure. Creates an activation record for a procedure. The first constant specifies the number of bytes of local variables. The second parameter (in the range 0..31) specifies the static nesting level (lex level) of the procedure.

idiv(reg);
idiv(reg8, ax);
idiv(reg16, dx:ax);
idiv(reg32, edx:eax);
idiv(mem);
idiv(mem8, ax);
idiv(mem16, dx:ax);
idiv(mem32, edx:eax);
idiv(imm8, ax);
idiv(imm16, dx:ax);
idiv(imm32, edx:eax);
	Divides accumulator or extended accumulator (dx:ax or edx:eax) by the source operand. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then divides the accumulator or extended accumulator by the contents of this memory location. Note that the accumulator operand is twice the size of the source (divisor) operand. This instruction computes the quotient and places it in AL, AX, or EAX and it computes the remainder and places it in AH, DX, or EDX (depending on the divisor's size). This instruction raises an exception if you attempt to divide by zero or if the quotient doesn't fit in the destination register (AL, AX, or EAX).
	
	This instruction performs a signed division. The condition code bits are undefined after executing this instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

imul(reg);
imul(reg8, al);
imul(reg16, ax);
imul(reg32, eax);
imul(mem);
imul(mem8, al);
imul(mem16, ax);
imul(mem32, eax);
imul(imm8, al);
imul(imm16, ax);
imul(imm32, eax);
	Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.
	
	This instruction performs a signed multiplication. Also see INTMUL.
	This instruction sets the carry and overflow flag if the H.O. portion of the result (AH, DX, EDX) is not a sign extension of the L.O. portion of the product. The sign and zero flags are undefined after the execution of this instruction.

in(imm8, al);
in(imm8, ax);
in(imm8, eax);
in(dx, al);
in(dx, ax);
in(dx, eax);
	Input data from a port. These instructions read a byte, word, or double word from an input port and place the input data into the accumulator register. Immediate port constants must be in the range 0..255. For all other port addresses you must use the DX register to hold the 16-bit port number. Note that this is a privileged instruction that will raise an exception in many Win32 Operating Systems.

inc(reg);
inc(mem);
	Increment. Adds one to the specified memory or register operand. Does not affect the carry flag. Sets the overflow flag if there was signed overflow. Sets the zero and sign flags according to the result. Note that Z=1 indicates an unsigned overflow.

int(imm8);	Call an interrupt service routine specified by the immediate operand. Note that Windows does not use this instruction for system calls, so you will probably never use this instruction under Windows. Note that INT(3); is the user breakpoint instruction (that raises an appropriate exception). INT(0) is the divide error exception. INT(4) is the overflow exception. However, it’s better to use the HLA RAISE statement than to use this instruction for these exceptions.

intmul(imm, reg);
intmul(imm, reg, reg);
intmul(imm, mem, reg);
intmul(reg, reg);
intmul(mem, reg);
	Integer mutiply. Multiplies the destination (last) operand by the source operand (if there are only two operands; or it multiplies the two source operands together and stores the result in the destination operand (if there are three operands). The operands must all be 16 or 32-bit operands and they must all be the same size.
	
	This instruction computes a signed product. This instruction sets the overflow and carry flags if there was a signed arithmetic overflow; the zero and sign flags are undefined after the execution of this instruction.

into();	Raises an exception if the overflow flag is set. Note: the HLA pseudo-variable “@into” controls the code generation for this instruction. If @into is false, HLA ignores this instruction; if @into is true (default), then HLA emits the object code for this instruction. Note that if the overflow flag is set, this instruction behaves like the “INT(4);” instruction.
iret();	Return from an interrupt. This instruction is not generally usable from an application program. It is for use in interrupt service routines only.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
ja *label*;	Conditional jump if (unsigned) above. You would generally use this instruction immediately after a CMP instruction to test to see if one operand is greater than another using an unsigned comparison. Control transfers to the specified label if this condition is true, control falls through to the next instruction if the condition is false.
jae *label*;	Conditional jump if (unsigned) above or equal. See JA above for details.
jb *label*;	Conditional jump if (unsigned) below. See JA above for details.
jbe *label*;	Conditional jump if (unsigned) below or equal. See JA above for details.
jc *label*;	Conditional jump if carry is one. See JA above for details.
je *label*;	Conditional jump if equal. See JA above for details.
jg *label*;	Conditional jump if (signed) greater. See JA above for details.
jge *label*;	Conditional jump if (signed) greater or equal. See JA above for details.
jl *label*;	Conditional jump if (signed) less than. See JA above for details.
jle *label*;	Conditional jump if (signed) less than or equal. See JA above for details.
jna *label*;	Conditional jump if (unsigned) not above. See JA above for details.
jnae *label*;	Conditional jump if (unsigned) not above or equal. See JA above for details.
jnb *label*;	Conditional jump if (unsigned) below. See JA above for details.
jnbe *label*;	Conditional jump if (unsigned) below or equal. See JA above for details.
jnc *label*;	Conditional jump if carry flag is clear (no carry). See JA above for details.
jne *label*;	Conditional jump if not equal. See JA above for details.
jng *label*;	Conditional jump if (signed) not greater. See JA above for details.
jnge *label*;	Conditional jump if (signed) not greater or equal. See JA above for details.
jnl *label*;	Conditional jump if (signed) not less than. See JA above for details.
jnle *label*;	Conditional jump if (signed) not less than or equal. See JA above for details.
jno *label*;	Conditional jump if no overflow (overflow flag = 0). See JA above for details.
jnp *label*;	Conditional jump if no parity/parity odd (parity flag = 0). See JA above for details.
jns *label*;	Conditional jump if no sign (sign flag = 0). See JA above for details.
jnz *label*;	Conditional jump if not zero (zero flag = 0). See JA above for details.
jo *label*;	Conditional jump if overflow (overflow flag = 1). See JA above for details.
jp *label*;	Conditional jump if parity (parity flag = 1). See JA above for details.
jpe *label*;	Conditional jump if parity even (parity flag = 1). See JA above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
jpo *label*;	Conditional jump if parity odd (parity flag = 0). See JA above for details.
js *label*;	Conditional jump if sign (sign flag = 0). See JA above for details.
jz *label*;	Conditional jump if zero (zero flag = 0). See JA above for details.
jcxz *label*;	Conditional jump if CX is zero. See JA above for details. Note: the range of this branch is limited to ±128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing CX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.
jecxz *label*;	Conditional jump if ECX is zero. See JA above for details. Note: the range of this branch is limited to ±128 bytes around the instruction. HLA does not check for this (MASM reports the error when it assembles HLA's output). Since this instruction is slower than comparing ECX to zero and using JZ, you probably shouldn't even use this instruction. If you do, be sure that the target label is nearby in your code.

jmp *label*;
jmp(*label*);
jmp *ProcedureName*;
jmp(mem32);
jmp(reg32);
	Jump Instruction. This instruction unconditionally transfers control to the specified destination operand. If the operand is a 32-bit register or memory location, the JMP instruction transfers control to the instruction whose address appears in the register or the memory location.
	
	Note: you should execise great care when jumping to a procedure label. The JMP instruction does not push a return address or any other data associated with a procedure's activation record. Hence, when the procedure attempts to return it will use data on the stack that was pushed prior to the execution of the JMP instruction; it is your responsibility to ensure such data is present on the stack when using JMP to transfer control to a procedure.

lahf();	Load AH from FLAGs. This instruction loads the AH register with the L.O. eight bits of the FLAGs register. See SAHF for the flag layout.

lea(reg32, mem);
lea(mem, reg32);
	Load Effective Address. These instructions, which are both semantically identical, load the 32-bit register with the address of the specified memory location. The memory location does not need to be a double word object. Note that there is never any ambiguity in this instruction since the register is always the destination operand and the memory location is always the source.

leave();	Leave procedure. This instruction cleans up the activation record for a procedure prior to returning from the procedure. You would normally use this instruction to clean up the activation record created by the ENTER instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

lock prefix	The lock prefix assets a special pin on the processor during the execution of the following instruction. In a multiprocessor environment, this ensures that the processor has exclusive use of a shared memory object while the instruction executes. The lock prefix may only precede one of the folowing instructions:
	ADD, ADC, AND BTC, BTR, BTS, CMPXCHG, DEC, INC NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG. Furthermore, this prefix is only valid for the forms that have a memory operand as their destination operand. Any other instruction or addressing mode will raise an undefined opcode exception.
	
	HLA does not directly support the LOCK prefix on these instructions (if it did, you would normally write instructions like “lock.add();” and “lock.bts();” However, you can easily add this instruction to HLA’s instruction set through the use of the following macro:
	`#macro lock;`
	`byte $F0; // $F0 is the opcode for the lock prefix.`
	`#endmacro;`
	
	To use this macro, simply precede the instruction you wish to lock with an invocation of the macro, e.g.,
	
	`lock add( al, mem );`
	
	Note that a LOCK prefix will dramatically slow an instruction down since it must access main memory (i.e., no cache) and it must negotiate for the use of that memory location with other processors in a multiprocessor system. The LOCK prefix has very little value in single processor systems.

lodsb();	Load String Byte. Loads AL with the byte whose address appears in ESI. Then it increments or decrements ESI by one depending on the value of the direction flag. See the chapter on string instructions for more details. Note: HLA does not allow the use of any repeat prefix with this instruction.
lodsw();	Load String Word. Loads AX from [ESI] and adds ±2 to ESI. See LODSB for more details.
loadsd();	Load String Double Word. Loads EAX from [ESI] and adds ±4 to ESI. See LODSB for more details.
loop *label*;	Decrements ECX and jumps to the target label if ECX is not zero. See JA for more details. Like JECX, this instruction is limited to a range of ±128 bytes around the instruction and only MASM will catch the range error. Since this instruction is actually slower than a DEC/JNZ pair, you should probably avoid using this instruction.
loope *label*;	Check the zero flag, decrement ECX, and branch if the zero flag was set and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.
loopne *label*;	Check the zero flag, decrement ECX, and branch if the zero flag was clear and ECX did not become zero. Same limitations as LOOP. See LOOP above for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
loopnz *label*;	Same instruction at LOOPNE.
loopz *label*;	Same instruction as LOOPZ.

mov(imm, reg);
mov(imm, mem);
mov(reg, reg);
mov(reg, mem);
mov(mem, reg);
 	
mov(mem16, mem16);
mov(mem32, mem32);
	Move. Copies the data from the source (first) operand to the destination (second) operand. The operands must be the same size. Note that the memory to memory moves are an HLA extension. HLA compiles these statements into a push(source)/pop(dest) instruction pair.

movsb();
rep.movsb();
	Move string of bytes. Copies the byte pointed at byte ESI to the byte pointed at by EDI and then adjusts ESI and EDI by ±1 depending on the value of the direction flag. With the REP (repeat) prefix, this instruction moves ECX bytes. See the chapter on the String Instructions for more details.

movsw();
rep.movsw();
	Move string of words. Like MOVSB above except it copies words and adjusts ESI/EDI by ±2.

movsd();
rep.movsd();
	Move string of words. Like MOVSB above except it copies double words and adjusts ESI/EDI by ±4.

movsx(reg, reg);
movsx(mem, reg);
	Move with sign extension. Copies the (smaller) source operand to the (larger) destination operand and sign extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.

movzx(reg, reg);
movzx(mem, reg);
	Move with zero extension. Copies the (smaller) source operand to the (larger) destination operand and zero extends the value to the size of the larger operand. The source operand must be smaller than the destination operand.

mul(reg);
mul(reg8, al);
mul(reg16, ax);
mul(reg32, eax);
mul(mem);
mul(mem8, al);
mul(mem16, ax);
mul(mem32, eax);
mul(imm8, al);
mul(imm16, ax);
mul(imm32, eax);
	Multiplies the accumulator (AL, AX, or EAX) by the source operand. The source operand will be the same size as the accumulator. The product produces an operand that is twice the size of the two operands with the product winding up in AX, DX:AX, or EDX:EAX. Note that the instructions involving an immediate operand are HLA extensions. HLA creates a memory object holding these constants and then multiplies the accumulator by the contents of this memory location.
	
	This instruction performs a signed multiplication. Also see INTMUL. The carry and overflow flags are cleared if the H.O. portion of the result is zero, they are set otherwise. The sign and zero flags are undefined after this instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

neg(reg);
neg(mem);
	Negate. Computes the two's complement of the operand and leaves the result in the operand. This instruction clears the carry flag if the result is zero, it sets the carry flag otherwise. It sets the overflow flag if the original value was the smallest possible negative value (which has no positive counterpart). It sets the sign and zero flags according to the result obtained.

nop();	No Operation. Consumes space and time but does nothing else. Same instruction as "xchg(eax, eax);"

not(reg);
not(mem);
	Bitwise NOT. Inverts all the bits in its operand. **Note**: this instruction does not affect any flags.

or(imm, reg);
or(imm, mem);
or(reg, reg);
or(reg, mem);
or(mem, reg);
	Bitwise OR. Logically ORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

out(al, imm8);
out(ax, imm8);
out(eax, imm8);
out(al, dx);
out(ax, dx);
out(eax, dx);
	Outputs the accumulator to the specified port. See the IN instruction for limitations under Win32.

pop(reg);
pop(mem);
	Pop a value off the stack. Operands must be 16 or 32 bits.

popa();	Pops all the 16-bit registers off the stack. The popping order is DI, SI, BP, SP, BX, DX, CX, AX.
popad();	Pops all the 32-bit registers off the stack. The popping order is EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX.
popf();	Pops the 16-bit FLAGS register off the stack. Note that in user (application) mode, this instruction ignores the interrupt disable flag value it pops off the stack.
popfd();	Pops the 32-bit EFLAGS register off the stack. Note that in user (application) mode, this instruction ignores many of the bits it pops off the stack.

push(reg);
push(mem);
	Pushes the specified 16-bit or 32-bit register or memory location onto the stack. Note that you cannot push eight-bit objects.

pusha();	Pushes all the 16-bit general purpose registers onto the stack in the roder AX, CX, DX, BX, SP, BP, SI, DI.
pushad();	Pushes all the 32-bit general purpose registers onto the stack in the roder EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

pushd(imm);
pushd(reg);
pushd(mem);
	Pushes the 32-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.

pushf();	Pushes the value of the 16-bit FLAGS register onto the stack.
pushfd();	Pushes the value of the 32-bit FLAGS register onto the stack.

pushw(imm);
pushw(reg);
pushw(mem);
	Pushes the 16-bit operand on to the stack. Generally used to push constants or anonymous variables. Note that this is a synonym for PUSH if you specify a register or typed memory operand.

rcl(imm, reg);
rcl(imm, mem);
rcl(cl, reg);
rcl(cl, mem);
	Rotate through carry, left. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

rcr(imm, reg);
rcr(imm, mem);
rcr(cl, reg);
rcr(cl, mem);
	Rotate through carry, right. Rotates the destination (second) operand through the carry the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right).
	The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

rdtsc();	Read Time Stamp Counter. Returns in EDX:EAX the number of clock cycles that have transpired since the last reset of the processor. You can use this instruction to time events in your code (i.e., to determine whether one instruction sequence takes more time than another).

ret();
ret(imm16);
	Return from subroutine. Pops a return address off the stack and transfers control to that location. The second form of the instruction adds the immediate constant to the ESP register to remove the procedure's parameters from the stack.

rol(imm, reg);
rol(imm, mem);
rol(cl, reg);
rol(cl, mem);
	Rotate left. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the L.O. to the H.O. position (i.e., rotate left). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

ror(imm, reg);
ror(imm, mem);
ror(cl, reg);
ror(cl, mem);
	Rotate right. Rotates the destination (second) operand the number of bits specified by the first operand, shifting the bits from the H.O. to the L.O. position (i.e., rotate right). The carry flag contains the last bit shifted into it. The overflow flag, which is valid only when the shift count is one, is set if the sign changes as a result of the rotate. This instruction does not affect the other flags. In particular, **note that this instruction does not affect the sign or zero flags.**

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description
sahf();	Store AH into FLAGs. Copies the value in AH into the L.O. eight bits of the FLAGs register. Note that this instruction will not affect the interrupt disable flag when operating in user (application) mode. Bit #7 of AH goes into the Sign flag, bit #6 goes into the zero flag, bit #4 goes into the auxilary carry (BCD carry) flag, bit #2 goes into the parity flag, and bit #0 goes into the carry flag. This instruction also clears bits one, three, and five of the FLAGs register. It does not affect any other bits in FLAGs or EFLAGs.

sal(imm, reg);
sal(imm, mem);
sal(cl, reg);
sal(cl, mem);
	Shift Arithmetic Left. Same instruction as SHL. See SHL for details.

sar(imm, reg);
sar(imm, mem);
sar(cl, reg);
sar(cl, mem);
	Shift Arithmetic Right. Shifts the destination (second) operand to the right the specified number of bits using an arithmetic shift right algorithm. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction always clears the overflow flag. The sign and zero flags are set according to the result.

sbb(imm, reg);
sbb(imm, mem);
sbb(reg, reg);
sbb(reg, mem);
sbb(mem, reg);
	Subtract with borrow. Subtracts the source (first) operand and the carry from the destination (second) operand. Sets the condition code bits according to the result it computes. This instruction sets the flags the same way as the SUB instruction. See SUB for details.

scasb();
repe.scasb();
repne.scasb();
	Scan string byte. Compares the value in AL against the byte that EDI points at and sets the flags accordingly (same as the CMP instruction). Adds ±1 to EDI after the comparison (based on the setting of the direction flag). With the REPE (repeat while equal) prefix, this instruction will scan through as many as ECX bytes in memory as long as each byte that EDI points at is equal to the value in AL (i.e., it scans for the first value not equal to the value in AL). With the REPNE prefix, this instruction scans through as many as ECX bytes as long as the value that EDI points at is not equal to AL (i.e., it scans for the first byte matching AL’s value). See the chapter on string instructions for more details.

scasw();
repe.scasw();
repne.scasw();
	Scan String Word. Compares the value in AX against the word that EDI points at and set the flags. Adds ±2 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).

scasd();
repe.scasd();
repne.scasd();
	Scan String Double word. Compares the value in EAX against the double word that EDI points at and set the flags. Adds ±4 to EDI after the operation. Also supports the REPE and REPNE prefixes (see SCASB above).

seta(reg);
seta(mem);
	Conditional set if (unsigned) above (Carry=0 and Zero=0). Stores a one in the destination operand if the result of the previous comparison found the first operand to be greater than the second using an unsigned comparison. Stores a zero into the destination operand otherwise.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

setae(reg);
setae(mem);
	Conditional set if (unsigned) above or equal (Carry=0). See SETA for details.

setb(reg);
setb(mem);
	Conditional set if (unsigned) below (Carry=1). See SETA for details.

setbe(reg);
setbe(mem);
	Conditional set if (unsigned) below or equal (Carry=1 or Zero=1). See SETA for details.

setc(reg);
setc(mem);
	Conditional set if carry set (Carry=1). See SETA for details.

sete(reg);
sete(mem);
	Conditional set if equal (Zero=1). See SETA for details.

setg(reg);
setg(mem);
	Conditional set if (signed) greater (Sign=Overflow and Zero=0). See SETA for details.

setge(reg);
setge(mem);
	Conditional set if (signed) greater or equal (Sign=Overflow or Zero=1). See SETA for details.

setl(reg);
setl(mem);
	Conditional set if (signed) less than (Sign<>Overflow). See SETA for details.

setle(reg);
setle(mem);
	Conditional set if (signed) less than or equal (Sign<>Overflow or Zero = 1). See SETA for details.

setna(reg);
setna(mem);
	Conditional set if (unsigned) not above (Carry=1 or Zero=1). See SETA for details.

setnae(reg);
setnae(mem);
	Conditional set if (unsigned) not above or equal (Carry=1). See SETA for details.

setnb(reg);
setnb(mem);
	Conditional set if (unsigned) not below (Carry=0). See SETA for details.

setnbe(reg);
setnbe(mem);
	Conditional set if (unsigned) not below or equal (Carry=0 and Zero=0). See SETA for details.

setnc(reg);
setnc(mem);
	Conditional set if carry clear (Carry=0). See SETA for details.

setne(reg);
setne(mem);
	Conditional set if not equal (Zero=0). See SETA for details.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

setng(reg);
setng(mem);
	Conditional set if (signed) not greater (Sign<>Overflow or Zero = 1). See SETA for details.

setnge(reg);
setnge(mem);
	Conditional set if (signed) not greater than (Sign<>Overflow). See SETA for details.

setnl(reg);
setnl(mem);
	Conditional set if (signed) not less than (Sign=Overflow or Zero=1). See SETA for details.

setnle(reg);
setnle(mem);
	Conditional set if (signed) not less than or equal (Sign=Overflow and Zero=0). See SETA for details.

setno(reg);
setno(mem);
	Conditional set if no overflow (Overflow=0). See SETA for details.

setnp(reg);
setnp(mem);
	Conditional set if no parity (Parity=0). See SETA for details.

setns(reg);
setns(mem);
	Conditional set if no sign (Sign=0). See SETA for details.

setnz(reg);
setnz(mem);
	Conditional set if not zero (Zero=0). See SETA for details.

seto(reg);
seto(mem);
	Conditional set if Overflow (Overflow=1). See SETA for details.

setp(reg);
setp(mem);
	Conditional set if Parity (Parity=1). See SETA for details.

setpe(reg);
setpe(mem);
	Conditional set if Parity even (Parity=1). See SETA for details.

setpo(reg);
setpo(mem);
	Conditional set if Parity odd (Parity=0). See SETA for details.

sets(reg);
sets(mem);
	Conditional set if sign set(Sign=1). See SETA for details.

setz(reg);
setz(mem);
	Conditional set if zero (Zero=1). See SETA for details.

shl(imm, reg);
shl(imm, mem);
shl(cl, reg);
shl(cl, mem);
	Shift left. Shifts the destination (second) operand to the left the number of bit positions specified by the first operand. The carry flag contains the value of the last bit shifted out of the second operand. The overflow flag is only defined when the bit shift count is one, this instruction sets overflow flag if the sign changes as a result of this instruction’s execution. The sign and zero flags are set according to the result.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

shld(imm8, reg, reg);
shld(imm8, reg, mem);
shld(cl, reg, reg);
shld(cl, reg, mem);
	Shift Left Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the L.O. word or double word and the third operand being the H.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHL. Note that this instruction does not affect the source (second) operand's value.

shr(imm, reg);
shr(imm, mem);
shr(cl, reg);
shr(cl, mem);
	Shift right. Shifts the destination (second) operand to the right the number of bit positions specified by the first operand. The last bit shifted out goes into the carry flag. The overflow flag is set if the H.O. bit originally contained one. The sign flag is cleared and the zero flag is set if the result is zero.

shrd(imm8, reg, reg);
shrd(imm8, reg, mem);
shrd(cl, reg, reg);
shrd(cl, reg, mem);
	Shift Right Double precision. The first operand is a bit count. The second operand is a source and the third operand is a destination. These operands must be the same size and they must be 16- or 32-bit values (no eight bit operands). This instruction treats the second and third operands as a double precision value with the second operand being the H.O. word or double word and the third operand being the L.O. word or double word. The instruction shifts this double precision value the specified number of bits and sets the flags in a manner identical to SHR. Note that this instruction does not affect the source (second) operand's value.

stc();	Set Carry. Sets the carry flag to one.
std();	Set Direction. Sets the direction flag to one. If the direction flag is one, the string instructions decrement ESI and/or EDI after each operation.
sti();	Set interrupt enable flag. Generally this instruction is not usable in user (application) mode. In kernel mode it allows the CPU to begin processing interrupts.

stosb();
rep.stosb();
	Store String Byte. Stores the value in AL at the location whose address EDI contains. Then in adds ±1 to EDI. If the REP prefix is present, this instruction repeats the number of times specified in the ECX register. This instruction is useful for quickly clearing out byte arrays.

stosw();
rep.stosw();
	Store String Word. Stores the value in AX at location [EDI] and then adds ±2 to EDI. See STOSB for details.

stosd();
rep.stosd();
	Store String Double word. Stores the value in EAX at location [EDI] and then adds ±4 to EDI. See STOSB for details.

sub(imm, reg);
sub(imm, mem);
sub(reg, reg);
sub(reg, mem);
sub(mem, reg);
	Subtract. Subtracts the first operand from the second operand and leaves the difference in the destination (second) operand. Sets the zero flag if the two values were equal (which produces a zero result), sets the carry flag if there was unsigned overflow or underflow; sets the overflow if there was signed overflow or underflow; sets the sign flag if the result is negative (H.O. bit is one). Note that SUB sets the flags identically to the CMP instruction, so you can use conditional jump or set instructions after SUB the same way you would use them after a CMP instruction.

Table 1: 80x86 Integer and Control Instruction Set

Instruction Syntax	Description

test(imm, reg);
test(imm, mem);
test(reg, reg);
test(reg, mem);
test(mem, reg);
	Test operands. Logically ANDs the two operands together and sets the flags but does not store the computed result (i.e., it does not disturb the value in either operand). Always clears the carry and overflow flags. Sets the sign flag if the H.O. bit of the computed result is one. Sets the zero flag if the computed result is zero.

xadd(mem, reg);
xadd(reg, reg);
	Adds the first operand to the second operand and then stores the original value of the second operand into the first operand:
	`xadd( source, dest );`
	`temp := dest`
	`dest := dest + source`
	`source := temp`
	This instruction sets the flags in a manner identical to the ADD instruction.

xchg(reg, reg);
xchg(reg, mem);
xchg(mem, reg);
	Swaps the values in the two operands which must be the same size. Does not affect any flags.

xlat();	Translate. Computes AL := [EBX + AL]; That is, it uses the value in AL as an index into a lookup table whose base address is in EBX. It copies the specified byte from this table into AL.

xor(imm, reg);
xor(imm, mem);
xor(reg, reg);
xor(reg, mem);
xor(mem, reg);
	Exclusive-OR. Logically XORs the source operand with the destination operand and leaves the result in the destination. The two operands must be the same size. Clears the carry and overflow flags and sets the sign and zero flags according to the result.

Table 2: Floating Point Instruction Set

Instruction	Description
f2xm1();	Compute 2^x-1 in ST0, leaving the result in ST0.
fabs();	Computes the absolute value of ST0.

fadd(mem);
fadd(st*i*, st0);
fadd(st0, st*i*);
	Add operand to st0 or add st0 to destination register (st*i*, i=0..7). If the operand is a memory operand, it must be a *real32* or *real64* object.

faddp();
faddp(st0, st*i*);
	With no operands, this instruction adds st0 to st1 and then pops st0 off the FPU stack.

Table 2: Floating Point Instruction Set

Instruction	Description
fbld(mem80);	This instruction loads a ten-byte (80-bit) packed BCD value from memory and converts it to a real80 object. This instruction does not check for an invalid BCD value. If the BCD number contains illegal digits, the result is undefined.
fbstp(mem80);	This instruction pops the real80 object off the top of the FPU stack, converts it to an 80-bit BCD value, and stores the result in the specified memory location (tbyte).
fchs();	This instruction negates the floating point value on the top of the stack (st0).
fclex();	This instruction clears the floating point exception flags.
fcmova(st*i*, st0);[a]	Floating point conditional move if above. Copies st*i* to st0 if c=0 and z=0 (unsigned greater than after a CMP).
fcmovae(st*i*, st0);	Floating point conditional move if above or equal. Copies st*i* to st0 if c=0 (unsigned greater or equal after a CMP).
fcmovb(st*i*, st0);	Floating point conditional move if below. Copies st*i* to st0 if c=1 (unsigned less than after a CMP).
fcmovbe(st*i*, st0);	Floating point conditional move if below or equal. Copies st*i* to st0 if c=1 or z=1 (unsigned less than or equal after a CMP).
fcmove(st*i*, st0);	Floating point conditional move if equal. Copies st*i* to st0 if z=1 (equal after a CMP).
fcmovna(st*i*, st0);	Floating point conditional move if not above. Copies st*i* to st0 if c=1 or z=1 (unsigned not above after a CMP).
fcmovnae(st*i*, st0);	Floating point conditional move if not above or equal. Copies st*i* to st0 if c=1 (unsigned not above or equal after a CMP).
fcmovnb(st*i*, st0);	Floating point conditional move if not below. Copies st*i* to st0 if c=0 (unsigned not below after a CMP).
fcmovnbe(st*i*, st0);	Floating point conditional move if not below or equal. Copies st*i* to st0 if c=0 and z=0 (unsigned not below or equal after a CMP).
fcmovne(st*i*, st0);	Floating point conditional move if not equal. Copies st*i* to st0 if z=0 (not equal after a CMP).
fcmovnu(st*i*, st0);	Floating point conditional move if not unordered. Copies st*i* to st0 if the last floating point comparison did not produce an unordered result (parity flag = 0).
fcmovu(st*i*, st0);	Floating point conditional move if not unordered. Copies st*i* to st0 if the last floating point comparison produced an unordered result (parity flag = 1).

fcom();
fcom(mem);
fcom(st0, st*i*);
	Compares the value of ST0 with the operand and sets the floating point condition bits based on the comparison. If the operand is a memory operand, it must be a *real32* or *real64* value. Note that to test the condition codes you will have to copy the floating point status word to the FLAGs register; see the chapter on floating point arithmetic for details.

Table 2: Floating Point Instruction Set

Instruction	Description
fcomi(st0, st*i*);[b]	Compares the value of ST0 with the second operand and sets the appropriate bits in the FLAGs register.
fcomip(st0, st*i*);	Compares the value of ST0 with the second operand, sets the appropriate bits in the FLAGs register, and then pops ST0 off the FPU stack.

fcomp();
fcomp(mem);
fcomp(st*i*);
	Compares the value of ST0 with the operand, sets the floating point status bits, and then pops ST0 off the floating point stack. With no operands, this instruction compares ST0 to ST1. Memory operands must be *real32* or *real64* objects.

fcompp();	Compares ST0 to ST1 and then pops both values off the stack. Leaves the result of the comparison in the floating point status register.
fcos();	Computes ST0 = cos(ST0).
fdecstp();	Rotates the items on the FPU stack.

fdiv(mem);
fdiv(st*i*, st0);
fdiv(st0, st*i*);
	Floating point division. If a memory operand is present, it must be a *real32* or *real64* object; FDIV will divide ST0 by the memory operand and leave the quotient in ST0. If the FDIV operands are registers, FDIV diviides the destination (second) operand by the source (first) operand and leaves the result in the destination operand.

fdivp();
fdivp(st*i*);
	With no operands, this instruction divides ST1 by ST0, pops ST0, and replaces the new top of stack with the quotient (replacing the previous ST1 value).

fdivr(mem);
fdivr(st*i*, st0);
fdivr(st0, st*i*);
	Floating point divide with reversed operands. Like FDIV, but computes operand/ST0 rather than ST0/operand.

fdivrp();
fdivrp(st*i*);
	Floating point divide and pop, reversed. Like FDIVP except it computes operand/ST0 rather than ST0/operand.

ffree(st*i*);	Frees the specified floating point register.
fiadd(mem);	Memory operand must be a 16-bit or 32-bit signed integer. This instruction converts the integer to a real, pushes the value, and then executes FADDP();
ficom(mem);	Floating point compare to integer. Memory operand must be an *int16* or *int32* object. This instruction converts the memory operand to a *real80* value and compares ST0 to this value and sets the status bits in the floating point status register.
ficomp(mem);	Floating point compare to integer and pop. Memory operand must be an *int16* or *int32* object. This instruction converts the memory operand to a *real80* value and compares ST0 to this value and sets the status bits in the floating point status register. After the comparison, this instructions pop ST0 from the FPU stack.
fidiv(mem);	Floating point divide by integer. Memory operand must be an *int16* or *int32* object. These instructions convert their integer operands to a *real80* value and then divide ST0 by this value, leaving the result in ST0.

Table 2: Floating Point Instruction Set

Instruction	Description
fidivr(mem);	Floating point divide by integer, reversed. Like FIDIV above, except this instruction computes mem/ST0 rather than ST0/mem.
fild(mem);	Floating point load integer. Mem operand must be an int16 or int32 object. This instructions converts the integer to a real80 object and pushes it onto the FPU stack.
fimul(mem);	Floating point multiply by integer. Converts *int16* or *int32* operand to a *real80* value and multiplies ST0 by this result. Leaves product in ST0.
fincstp();	Rotates the registers on the FPU stack.
finit();	Initializes the FPU for use.
fist(mem);	Converts ST0 to an integer and stores the result in the specified memory operand. Memory operand must be an *int16* or *int32* object.
fistp(mem);	Floating point integer store and pop. Pops ST0 value off the stack, converts it to an integer, and stores the integer in the specified location. Memory operand must be a word, double word, or quad word (64-bit integer) object.
fisub(mem);	Floating point subtract integer. Converts *int16* or *int32* operand to a *real80* value and subtracts it from ST0. Leaves the result in ST0.
fisubr(mem);	Floating point subtract integer, reversed. Like FISUB except this instruction compute mem-ST0 rather than ST0-mem. Still leaves the result in ST0.

fld(mem);
fld(st*i*);
	Floating point load. Loads (pushes) the specified operand onto the FPU stack. Memory operands must be *real32*, *real64*, or *real80* objects. Note that FLD(ST0) duplicates the value on the top of the floating point stack.

fld1();	Floating point load 1.0. This instruction pushes 1.0 onto the FPU stack.
fldcw(mem16);	Load floating point control word. This instruction copies the word operand into the floating point control register.
fldenv(mem28);	This instruction loads the FPU status from the block of 28 bytes specified by the operand. Generally, only an operating system would use this instruction.
fldl2e();	Floating point load constant. Loads $\log_2(e)$ onto the stack.
fldl2t();	Floating point load constant. Loads $\log_2(10)$ onto the stack.
fldlg2();	Floating point load constant. Loads $\log_{10}(2)$ onto the stack.
fldln2();	Floating point load constant. Loads $\log_e(2)$ onto the stack.
fldpi();	Floating point load constant. Loads the value of pi (π) onto the stack.
fldz();	Floating point load constant. Pushes the value 0.0 onto the stack.

Table 2: Floating Point Instruction Set

Instruction	Description

fmul(mem);
fmul(st*i*, st0);
fmul(st0, st*i*);
	Floating point multiply. If the operand is a memory operand, it must be a real32 or real64 value; in this case, FMUL multiplies the memory operand and ST0, leaving the product in ST0. For the other two forms, the FMUL instruction multiplies the first operand by the second and leaves the result in the second operand.

fmulp();
fmulp(st0, st*i*);
	Floating point multiply and pop. With no operands this instruction computes ST1:=ST0*ST1 and then pops ST0. With two register operands, this instruction computes ST0 times the destination register and then pops ST0.

fnop();	Floating point no-operation.
fpatan();	Floating point partial arctangent. Computes ATAN(ST1/ST0), pops ST0, and then stores the result in the new TOS value (previous ST1 value).
fprem();	Floating point remainder. This instruction is retained for compatibility with older programs. Use the FPREM1 instruction instead.
fprem1();	Floating point partial remainder. This instruction computes the remainder obtained by dviding ST0 by ST1, leaving the result in ST0 (it does not pop either operand). If the C2 flag in the FPU status register is set after this instruction, then the computation is not complete; you must repeatedly execute this instruction until C2 is cleared.
fptan();	Floating point partial tangent. This instruction computes TAN(ST0) and replaces the value in ST0 with this result. Then it pushes 1.0 onto the stack. This instruction sets the C2 flag if the input value is outside the acceptable range of $\pm 2^{63}$.
frndint();	Floating point round to integer. This instruction rounds the value in ST0 to an integer using the rounding control bits in the floating point control register. Note that the result left on TOS is still a real value. It simply doesn't have a fractional component. You may use this instruction to round or truncate a floating point value by setting the rounding control bits appropriately. See the chapter on floating point arithmetic for details.
frstor(mem108);	Restores the FPU status from a 108-byte memory block.
fsave(mem108);	Writes the FPU status to a 108-bye memory block.
fscale();	Floating point scale by power of two. ST1 contains a scaling value. This instruction multiplies ST0 by 2^{st1}.
fsin();	Floating point sine. Replaces ST0 with sin(ST0).
fsincos();	Simultaneously computes the sin and cosine values of ST0. Replaces ST0 with the sine of ST0 and then it pushes the cosine of (the original value of) ST0 onto the stack. Original ST0 value must be in the range $\pm 2^{63}$.
fsqrt();	Floating point square root. Replaces ST0 with the square root of ST0.

Table 2: Floating Point Instruction Set

Instruction	Description

fst(mem);
fst(st*i*);
	Floating point store. Stores a copy of ST0 in the destination operand. Memory operands must be *real32* or *real64* objects. When storing the value to memory, FST converts the value to the smaller format using the rounding control bits in the floating point control register to determine how to convert the *real80* value in ST0 to a *real32* or *real64* value.

fstcw(mem16);	Floating point store control word. Stores a copy of the floating point control word in the specified *word* memory location.
fstenv(mem28);	Floating point store FPU environment. Stores a copy of the 28-byte floating point environment in the specified memory location. Normally, an OS would use this when switch contexts.

fstp(mem);
fstp(st*i*);
	Floating point store and pop. Stores ST0 into the destination operand and then pops ST0 off the stack. If the operand is a memory object, it must be a *real32, real64*, or *real80* object.

fstsw(ax);
fstsw(mem16);
	Stores a copy of the 16-bit floating point status register into the specified word operand. Note that this instruction automatically places the C1, C2, C3, and C4 condition bits in appropriate places in AH so that a following SAHF instruction will set the processor flags to allow the use of a conditional jump or conditional set instruction after a floating point comparison. See the chapter on floating point arithmetic for more details.

fsub(mem);
fsub(st0, st*i*);
fsub(st*i*, st0);
	Floating point subtract. With a single memory operand (which must be a real32 or real64 object), this instruction subtracts the memory operand from ST0. With two register operands, this instruction computes *dest := dest - src* (where *src* is the first operand and *dest* is the second operand).

fsubp();
fsubp(st0, st*i*);
	Floating point subtract and pop. With no operands, this instruction computes ST1 := ST0 - ST1 and then pops ST0 off the stack. With two operands, this instruction computes STi := ST*i* - ST0 and then pops ST0 off the stack.

fsubr(mem);
fsubr(st0, st*i*);
fsubr(st*i*, st0);
	Floating point subtract, reversed, With a real32 or real64 memory operand, this instruction computes ST0 := mem - ST0. For the other two forms, this instruction computes *dest := src - dest* where *src* is the first operand and *dest* is the second operand.

fsubrp();
fsubrp(st0, st*i*);
	Floating point subtract and pop, reversed. With no operands, this instruction computes ST1 := ST0 - ST1 and then pops ST0. With two operands, this instruction computes STi := ST0 - STi and then pops ST0 from the stack.

ftst();	Floating point test against zero. Compares ST0 with 0.0 and sets the floating point condition code bits accordingly.

Table 2: Floating Point Instruction Set

Instruction	Description

fucom(st*i*);
fucom();
	Floating point unordered comparison. With no operand, this instruction compares ST0 to ST1. With an operand, this instruction compares ST0 to STi and sets floating point status bits accordingly. Unlike FCOM, this instruction will not generate an exception if either of the operands is an illegal floating point value; instead, this sets a special status value in the FPU status register.

fucomi(st*i*, st0);	Floating point unordered comparison.

fucomp();
fucomp(st*i*);
	Floating point unorder comparison and pop. With no operands, compares ST0 to ST1 using an unordered comparsion (see FUCOM) and then pops ST0 off the stack. With an FPU operand, this instruction compares ST0 to the specified register and then pops ST0 off the stack. See FUCOM for more details.

fucompp();
fucompp(st*i*);
	Floating point unordered compare and double pop. Compares ST0 to ST1, sets the condition code bits (without raising an exception for illegal values, see FUCOM), and then pops both ST0 and ST1.

fwait();	Floating point wait. Waits for current FPU operation to complete. Generally an obsolete instruction. Used back in the days when the FPU was on a different chip than the CPU.
fxam();	Floating point Examine ST0. Checks the value in ST0 and sets the condition code bits according to the type of the value in ST0. See the chapter on floating point arithmetic for details.

fxch();
fxch(st*i*);
	Floating point exchange. With no operands this instruction exchanges ST0 and ST1 on the FPU stack. With a single register operand, this instruction swaps ST0 and STi.

fxtract();	Floating point exponent/mantissa extraction. This instruction breaks the value in ST0 into two pieces. It replaces ST0 with the real representation of the binary exponent (e.g. 2^5 becomes 5.0) and then it pushes the mantissa of the value with an exponent of zero.
fyl2x();	Floating point partial logarithm computation. Computes ST1 := ST1 * $\log_2$(ST0); and then pops ST0.
fyl2xp1();	Floating point partial logarithm computation. Computes ST1 := ST1 * $\log_2$(ST0 + 1.0) and then pops ST0 off the stack. Original ST0 value must be in the range $$(-\left(1-\frac{\sqrt{2}}{2}\right),\left(1-\frac{\sqrt{2}}{2}\right))$$

a. Floating point conditional move instructions are only available on Pentium Pro and later processors.
b. FCOMIx instructions are only available on Pentium Pro and later processors.

The following table uses these abbreviations:

Reg32-	A 32-bit general purpose (integer) register.
mm*i*-	One of the eight MMX registers, MM0..MM7.
imm8-	An eight-bit constant value; some instructions have smaller ranges that 0..255. See the particular instruction for details.
mem64-	A memory location (using an arbitrary addressing mode) that references a qword value.

Note: Most instructions have two operands. Typically the first operand is a source operand and the second operand is a destination operand. For exceptions, see the description of the instruction.

Table 3: MMX Instruction Set

Instruction	Description
emms();	Empty MMX State. You must execute this instruction when you are finished using MMX instructions and before any following floating point instructions.

movd(reg32, mm*i*);
movd(mem32, mm*i*);
movd(mm*i*, reg32);
movd(mm*i*, mem32);
	Moves data between a 32-bit integer register or dword memory location and an MMX register (mm0..mm7). If the destination operand is an MMX register, then the source operand is zero-extended to 64 bits during the transfer. If the destination operand is a dword memory location or 32-bit register, this instruction copies only the L.O. 32 bits of the MMX register to the destination.

movq(mem64, mm*i*);
movq(mm*i*, mem64);
movq(mm*i*, mm*i*);
	This instruction moves 64 bits between an MMX register and a *qword* variable in memory or between two MMX registers.

packssdw(mem64, mm*i*);
packssdw(mm*i*, mm*i*);
	Pack and saturate two signed double words from source and two signed double words from destination and store result into destination MMX register. This process involves taking these four double words and “saturating” them. This means that if the value is in the range -32768..32768 the value is left unchanged, but if it’s greater than 32767 the value is set to 32767 or if it’s less than -32768 the value is clipped to -32768. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.

packsswb(mem64, mm*i*);
packsswb(mm*i*, mm*i*);
	Pack and saturate four signed words from source and four signed words from destination and store the result as eight signed bytes into the destination MMX register. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.

Table 3: MMX Instruction Set

Instruction	Description

packusdw(mem64, mm*i*);
packusdw(mm*i*, mm*i*);
	Pack and saturate two unsigned double words from source and two unsigned double words from destination and store result into destination MMX register. This process involves taking these four double words and “saturating” them. This means that if the value is in the range 0..65535 the value is left unchanged, but if it’s greater than 65535 the value is clipped to 65535. The four double words are packed into a single 64-bit MMX register. The source operand supplies the upper two words and the destination operand supplies the lower two words of the packed 64-bit result. See the chapter on the MMX instructions for more details.

packuswb(mem64, mm*i*);
packuswb(mm*i*, mm*i*);
	Pack and saturate four unsigned words from source and four unsigned words from destination and store the result as eight unsigned bytes into the destination MMX register. Word values greater than 255 are clipped to 255 during the saturation operation. See the chapter on the MMX instructions for more details. The bytes obtained from the destination register wind up in the L.O. four bytes of the destination; the bytes computed from the signed saturation of the source register wind up in the H.O. four bytes of the destination register.

paddb(mem64, mm*i*);
paddb(mm*i*, mm*i*);
	Packed Add of Bytes. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

paddd(mem64, mm*i*);
paddd(mm*i*, mm*i*);
	Packed Add of Double Words. This instruction adds together the individual dwords of the two operands. The addition of each dword is independent of the other two dwords; there is no carry from dword to dword. If an overflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

paddsb(mem64, mm*i*);
paddsb(mm*i*, mm*i*);
	Packed Add of Bytes, signed saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at -128 or +127. This instruction does not affect any flags.

paddsw(mem64, mm*i*);
paddsw(mm*i*, mm*i*);
	Packed Add of Words, signed saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either -32768 or +32767. This instruction does not affect any flags.

paddusb(mem64, mm*i*);
paddusb(mm*i*, mm*i*);
	Packed Add of Bytes, unsigned saturated. This instruction adds together the individual bytes of the two operands. The addition of each byte is independent of the other eight bytes; there is no carry from byte to byte. If an overflow or underflow occurs in any byte, then the value saturates at 0 or 255. This instruction does not affect any flags.

Table 3: MMX Instruction Set

Instruction	Description

paddusw(mem64, mm*i*);
paddusw(mm*i*, mm*i*);
	Packed Add of Words, unsigned saturated. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow or underflow occurs in any word, the value saturates at either 0 or 65535. This instruction does not affect any flags.

paddw(mem64, mm*i*);
paddw(mm*i*, mm*i*);
	Packed Add of Words. This instruction adds together the individual words of the two operands. The addition of each word is independent of the other four words; there is no carry from word to word. If an overflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

pand(mem64, mm*i*);
pand(mm*i*, mm*i*);
	Packed AND. This instruction computes the bitwise AND of the source and the destination values, leaving the result in the destination. This instruction does not affect any flags.

pandn(mem64, mm*i*);
pandn(mm*i*, mm*i*);
	Packed AND NOT. This instruction makes a temporary copy of the first operand and inverts all of the bits in this copy; then it ANDs this value with the destination MMX register. This instruction does not affect any flags.

pavgb(mem64, mm*i*);
pavgb(mm*i*, mm*i*);
	Packed Average of Bytes. This instruction computes the average of the eight pairs of bytes in the two operands. It leaves the result in the destination (second) operand.

pavgw(mem64, mm*i*);
pavgw(mm*i*, mm*i*);
	Packed Average of Words. This instruction computes the average of the four pairs of words in the two operands. It leaves the result in the destination (second) operand.

pcmpeqb(mem64, mm*i*);
pcmpeqb(mm*i*, mm*i*);
	Packed Compare for Equal Bytes. This instruction compares the individual bytes in the two operands. If they are equal this instruction sets the corresponding byte in the destination (second) register to $FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero.

pcmpeqd(mem64, mm*i*);
pcmpeqd(mm*i*, mm*i*);
	Packed Compare for Equal Double Words. This instruction compares the individual double wordsin the two operands. If they are equal this instruction sets the corresponding double wordin the destination (second) register to $FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero.

pcmpeqw(mem64, mm*i*);
pcmpeqw(mm*i*, mm*i*);
	Packed Compare for Equal Words. This instruction compares the individual words in the two operands. If they are equal this instruction sets the corresponding word in the destination (second) register to $FFFF (all ones); if they are not equal, this instruction sets the corresponding word to zero.

Table 3: MMX Instruction Set

Instruction	Description

pcmpgtb(mem64, mm*i*);
pcmpgtb(mm*i*, mm*i*);
	Packed Compare for Greater Than, Bytes. This instruction compares the individual bytes in the two operands. If the destination (second) operand byte is greater than the source (first) operand byte, then this instruction sets the corresponding byte in the destination (second) register to $FF (all ones); if they are not equal, this instruction sets the corresponding byte to zero. Note that there is no PCMPLEB instruction. You can simulate this instruction by swapping the operands in the PCMPGTB instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.

pcmpgtd(mem64, mm*i*);
pcmpgtd(mm*i*, mm*i*);
	Packed Compare for Greater Than, Double Words. This instruction compares the individual dwords in the two operands. If the destination (second) operand dword is greater than the source (first) operand dword, then this instruction sets the corresponding dword in the destination (second) register to $FFFF_FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLED instruction. You can simulate this instruction by swapping the operands in the PCMPGTD instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.

pcmpgtw(mem64, mm*i*);
pcmpgtw(mm*i*, mm*i*);
	Packed Compare for Greater Than,Words. This instruction compares the individual words in the two operands. If the destination (second) operand word is greater than the source (first) operand word, then this instruction sets the corresponding word in the destination (second) register to $FFFF (all ones); if they are not equal, this instruction sets the corresponding dword to zero. Note that there is no PCMPLEW instruction. You can simulate this instruction by swapping the operands in the PCMPGTW instruction (i.e., compare in the opposite direction). Also note that these operands are, in a sense, backwards compared with the standard CMP instruction. This instruction compares the second operand to the first rather than the other way around. This was done because the second operand is always the destination operand and, unlike the CMP instruction, this instruction writes data to the destination operand.

Table 3: MMX Instruction Set

Instruction	Description
pextrw(imm8, mm*i*, reg32);	Packed Extraction of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the specified word from the MMX register into the L.O. word of the destination 32-bit integer register. This instruction zero extends the 16-bit value to 32 bits in the integer register. Note that there are no extraction instructions for bytes or dwords. However, you can easily extract a byte using PEXTRW and an AND or XCHG instruction (depending on whether the byte number is even or odd). You can use MOVD to extract the L.O. dword. to extract the H.O. dword of an MMX register requires a bit more work; either extract the two words and merge them or move the data to memory and grab the dword you're interested in.
pinsw(imm8, reg32, mm*i*);	Packed Insertion of a word. The imm8 value must be a constant in the range 0..3. This instruction copies the L.O. word from the 32-bit integer register into the specified word of the destination MMX register. This instruction ignores the H.O. word of the integer register.

pmaddwd(mem64, mm*i*);
pmaddwd(mm*i*, mm*i*);
	Packed Multiple and Accumulate (Add). This instruction multiplies together the corresponding words in the source and destination operands. Then it adds the two double word products from the multiplication of the two L.O. words and stores this double word sum in the L.O. dword of the destination MMX register. Finally, it adds the two double word products from the multiplication of the H.O. words and stores this double word sum in the H.O. dword of the destination MMX register.

pmaxw(mem64, mm*i*);
pmaxw(mm*i*, mm*i*);
	Packed Signed Integer Word Maximum. This instruction compares the four words between the two operands and stores the signed maximum of each corresponding word in the destination MMX register.

pmaxub(mem64, mm*i*);
pmaxub(mm*i*, mm*i*);
	Packed Unsigned Byte Maximum. This instruction compares the eight bytes between the two operands and stores the unsigned maximum of each corresponding byte in the destination MMX register.

pminw(mem64, mm*i*);
pminw(mm*i*, mm*i*);
	Packed Signed Integer Word Minimum. This instruction compares the four words between the two operands and stores the signed minimum of each corresponding word in the destination MMX register.

pminub(mem64, mm*i*);
pminub(mm*i*, mm*i*);
	Packed Unsigned Byte Minimum. This instruction compares the eight bytes between the two operands and stores the unsigned minimum of each corresponding byte in the destination MMX register.

pmovmskb(mm*i*, reg32);	Move Byte Mask to Integer. This instruction creates a byte by extracting the H.O. bit of the eight bytes from the MMX source register. It zero extends this value to 32 bits and stores the result in the 32-bit integer register.

pmulhuw(mem64, mm*i*);
pmulhuw(mm*i*, mm*i*);
	Packed Multiply High, Unsigned Words. This instruction multiplies the four unsigned words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.

Table 3: MMX Instruction Set

Instruction	Description

pmulhw(mem64, mm*i*);
pmulhw(mm*i*, mm*i*);
	Packed Multiply High, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the H.O. word of the resulting products into the corresponding word of the destination MMX register.

pmullw(mem64, mm*i*);
pmullw(mm*i*, mm*i*);
	Packed Multiply Low, Signed Words. This instruction multiplies the four signed words of the two operands together and stores the L.O. word of the resulting products into the corresponding word of the destination MMX register.

por(mem64, mm*i*);
por(mm*i*, mm*i*);
	Packed OR. Computes the bitwise OR of the two operands and stores the result in the destination (second) MMX register.

psadbw(mem64, mm*i*);
psadbw(mm*i*, mm*i*);
	Packed Sum of Absolute Differences. This instruction computes the absolute value of the difference of each of the unsigned bytes between the two operands. Then it adds these eight results together to form a word sum. Finally, the instruction zero extends this word to 64 bits and stores the result in the destination (second) operand.

pshufw(imm8,mem64,mm*i*);	Packed Shuffle Word. This instruction treats the imm8 value as an array of four two-bit values. These bits specify where the destination (third) operand's words obtain their values. Bits zero and one tell this instruction where to obtain the L.O. word, bits two and three specify where word #1 comes from, bits four and five specify the source for word #2, and bits six and seven specify the source of the H.O. word in the destination operand. Each pair of bytes specifies a word number in the source (second) operand. For example, an immediate value of %00011011 tells this instruction to grab word #3 from the source and place it in the L.O. word of the destination; grab word #2 from the source and place it in word #1 of the destination; grab word #1 from the source and place it in word #2 of the destination; and grab the L.O. word of the source and place it in the H.O. word of the destination (i.e., swap all the words in a manner similar to the BSWAP instruction).

pslld(mem, mm*i*);
pslld(mm*i*, mm*i*);
pslld(imm8, mm*i*);
	Packed Shift Left Logical, Double Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the L.O. dword to the H.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.

psllq(mem, mm*i*);
psllq(mm*i*, mm*i*);
psllq(imm8, mm*i*);
	Packed Shift Left Logical, Quad Word. This instruction shifts the destination operand to the left the number of bits specified by the first operand.

psllw(mem, mm*i*);
psllw(mm*i*, mm*i*);
psllw(imm8, mm*i*);
	Packed Shift Left Logical,Words. This instruction shifts the destination (second) operand to the left the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the L.O. words into the next higher word. Bits shifted out are lost and this instruction always shifts in zeros.

Table 3: MMX Instruction Set

Instruction	Description

psard(mem, mm*i*);
psard(mm*i*, mm*i*);
psard(imm8, mm*i*);
	Packed Shift Right Arithmetic, Double Word. This instruction treats the two halves of the 64-bit register as two double words and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the two double words is lost.

psarw(mem, mm*i*);
psarw(mm*i*, mm*i*);
psarw(imm8, mm*i*);
	Packed Shift Right Arithmetic, Word. This instruction operates independently on the four words of the 64-bit destination register and performs separate arithmetic shift rights on them. The bit shifted out of the bottom of the four words is lost.

psrld(mem, mm*i*);
psrld(mm*i*, mm*i*);
psrld(imm8, mm*i*);
	Packed Shift Right Logical, Double Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Each double word in the destination is treated as an independent entity. Bits are not carried over from the H.O. dword to the L.O. dword. Bits shifted out are lost and this instruction always shifts in zeros.

pslrq(mem, mm*i*);
pslrq(mm*i*, mm*i*);
pslrq(imm8, mm*i*);
	Packed Shift Right Logical, Quad Word. This instruction shifts the destination operand to the right the number of bits specified by the first operand.

pslrw(mem, mm*i*);
pslrw(mm*i*, mm*i*);
pslrw(imm8, mm*i*);
	Packed Shift Right Logical,Words. This instruction shifts the destination (second) operand to the right the number of bits specified by the first operand. Bits shifted out are lost and this instruction always shifts in zeros. Each word in the destination is treated as an independent entity. Bits are not carried over from the H.O. words into the next lower word. Bits shifted out are lost and this instruction always shifts in zeros.

psubb(mem64, mm*i*);
psubb(mm*i*, mm*i*);
	Packed Subtract of Bytes. This instruction subtracts the individual bytes of the source (first) operand from the corresponding bytes of the destination (second) operand. The subtraction of each byte is independent of the other eight bytes; there is no borrow from byte to byte. If an overflow or underflow occurs in any byte, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

psubd(mem64, mm*i*);
psubd(mm*i*, mm*i*);
	Packed Subtract of Double Words. This instruction subtracts the individual dwords of the source (first) operand from the corresponding dwords of the destination (second) operand. The subtraction of each dword is independent of the other; there is no borrow from dword to dword. If an overflow or underflow occurs in any dword, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

psubsb(mem64, mm*i*);
psubsb(mm*i*, mm*i*);
	Packed Subtract of Bytes, signed saturated. This instruction subracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to -128 or +127 if overflow or underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.

Table 3: MMX Instruction Set

Instruction	Description

psubsw(mem64, mm*i*);
psubsw(mm*i*, mm*i*);
	Packed Subtract of Words, signed saturated. This instruction subracts the individual words of the source operand from the corresponding words of the destination operand, saturating to -32768 or +32767if overflow or underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.

psubusb(mem64, mm*i*);
psubusb(mm*i*, mm*i*);
	Packed Subtract of Bytes, unsigned saturated. This instruction subracts the individual bytes of the source operand from the corresponding bytes of the destination operand, saturating to 0 if underflow occurs. The subtraction of each byte is independent of the other seven bytes; there is no carry from byte to byte. This instruction does not affect any flags.

psubusw(mem64, mm*i*);
psubusw(mm*i*, mm*i*);
	Packed Subtract of Words, unsigned saturated. This instruction subracts the individual words of the source operand from the corresponding words of the destination operand, saturating to 0 if underflow occurs. The subtraction of each word is independent of the other three words; there is no carry from word to word. This instruction does not affect any flags.

psubw(mem64, mm*i*);
psubw(mm*i*, mm*i*);
	Packed Subtract of Words. This instruction subtracts the individual words of the source (first) operand from the corresponding words of the destination (second) operand. The subtraction of each word is independent of the others; there is no borrow from word to word. If an overflow or underflow occurs in any word, the value simply wraps around to zero with no indication of the overflow. This instruction does not affect any flags.

punpckhbw(mem64, mm*i*);
punpckhbw(mm*i*, mm*i*);
	Unpack high packed data, bytes to words. This instruction unpacks and interleaves the high-order four bytes of the source (first) and destination (second) operands. It places the H.O. four bytes of the destination operand at the even byte positions in the destination and it places the H.O. four bytes of the source operand in the odd byte positions of the destination operand.

punpckhdq(mem64, mm*i*);
punpckhdq(mm*i*, mm*i*);
	Unpack high packed data, dwords to qword. This instruction copies the H.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) H.O. dword of the destination operand to the L.O. dword of the destination.

punpckhwd(mem64, mm*i*);
punpckhwd(mm*i*, mm*i*);
	Unpack high packed data, words to dwords. This instruction unpacks and interleaves the high-order two words of the source (first) and destination (second) operands. It places the H.O. two words of the destination operand at the even word positions in the destination and it places the H.O. words of the source operand in the odd word positions of the destination operand.

punpcklbw(mem64, mm*i*);
punpcklbw(mm*i*, mm*i*);
	Unpack low packed data, bytes to words. This instruction unpacks and interleaves the low-order four bytes of the source (first) and destination (second) operands. It places the L.O. four bytes of the destination operand at the even byte positions in the destination and it places the L.O. four bytes of the source operand in the odd byte positions of the destination operand.

Table 3: MMX Instruction Set

Instruction	Description

punpckldq(mem64, mm*i*);
punpckldq(mm*i*, mm*i*);
	Unpack low packed data, dwords to qword. This instruction copies the L.O. dword of the source operand to the H.O. dword of the destination operand and it copies the (original) L.O. dword of the destination operand to the L.O. dword of the destination (i.e., it doesn't change the L.O. dword of the destination).

punpcklwd(mem64, mm*i*);
punpcklwd(mm*i*, mm*i*);
	Unpack low packed data, words to dwords. This instruction unpacks and interleaves the low-order two words of the source (first) and destination (second) operands. It places the L.O. two words of the destination operand at the even word positions in the destination and it places the L.O. words of the source operand in the odd word positions of the destination operand.

pxor(mem64, mm*i*);
pxor(mm*i*, mm*i*);
	Packed Exclusive-OR. This instruction exclusive-ORs the source operand with the destination operand leaving the result in the destination operand.

The HLA Language Reference — Appendix E

To be written when the HLA language settles down a little bit.

In the meantime, please consult the HLA Language Reference Manual (a separate document that is part of the HLA distribution).

The HLA Standard Library Reference Appendix F

To be written when the HLA language settles down a bit. See the HLA Standard Library documentation in the meantime.

HLA Exceptions Appendix G

The HLA Standard Library provides the following exception types[1]:

ex.StringOverflow

HLA raises this exception if you attempt to store too many characters into preallocated string variable. The following standard library routines can raise this exception:

pat.extract, strrealloc, stdin.gets, str.cpy, str.setstr, str.cat, str.substr, str.insert, console.gets, cStrToStr, dToStr, e80ToStr, hToStr, u64ToStr, i64ToStr, qToStr, r80ToStr, tbToStr, wToStr, date.print, date.toString, and date.a_toString.

ex.StringIndexError

HLA raises this exception if a routine attempts to use an index that is beyond the last valid character in a string. The following standard library routines can raise this exception:

str.span2, str.rspan2, str.brk2, str.rbrk2, str.substr, str.a_substr, strToFlt, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou32, StrTou64, StrToh, StrTow, StrTod, and StrToq.

ex.ValueOutOfRange

HLA raises this exception if an arithmetic overflow occurs, if an input parameter is out of range, or if user input is too great for the destination variable. The following standard library routines can raise this exception:

arg.v, arg.delete, rand.urange, rand.range, stdin.geti8, stdin.geti16, stdin.geti32, stdin.geti64, stdin.getu8, stdin.getu16, stdin.getu32, stdin.getu64, stdin.geth, stdin.getw, stdin.getd, stdin.getq, stdin.getf, table.create, console.a_getRect, console.fillRect, console.fillRectAttr, console.getc, console.getRect, console.gets, console.gotoxy, console.putRect, console.scrollDnRect, console.scrololUpRect, atof, atoh, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, atou64, e80ToStr, r80ToStr, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou32, StrTou64, StrToh, StrTow, StrTod, StrToq, fileio.getd, fileio.geth, fileio.getw, fileio.getq, fileio.geti8, fileio.geti16, fileio.geti32, fileio.geti64, fileio.getu8, fileio.getu16, fileio.getu32, fileio.getu64, fileio.pute80pad, fileio.pute64pad, fileio.pute32pad, fileio.putr32Pad, fileio.putr64Pad, and fileio.putr80Pad.

ex.IllegalChar

Several HLA routines raise this exception if they encounter a non-ASCII character (character code $80..$FF) where a delimiter character is expected. Generally, you can treat this error as though it were a conversion error. Routines that raise this exception include:

atoh, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, and atou64.

ex.ConversionError

Routines in the HLA Standard Library raise this exception if there is an error convertion data from one format to another. Typically this occurs when converting strings to numeric data. Routines that raise this exception include:

1. Please note that the HLA Standard Library is under constant revision and the list appearing in this chapter may be slightly out of date. Please consult the HLA Standard Library documentation for an up-to-date listing of exceptions and the routines that raise them.

stdin.geti8, stdin.geti16, stdin.geti32, stdin.geti64, stdin.getu8, stdin.getu16, stdin.getu32, stdin.getu64, stdin.geth, stdin.getw, stdin.getd, stdin.getq, stdin.getf, atof, atoh, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, atou64, fgetf, fileio.geti8, fileio.geti16, fileio.geti32, fileio.geti64, fileio.getu8, fileio.getu16, fileio.getu32, fileio.getu64, fileio.geth, fileio.getw, fileio.getd, and fileio.getq.

ex.BadFileHandle

The file class method file.handle raises this exception if the handle associated with a file class object is illegal (has not been initialized).

ex.FileOpenFailure

The fileio.Open, fileio.OpenNew, file.Open, and file.OpenNew procedures raise this exception is there is an error opening a file.

ex.FileCloseError

The fileio.Close and file.Close procedures raise this exception if there is some sort of error when attempting to close a file.

ex.FileWriteError

Those routines that write data to a file (e.g., fputi8) raise this exception if there is an error writing data to the output file.

ex.FileReadError

Those routines that read data from a file (e.g., fileio.geti8) raise this exception if there is a physical error reading the data from the file.

ex.DiskFullError

Those routines that write data to a file will raise this exception if an attempt is made to write data to a full disk.

ex.EndOfFile

Those routines that read data from a file will raise this exception if your program attempts to read data beyond the end of the file.

ex.MemoryAllocationFailure

Routines that allocation storage (e.g., malloc and realloc) will raise this exception if Windows cannot satisify the memory allocaiton requestion. Several routines in the standard library may raise this exception since they indirectly call the HLA *malloc* routine. Examples include *stdin.a_gets* and almost any other Standard Library routine that has “a_” as a prefix to the name.

ex.AttemptToDerefNULL

Several routines in the Standard Library that expect a string pointer will raise this exception if the string pointer (or other pointer) contains NULL (zero). Examples include:

getf, str.cpy, str.a_cpy, str.setstr, str.cat, str.a_cat, str.index, str.rindex, str.chpos, str.rchpos, str.span, str.span2, str.rspan, str.rspan2, str.brk, str.brk2, str.rbrk, str.rbrk2, str.eq, str,ne, str.lt, str.le, str,gt, str.ge, str.substr, str.a_substr, str.insert, str.a_insert, str.delete, str.a_delete, str.ieq, str.ine, str.ilt, str.ile, str.igt, str.ige, str.upper, str.a_upper, str.lower, str.a_lower, str.delspace, str.a_delspace, str.trim, str.a_trim, str.tokenize, str.tokenize2, atof, atoi8, atoi16, atoi32, atoi64, atou8, atou16, atou32, atou64, dtostr, htostr, wtostr, qtostr, strToFlt, StrToi8, StrToi16, StrToi32, StrToi64, StrTou8, StrTou16, StrTou64, StrToh, StrTow, StrTod, StrToq, and tbtostr.

ex.WidthTooBig

HLA routines that print a numeric value within a field width will raise this exception if the specified width exceeds 1,024 characters. Routines that raise this exception include *stdout.pu*t, *stdout.puti8Size*, *fputu32Size*, and all other *xxxxSize* output routines.

ex.TooManyCmdLnParms

The *arg.CmdLn* procedure raises this exception if it determines that there are more than 256 command line parameters on the command line (a virtual impossibility since command lines are generally limited to 128 characters). Since all of the other routines in the *args* module can call arg.CmdLn, it is possible for any of the routines in this module to raise this exception.

ex.ArrayShapeViolation

Routines in the arrays module raise this exception if the dimension on some array are inappropriate for the specified operation. For example, the array.cpy code will raise this exception if you attempt to copy a source array to a destination whose dimensions don't exactly match the source array.

ex.InvalidDate

The routines in the datetime module will raise this exception if you pass them an illegal date value as a parameter.

ex.InvalidDateFormat

The date output routines (date.print and date.toString) raise this exception if you attempt convert a date to a string but the current (internal) date format variable contains an invalid value. This usually implies that you've passed an incorrect parameter to the *date.SetFormat* procedure.

ex.TimeOverflow

The time.secsToHMS procedure raises this overflow if there is an error converting time in seconds to hours, minutes, and seconds (very rare, only occurs above two billion seconds).

ex.AccessViolation

This is a hardware exception that Windows raise if your program attempts to access an illegal memory location (generally a NULL reference or an uninitialized pointer).

ex.Breakpoint

This exception is raised by Windows for debugger programs; you should never see this exception unless you are writing a debugger program.

ex.SingleStep

This is another exception that is raised by Windows for debugger programs; you should never see this exception unless you are writing a debugger program.

ex.PrivInstr

Windows raises this instruction if you attempt to execute a special instruction that is illegal in "user mode" (that is, can only be executed by Windows). Since HLA only compiles a few priviledged instructions, and this book doesn't discuss them at all, the only way you'll probably see this exception occur is if your program jumps off into (non-code) memory somewhere and begins executing data as instructions.

ex.IllegalInstr

Windows raises this exception if the CPU attempts to execute some code that is not a valid instruction. This generally implies that your program has jumped off into data memory and is attempting to execute data as machine instructions.

ex.BoundInstr

Windows raises this exception if you execute the BOUND instruction (see "Some Additional Instructions: INTMUL, BOUND, INTO" on page 393) and the register value is outside the specified range.

ex.IntoInstr

Windows raises this exception if you execute the INTO instruction and the overflow flag is set (see "Some Additional Instructions: INTMUL, BOUND, INTO" on page 393).

ex.DivideError

Windows raises this exception if you attempt an integer division by zero, or if the quotient of a division is too large to fit within the destination operand(AL, AX, or EAX).

ex.fDenormal

ex.fDivByZero

ex.fInexactResult

ex.fInvalidOperation

ex.fOverflow

ex.fStackCheck

ex.fUnderflow

Windows raises one of these exceptions if you've enabled exceptions on the FPU and one of the specified conditions occurs (e.g., a floating point division by zero will raise the *ex.fDivByZero* exception).

InvalidHandle

Windows will raise this exception if you pass an uninitialized or otherwise invalid handle value to a Windows API (application programmer's interface) routine. Many of the HLA Standard Library routines pass handles to Windows, so this exception could occur as a result of a call to an input/output routine.

StackOverflow

Windows raises this exception if the stack exceeds the storage allocated to it (16Mbytes in a typical HLA program).

ControlC

HLA raises this exception if the user presses control-C or control-Break during program execution.

HLA Compile-Time Functions — Appendix H

H.1 Conversion Functions

The conversion functions translate data from one format to another. For example, functions in this group can convert integers to strings or strings to integers. These routines provide the compile-time equivalent of the HLA Standard Library CONV module.

The compile-time conversion routines are unusual in the set of compile-time function insofar as they do not require a leading "@" symbol. Instead, the conversion routines use the names of several of the built-in data types. The following table describes each of these functions:

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
boolean	boolean(*constExpr*) ConstExpr can be a boolean, integer, character, or string operand.	If constExpr is numeric, this function returns false for zero and true for any other value. For characters, "t" or "f" returns true or false (respectively), anything else is an error. For strings, the operands must be "true" or "false" (else an error occurs). The boolean function returns boolean values unchanged and returns an error for any other type.
int8 int16 int32 uns8 uns16 uns32 byte word dword	*xxxx*(*constExpr*) Note: *xxxx* represents one of the function names to the left. *constExpr* can be any constant expression that evaluates to a numeric, character, boolean, or string operand.	These functions will convert their operand to the specified data type. These functions generate an error if the resulting value will not fit in the specified data type (e.g., int8(-1000) will generate an error). Note that HLA treats *byte*, *word*, and dword functions identically to *uns8*, *uns16*, and *uns32* (respectively). For boolean operands, true returns one and false returns zero. If the operand is a real value, then these functions truncate the value to obtain the corresponding integer return value. For character operands, these function return the corresponding ASCII code of the character. For string operands, the string must be a legal sequence of characters that form a decimal number.

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
real32 real64 real80	*xxxxxx*(*constExpr*) Note: *xxxxxx* represents one of the function names to the left. *constExpr* can be any constant expression that evaluates to a numeric or string operand.	These functions convert their specified operand to the corresponding real value. These functions convert integer operands to the corresponding real value. If the operand is a string expression, it must be a valid sequence of characters that corresponds to an HLA floating point value. These functions convert that string to the corresponding real value.
char	char(*constExpr*) *constExpr* must be a positive integer value, a character, or a string.	If the parameter is an integer value, this function returns the character with that ASCII code. If the parameter is a string, this function returns the first character of that string. If the operand is a character, this function simply returns that character value.
string	string(*constExpr*) *constExpr* can be any legal constant data type.	This function returns the string representation of the specified parameter. For real values, this function returns the scientific notation format for the value. For boolean expressions, this function returns the value "true" or "false". For character operands, this function returns a string containing the single character specified as the operand. For integer parameters, this function returns a string containing the decimal equivalent of that value. For character set operands, this function returns a string listing all the characters in the character set. If the operand is a string expression, this function simply returns that string.
cset	cset(*constExpr*) *constExpr* can be a character, string, or a cset.	This function returns a character set containing the characters specified by the operand. If the operand is a character, then this function returns the singleton set containing that single character. If the operand is a string, this function returns the union of all the characters in that string. If the operand is a character set, this function returns that character set.

Table 1: Compile-Time Data Conversion Functions

Function	Parameters[a]	Description
text	text(*constExpr*) *constExpr*: same as for string.	This function takes the same parameters as the string function. Instead of returning a string constant, however, this function expands the text in-line at the point of the function. This function is equivalent to the *@text* function.
@odd	@odd(*constExpr*) *constExpr* must be an integer value.	This function returns true if the integer operand is an odd number, it returns false otherwise.

a. Integer operands can be any of the *intXX*, *unsXX*, *byte*, *word*, or *dword* types.

Internally, HLA generally maintains all numeric constant values as *int32, uns32, dword*, or *real80*. Therefore, it is unlikely that you would want to use the *int8*, *int16*, *uns8*, *uns16*, *real32,* or *real64* compile-time functions in your program unless you want to force an error if a value is out of range.

Of these conversion functions, the *string* function is, perhaps, the most useful. Many compile-time functions and statements accept only string operands. You can use the *string* function to translate other data types to a string in a situation where you wish to use one of these other data types. For example, the #ERROR statement only allows a single string parameter. However, you can construct complex error messages by using the string concatentation operator ("+") with the *string* function, e.g.,

```
#error( "Constant value i32=" + string(i32) + " and that is out of range" )
```

H.2 Numeric Functions

These functions provide common mathematical functions. Remember, these functions compute their values at compile-time. Their parameters are constants and they return constant values. These functions are not useable with variables at run-time. See the HLA Standard Library for comparable functions you can call from your assembly language programs while they are running.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@abs	@abs(*constExpr*) *constExpr* must be a numeric value.	Returns the absolute value of the parameter. The return type is the same as the parameter type.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@byte	@byte(*Expr*, *select*) *Expr* must be a 32-bit ordinal expression, a real expression (32, 64, or 80 bits), or a *cset* expression. *select* must be less than the size of *Expr's* data type.	This function exacts the specified byte from the value of *Expr* as selected via the *select* parameter. If *select* is zero, this function returns the L.O. byte, higher values for select return the corrsponding higher-order bytes of the object. Note that HLA usually extends literal constants to the largest representation possible, e.g., HLA treats 1.234 as a real80 value. Use coercion to change this, if necessary (e.g., @byte(real32(1.234), 3))
@ceil	@ceil(*constExpr*) *constExpr* must be a numeric value.	If the parameter is an integer value, this function simply converts it to a real value and returns that value. If the parameter is a real value, then this function returns the smallest integer value larger than or equal to the parameter's value (i.e., this function rounds a real value to the next highest integer if the real value contains a fractional part).
@cos	@cos(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the cosine of the specified parameter value.
@exp	@exp(constExpr) *constExpr* must be a numeric value.	This function return *e* raised to the specified power.
@floor	@floor(*constExpr*) *constExpr* must be a numeric value.	This function returns the largest integer value that is less than or equal to the numeric value passed as a parameter. For positive real numbers this is equivalent to truncation (for negative numbers, it rounds towards negative infinity).
@log	@log(*constExpr*) *constExpr* is a non-negative numeric value.	This function computes the natural (base *e*) logarithm of its operand.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@log10	@log10(*constExpr*) *constExpr* is a non-negative numeric value.	This function computes the base-10 logarithm of its operand.
@max	@max(*list*) *list* is a list of two or more comma separated numeric expressions.	This function returns the maximum of a set of numeric values. The values in the list must all be the same type.
@min	@min(*list*) *list* is a list of two or more comma separated numeric expressions.	This function returns the minimum of a set of numeric values. The values in the list must all be the same type.
@random	@random(*intExpr*) *intExpr* must be a positive integer value.	This function returns a pseudo-random number between zero and *intExpr-1*. Currently HLA uses the random function provided by the C standard library; so don't expect a high quality random number generator here. In particular, if *intExpr* is a small value, the quality of the random number generator is very low.
@randomize	@randomize(*intExpr*) *intExpr* must be a positive integer value.	This function attempts to "randomize" the random number generator seed. Then it returns a random number between zero and *intExpr-1*. You should not call this function multiple times in your source file.
@sin	@sin(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the sine of the specified parameter value.
@sqrt	@sqrt(constExpr) *constExpr* m ust be a non-negative numeric value.	This function returns the square root of the specified operand value.

Table 2: Numeric Compile-Time Functions

Function	Parameters[a]	Description
@tan	@tan(*constExpr*) *constExpr* must be a numeric value expressing an angle in radians.	This function returns the tangent of the specified parameter value.

a. Numeric parameters are *intX, unsX, byte, word, dword,* or *realX* values.

H.3 Date/Time Functions

The Date/Time compile-time functions return strings holding the current date and time.

Table 3: HLA Compile-Time Date/Time Functions

Function	Parameters	Description
@date	@date	This function returns a string specifying the current date. This string typically takes the form "year/month/day", e.g., "2000/12/31".
@time	@time	This function returns a string specifying the current time. This string typically takes the form "HH:MM:SS xM" (x= A or P).

H.4 Classification Functions

These functions test a character or string to see if the characters belong to a certain class (e.g., alphabetic characters). This functions return true or false depending upon the result of the comparison.

If the operand is a character expression, these functions return true if that character belongs to the specified class. If the operand is a string, these functions return true if all characters in the string belong to the specified class.

Also see the HLA Compile-Time Pattern Matching Functions for some different ways to test characters in a string. Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isalpha	@isalpha(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an alphabetic character. If the parameter is a string, this function returns true if all characters in the string are alphabetic.
@isalphanum	@isalphanum(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an alphanumeric character. If the parameter is a string, this function returns true if all characters in the string are alphanumeric.
@isdigit	@isdigit(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a decimal digit character. If the parameter is a string, this function returns true if all characters in the string are digits.
@islower	@islower(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a lower case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are lower case alphabetic characters.
@isspace	@isspace(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a space character[a]. If the parameter is a string, this function returns true if all characters in the string are spaces.
@isupper	@isupper(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is an upper case alphabetic character. If the parameter is a string, this function returns true if all characters in the string are upper case alphabetic characters

Table 4: HLA Compile-Time Character Classification Functions.

Function	Parameters	Description
@isxdigit	@isxdigit(*constExpr*) *constExpr* must be a character or a string expression.	This function returns true if the character operand is a hexadecimal digit character {0-9, a-f, A-F}. If the parameter is a string, this function returns true if all characters in the string are hexadecimal digits.

a. "Space" means any white space character. This includes tabs, newlines, etc.

H.5 String and Character Set Functions

The following functions provide a very powerful set of string manipulation functions to the HLA compile-time language. These functions let you easily maniipulate data like macro parameters and #text..#endtext blocks.

Remember, these are compile-time functions. The HLA Standard Library contains comparable routines for use at run-time in your programs. See the HLA Standard Library documentation for more details.

The @extract function behaves a little differently than the *cs.extract* function in the HLA Standard Library. @extract does not actually remove the specified character from the character set. Keep this in mind if you use both @extract and *cs.extract* frequently.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@delete	@delete(*strExpr, start, len*) strExpr must be a string expression. *start* and *len* must be positive integer expressions.	This function returns a string consisting of the strExpr parameter with *len* characters removed starting at position *start* in the string. The first character in the string is at position zero. Therefore, @delete("Hello", 2, 3) returns the string "He".

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description

@extract	@extract(*csetExpr*)
	
	csetExpr must be a character set value.
		This function returns an arbitrary character from the specified character set. Note that this function does not remove that character from the set. You must manually remove the character if you do not want the next call to @extract (with the same parameter) to return the same character, e.g.,
		
		val
		    c := @extract(someSet);
		    someSet := someSet - {c};

@index	@index(*strExpr, start, findStr*) *strExpr* and *findStr* must be string expressions. *start* must be a non-negative integer value.	This function searches for the string specified by *findStr* within the *strExpr* string starting at character position *start*. If this function finds the string, it returns the index into *strExpr* where it located *findStr.* If it does not find the string, it returns -1.
@insert	@insert(*destStr, start, strToIns*) *destStr* and *strToIns* must be string expressions. *start* must be a non-negative integer expression.	This function returns a string consisting of the combination of the *destStr* and *strToIns* strings. This function inserts *strToIns* into *destStr* at the position specified by *start*.
@length	@length(*strExpr*) *strExpr* must be a string expression.	This function returns the length of the specified string as an integer result.
@lowercase	@lowercase(*strExpr, start*) *strExpr* must be a string expression. *start* must be a non-negative integer expression.	This function translates all characters from position *start* to the end of the string to lower case (if they were previously upper case alphabetic characters).

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@rindex	@rindex(*strExpr, start, findStr*) *strExpr* and *findStr* must be string expressions. *start* must be a non-negative integer value.	This function searches backwards to find the last occurrence of the string specified by *findStr* within the *strExpr* string. This function will only search back to position *start*. If this function finds the string, it returns the index into *strExpr* where it located *findStr.* If it does not find the string, it returns -1.
@strbrk	@strbrk(*strExpr, start, csetExpr*) *strExpr* must be a string expression. *start* must be a non-negative integer expression. *csetExpr* must be a character set expression.	Starting at position *start* within *strExpr*, this function searches for the first character of *strExpr* that is a member of the *csetExpr* character set. It returns -1 if no such characters exist.
@strset	@strset(*charExpr, len*) *charExpr* must be a character value. *len* must be a non-negative integer value.	This function returns the string consisting of *len* copies of *charExpr* concatenated together.
@strspan	@strspan(*strExpr, start, csetExpr*) *strExpr* must be a string expression. *start* must be a non-negative integer expression. *csetExpr* must be a character set expression.	Starting at position *start* within *strExpr*, this function searches for the first character of *strExpr* that is not a member of the *csetExpr* character set. It returns the length of the string if all remaining characters are in the specified character set.
@substr	@substr(*strExpr, start, len*) *strExpr* must be a string expression. *start* and *len* must be non-negative integer values.	This function returns the sequence of *len* characters (the "substring") starting at position *start* in the *strExpr* string.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@tokenize	@tokenize(*strExpr, start, Delims, Quotes*) *strExpr* must be a string expression. *start* must be a non-negative integer value. *Delims* and *Quotes* must be character set expressions.	This function returns an array of strings consisting of the various substrings in *strExpr* obtained by separating the substrings using the *Delims* character set. This "lexical scan" operation begins at position *start* in *strExpr.* The function first skips over all characters in *Delims* and the collects all characters until it finds another delimiter character from the *Delims* set. It repeats this process, adding each scanned string to the array it returns, until it reaches the end of the string. The fourth parameter, *Quotes*, specifies special quoting characters. Typically, this character set is the empty set. However, if it contains a character, then that character is a quote character and all characters between two occurrences of the quote symbol constitute a single string, even if delimiters appear within that string. Typical characters in the *Quotes* character set would be apostrophes or quotation marks. If "(", "[", or "{" appears in the *Quotes* set, then the corresponding closing symbol must also appear and tokenize uses the pair of quote objects to surround a quoted item. This function is unusual insofar as it returns an array constant as its result. Typically you would assign this to a VAL or CONST object so you can gain access to the individual items in the array. To determine the number of elements in this array, use the @elements function.

Table 5: HLA Compile-Time String Functions

Function	Parameters	Description
@trim	@trim(*strExpr*) *strExpr* must be a string expression.	This function returns the specified string operand with leading and trailing spaces removed.
@uppercase	@uppercase(*strExpr*) *strExpr* must be a string expression.	The function returns a copy of its string operand with any lower case alphabetic characters converted to upper case.

H.6 Pattern Matching Functions

The HLA compile-time language provides a large set of pattern matching functions similar to the run-time functions available in the PATTERNS.HHF module of the HLA Standard Library. These pattern matching functions, combined with the string processing functions of the previous section, provide you with the ability to write your own scanners and parsers (i.e., compilers) within the HLA compile-time language. Indeed, one of the primary purposes of the HLA compile-time language is to let you expand the HLA language to suit your needs. The pattern matching functions, along with macros, provide the backbone for this capability in HLA.

Note that there are some pretty serious differences between the HLA compile-time pattern matching functions and the routines in the PATTERNS.HHF module. Perhaps the biggest difference is the fact that the compile-time functions do not support backtracking while the run-time routines do. Fortunately, it is not that difficult to simulate backtracking on your own within the compile-time language.

Another difference between the comile-time functions and their run-time counterparts is the parameter list. The compile-time functions typically have a couple of optional parameters that let you extract information about the pattern match if it was successful. Consider, for a moment, the @matchStr function:

```
@matchStr( Str, tstStr )
```

When you call this function using the syntax above, this function will return true if the string expression *Str* begins with the sequence of characters found in *tstStr.* It returns false otherwise. Now consider the following invocation:

```
@matchStr( Str, tstStr, remainder )
```

This call to the function also returns true if *Str* begins with the characters found in the *tstStr* string. If this function returns true, then this function also copies the remaining charactes (i.e., those characters in *Str* that following the *tstStr* characters at the beginning of *Str*) to the *remainder* VAL object. If @matchStr returns false, the value of *remainder* is undefined (with the single exception noted below).

It is perfectly legal to specify the same string as the *Str* and *remainder* parameters. For example, consider the following invocation:

```
@matchStr( str, "Hello ", str )
```

If *str* begins with the string "Hello " then this function will return true and replace *str's* value with the string containing all characters beyond the sixth character of the string. If *str* does not begin with "Hello " then this function does not modify *str's* value.

Most of the HLA compile-time pattern matching functions also allow a second optional parameter. Consider the following invocation of the @oneOrMoreCset function:

```
@oneOrMoreCset( str, {’0’..’9’}, remainder, matched )
```

If this function returns true it will copied the sequence of characters at the beginning of *str* that are members of the specified character set (i.e., digits) to the *matched* VAL object. This function returns all characters beyond the digits in the *remainder* VAL object. If this function returns false, then *remainder* and *matched* will contain undefined values and this function will not affect *str*.

H.6.1 String/Cset Pattern Matching Functions

Among the most useful of the pattern matching functions are those that checking the leading characters of a string to see if they are members of a particular character set. The following rich set of functions provides this capability in the compile-time language.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@peekCset	@peekCset(*str, cset*, *rem*, *matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is a member of *cset*; it returns false otherwise. If *rem* is present, this function copies *str* to *rem* upon return. If *matched* is present and the function returns true, this function stores a copy of the first character into the *matched* VAL object. The value of *matched* is undefined if this function returns false.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneCset	@oneCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is a member of *cset*; it returns false otherwise. If *rem* is present, this function copies all characters of *str* beyond the first character to *rem* upon return. If *matched* is present and the function returns true, this function stores a copy of the first character into the *matched* VAL object. The value of *matched* is undefined if this function returns false.
@uptoCset	@uptoCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function copies all characters up to, but not including, a single character from the *cset* parameter. If the *str* parameter does not contain a character in the cset set, this function returns false. If it succeeds, and the matched parameter is present, it copies all characters it matches to the *matched* parameter and it copies all remaining characters to the *rem* parameter (if present).
@zeroOrOneCset	@zeroOrOneCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. This function copies the single character it matches (if any) to the *matched* string and any remaining characters in the string to the *rem* object (assuming *rem* and *matched* appear in the parameter list).

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@exactlyNCset	@exactlyNCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* are members of *cset*. The character at position *n+1* must not be a member of *cset*. If this function returns true, it copies the first *n* characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).
@firstNCset	@firstNCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* are members of *cset*. The character at position *n+1* may or may not be a member of *cset*. If this function returns true, it copies the first *n* characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nOrLessCset	@nOrLessCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. This function matches up to the first *n* characters of *str* are members of *cset*. The character at position *n+1* may or may not be a member of *cset*. If this function returns true, it copies the matching (up to *n*) characters of *str* to *matched* and copies any remaining characters to *rem* (assuming *rem* and *matched* are present).
@nOrMoreCset	@nOrMoreCset(*str, cset, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if it matches at least *n* characters from *str* in cset. If *rem* and *matched* appear in the parameter list, this function will copy all characters it matches to *matched* and copy any remaining characters into *rem*.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@nToMCset	@nToMCset(*str, cset, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if there are at least *n* characters in *cset* at the beginning of *str*. If *rem* and *matched* are present, this function will copy all the characters it matches (up to the mth position) into the *matched* string and copy any remaining characters into the *rem* string. The character at position *m+1* may or may not be a member of *cset*.
@exactlyNToMC-set	@ExactlyNToMCset(*str, cset, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if there are at least *n* characters in *cset* at the beginning of *str*. If *rem* and *matched* are present, this function will copy all the characters it matches (up to the mth position) into the *matched* string and copy any remaining characters into the *rem* string. If this function matches *m* characters, the character at position *m+1* must not be a member of *cset* or else this function will return false.
@zeroOrMoreCset	@zeroOrMoreCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If *rem* and *matched* are present, this function will copy all characters from the beginning of *str* that are members of *cset* to the *matched* string. It will copy all remaining characters to the *rem* string.

Table 6: HLA Compile-time Cset Pattern Matching Functions

Function	Parameters	Description
@oneOrMoreCset	@OneOrMoreCset(*str, cset, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *set* is a member of *cset*. If *rem* and *matched* are present, this function will copy all characters from the beginning of *str* that are members of *cset* to the *matched* string. It will copy all remaining characters to the *rem* string.

H.6.2 String/Character Pattern Matching Functions

Though not always as useful as the character set pattern matching functions, the HLA compile-time character matching functions are more efficent than the character set routines when matching single characters.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@peekChar	@peekChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is equal to *char*. If *rem* and *matched* are present and this function returns true, it also returns *str* in *rem* and *char* in *matched*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@oneChar	@oneChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character of *str* is equal to *char.* If *rem* and *matched* are present and this function returns true, it also returns all the characters *str* beyond the first character in *rem* and *char* in *matched*.
@uptoChar	@uptoChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the *char* exists some-where in *str.* If *rem* and *matched* are present and this function returns true, it also returns, in *rem*, all the characters in *str* start-ing with the first instance of *char.* It also returns all the characters up to (but not including) the first instance of *char* in *matched*.
@zeroOrOneChar	@zeroOrOneChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If *rem* and *matched* are present, it sets *matched* to the matched string (i.e., an empty string or the single character *char*) and it sets *rem* to the characters after the matched character value (the whole string if the first character of *str* is not equal to *char*).

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@zeroOrMore-Char	@zeroOrMoreChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function always returns true. If the first character of *str* does not match *char*, then this function returns the empty string in *matched* and it returns *str* in *rem*. If *str* begins with a sequence of characters all equal to *char*, then this function returns that sequence in *matched* and it returns the remaining characters in *rem*.
@oneOrMoreChar	@oneOrMoreChar(*str, char, rem, matched*) *str* must be a string expression. *char* must be a character expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first character in *str* is equal to *char*; other-wise it returns false. If this function returns true, it copies all leading occurrences of *char* into matched and any remain-ing characters from *str* into *rem*.
@exactlyNChar	@exactlyNChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* all match *char*. The $n+1^{th}$ character must not be equal to char. If this function returns true, it returns a string of *n* copies of *char* in *matched* and all remaining charac-ters (position *n* and beyond) in *rem*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@firstNChar	@firstNChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first *n* characters of *str* all match *char.* The *n+1*th character may or may not be equal to char. If this function returns true, it returns a string of *n* copies of *char* in *matched* and all remaining characters (position *n* and beyond) in *rem*.
@nOrLessChar	@nOrLessChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true as long as *str* begins with *n* or fewer (including zero) copies of *char.* It fails if *str* begins with more than *n* copies of *char.* If this function returns true, it returns a string, in *matched,* containing all copies of *char* that appear at the beginning of *str.* It returns all remaining characters in *rem*.
@nOrMoreChar	@nOrMoreChar(*str, char, n, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* must be a non-negative integer value. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if *str* begins with at least *n* copies of *char.* It returns false otherwise. If this function returns true, then it returns the leading characters that match *char* in *matched* and all following characters in *rem*.

Table 7: HLA Compile-time Character Matching Functions

Function	Parameters	Description
@nToMChar	@nToMChar(*str, char, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function matches any string that has at least *n* copies of *char* at the beginning of str. It will match up to *m* copies of *char*. The character at position *m+1* may be equal to *char*, but this function will not match that character. If this function returns true, it returns the string of matched characters in *matched* and any remaining characters in *rem*.
@exactlyNToM-Char	@exactlyNToMChar(*str, char, n, m, rem, matched*) *str* must be a string expression. *cset* must be a character set expression. *n* and *m* must be non-negative integer values. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function matches any string that has at least *n* copies of *char* at the beginning of str. It will match up to *m* copies of *char*. The character at position *m+1* must not be equal to *char*. If this function returns true, it returns the string of matched characters in *matched* and any remaining characters in *rem*.

H.6.3 String/Case Insenstive Character Pattern Matching Functions

HLA provides two sets of character matching routines: the previous section described the standard character matching routines, this section presents the case insensitive versions of those same routines.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@peekiChar	@peekiChar(*str, char, rem, matched*) See @peekChar for details.	Case insensitive version of @peekChar. See @peekChar for details.
@oneiChar	@oneiChar(*str, char, rem, matched*) See @oneChar for details.	Case insensitive version of @oneChar. See @oneChar for details.
@uptoiChar	@uptoiChar(*str, char, rem, matched*) See @uptoChar for details.	Case insensitive version of @uptoChar. See @uptoChar for details.
@zeroOrOneiChar	@zeroOrOneiChar(*str, char, rem, matched*) See @zeroOrOneChar for details.	Case insensitive version of @zeroOrOneChar. See @zeroOrOneChar for details.
@zeroOrMorei-Char	@zeroOrMoreiChar(*str, char, rem, matched*) See @zeroOrMoreChar for details.	Case insensitive version of @zeroOrMoreChar. See @zeroOrMoreChar for details.
@oneOrMoreiChar	@OneOrMoreiChar(*str, char, rem, matched*) See @OneOrMoreChar for details.	Case insensitive version of @OneOrMoreChar. See @OneOrMoreChar for details.
@exactlyNiChar	@exactlyNiChar(*str, char, n, rem, matched*) See @exactlyNChar for details.	Case insensitive version of @exactlyNChar. See @exactlyNChar for details.
@firstNiChar	@firstNiChar(*str, char, n, rem, matched*) See @firstNChar for details.	Case insensitive version of @firstNChar. See @firstNChar for details.
@nOrLessiChar	@nOrLessiChar(*str, char, n, rem, matched*) See @nOrLessChar for details.	Case insensitive version of @nOrLessChar. See @nOrLessChar for details.
@nOrMoreiChar	@nOrMoreiChar(*str, char, n, rem, matched*) See @nOrMoreChar for details.	Case insensitive version of @nOrMoreChar. See @nOrMoreChar for details.

Table 8: HLA Compile-time Case Insensitive Character Matching Functions

Function	Parameters	Description
@nToMiChar	@nToMiChar(*str, char, n, m, rem, matched*) See @nToMChar for details.	Case insensitive version of @nToMChar. See @nToMChar for details.
@exactlyNToMi-Char	@exactlyNToMiChar(*str, char, n, m, rem, matched*) See @exactlyNToMChar for details.	Case insensitive version of @exactlyNToM-Char. See @exactlyN-ToMChar for details.

H.6.4 String/String Pattern Matching Functions

Another set of popular pattern matching routines in the compile-time function set are the string matching routines. These routines check to see if a string begins with some specified sequence of characters. Next to the character set matching functions, these are probably the most commonly used pattern matching functions.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@matchStr	@matchStr(*str, tstStr, rem, matched*) *str* must be a string expression. *tstStr* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if *str* begins with the characters found in *tstStr.* If this function returns true, it also returns *tstStr* in *matched* and all characters in str following the *tstStr* characters in *rem*.
@matchiStr	@matchiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive version of @matchStr.

Table 9: Compile-Time String Matching Functions

Function	Parameters	Description
@uptoStr	@uptoStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This function returns true if *tst-Str* appears somewhere within *str*, it returns false otherwise. If this function returns true, then it returns all characters up to, but not including, the char-acters from *tstStr* in *matched*. It returns all following charac-ters (including the *tstStr* sub-string) in *rem*.
@uptoiStr	@uptoiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive ver-sion of @uptoStr.
@matchToStr	@matchToStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This function is similar to @uptoStr except if it returns true it will copy all characters up to and including *tstStr* to *matched* and all following characters to *rem*.
@matchToiStr	@matchToiStr(*str, tstStr, rem, matched*) *Same parameters as matchStr, see that function for details.*	This is a case insensitive ver-sion of @matchToStr.

H.6.5 String/Misc Pattern Matching Functions

This last group of compile-time pattern matching functions check for certain special types of strings, such as HLA identifiers, numeric constants, whitespace, and the end of the string.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchID	@matchID(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The types of *rem* and *matched* are irrelevant but they must be VAL objects. This function stores a result into these two objects.	This function returns true if the first sequence of characters in *str* match the definition of an HLA identifier. An HLA identifier is any sequence of characters that begins with an underscore or an alphabetic character that is followed by zero or more underscore or alphanumeric characters. If this function returns true, it copies the identifier to *matched* and all following characters to *rem*.
@matchIntConst	@matchIntConst(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The type of *rem* is irrelevant but it must be a VAL object. If *matched* is present, it must be an *int32* or *uns32* object. **Note: unlike most pattern matching functions, *matched* is not a string object.**	This function returns true if the leading characters of *str* are decimal digits (as per HLA, underscores are legal in the interior of the integer string). If this function returns true, it converts the matched integer string to integer form and stores this value in *matched*. This function returns the remaining characters after the numeric digits in *rem*.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@matchRealConst	@matchRealConst(str, rem, matched) *str* must be a string expression. *rem* and *matched* are optional arguments (both are optional, but if *matched* is present, *rem* must also be present). The type of *rem* is irrelevant but it must be a VAL object. If *matched* is present, it must be an *real80* object. **Note: unlike most pattern matching functions, *matched* is not a string object.**	This function returns true if the leading characters of *str* correspond to the HLA definition of a real literal constant. If this function returns true, it also returns the remaining characters in *rem* and it converts the real string to *real80* format and stores this value in *matched*.
@matchNumericCo-nst	@matchNumericConst(str, rem, matched) Same parameters as @matchIntConst or @matchRealConst; see those functions for details.	This is a combination of @matchIntConst and @matchRealConst. If this function matches a string at the beginning of *str* that is a legal numeric con-stant, it will convert that value to numeric form (*int32* or *real80*) and store the value into *matched*. This function returns any following characters in *rem*.
@matchStrConst	@matchStrConst(str, rem, matched) Same parameters as @matchID; see @matchID for details.	This function returns true if *str* begins with an HLA compatible literal string constant. If this function matches such a constant, it will store the matched string, minus the delimit-ing quotes, into the *matched* variable. It will also store any following charcters into *rem*.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@zeroOrMoreWS	@zeroOrMoreWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object.	This function always returns true. It matches zero or more "whitespace" characters at the beginning of str. Whitespace includes spaces, newlines, tabs, and certain other special characters. Note that this function does not return the matched string. To return matched whitespace characters, use @zeroOrMoreCset.
@oneOrMoreWS	@oneOrMoreWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function returns true if it matches at least one whitespace character. If it returns true, it also returns any characters following the leading whitespace characters in the *rem* variable.
@wsOrEOS	@wsOrEOS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function succeeds if there is whitespace at the beginning of *str* or if *str* is the empty string. If it succeeds and there is leading whitespace, this function returns the remaining characters in *rem*. If this function succeeds and the string was empty, this function returns the empty string in *rem*. This function fails if the string is not empty and it begins with non-whitespace characters.

Table 10: Miscellaneous Compile-time Pattern Matching Functions

Function	Parameters	Description
@wsThenEOS	@wsThenEOS(str) *str* must be a string expression.	This function returns true if *str* contains zero or more whitespace characters followed by the end of the string. If fails if there are any other characters in the string.
@peekWS	@peekWS(str, rem) *str* must be a string expression. *rem* is an optional argument. The type of *rem* is irrelevant but it must be a VAL object. This function stores a result into this object.	This function returns true if the next character in *str* is a whitespace character. This function returns a copy of *str* in *rem* if it is successful.
@eos	@eos(str) *str* must be a string expression.	This function returns true if and only if *str* is the empty string.

H.7 HLA Information and Symbol Table Functions

The symbol table functions provide access to information in HLA's internal symbol table database. These functions are particularly useful within macros to determine how to generate code for a particular macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@name	@name(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a string specifying the name of the specified identifier. The name this function returns is computed after macro and text constant expansion. This function is useful mainly in macros to determine the name of a macro parameter.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@type	@type(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a *dword* constant that should be unique for any given type in the program. You can use this to compare the types of two different objects. For guaranteed uniqueness, see @typeName.
@typeName	@typeName(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a string that specifies the type of of the parameter.
@pType	@pType(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that specifies the primitive type of the object. See the ptXXXXX constants in the HLA.HHF header file for the actual values this function returns.
@class	@class(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns a small numeric constant that classifies the identifier or expression as to whether it is a constant, VAL, variable, parameter, static object, procedure, etc. See the cXXXX constants in the HLA.HHF header file for a list of the possible return values.
@size	@size(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns the size, in bytes of the specified object.
@offset	@offset(identifier) *identifier* must be a defined VAR or parameter symbol in the HLA program.	This function returns a numeric constant providing the offset into a procedure's activation record for a VAR or parameter object.
@staticName	@staticName(identifier) *identifier* must be a defined static, procedure, method, iterator, or external symbol in the HLA program.	This function returns a string that specifies the internal name that HLA uses for the object.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@lex	@lex(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns a small integer constant the specifies the static nesting level for the specified symbol. All symbols appearing in the main program have a lex level of zero. Symbols you define in procedures within the main program have a lex level of one. Higher lex level values are possible if you define procedures inside procedures.
@isExternal	@isExternal(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns true if the specified identifier is an external symbol. Note that this function will return true if the symbol is defined external and a declaration for the symbol appears later in the code.
@arity	@arity(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns zero if the specified symbol is not an array. It returns a small integer constant denoting the number of dimensions if the identifier is an array object.
@dim	@dim(identifier) *identifier* must be a defined symbol in the HLA program.	If the specified identifier is an array object, this function returns an array constant with one element for each dimension in the array. Each element of this array constant specifies the number of elements for each dimension of the array. If the identifier is not an array object, this function returns an array with a single element and that element's value will be zero.
@elements	@elements(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns the number of elements in an array object. If the specified identifier is not an array object, this function returns zero. For multi-dimensional arrays, this function returns the product of each of the dimensions.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@elementSize	@elementSize(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns the size, in bytes, of an element of the specified array.
@defined	@defined(identifier) *identifier* must be a defined symbol in the HLA program.	This function returns true if the specified symbol is defined prior to that point in the program.
@pClass	@pClass(identifier) *identifier* must be a parameter in the current procedure of an HLA program.	This function returns a small integer constant specifying the parameter passing mechanism for the specified parameter. The HLA.HHF header file defines the return values for this function (see the XXXX_pc constant declarations).
@isConst	@isConst(identifierOrExpression) *identifierOrExpression* may be a defined symbol in the HLA program or a constant expression.	This function returns true if the expression is a constant expression that HLA can evaluate at that point in the program.
@isReg	@isReg(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an 80x86 general purpose register.
@isReg8	@isReg8(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an eight-bit 80x86 general purpose register.
@isReg16	@isReg16(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 16-bit 80x86 general purpose register.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@isReg32	@isReg32(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to a 32-bit 80x86 general purpose register.
@isFReg	@isFReg(Expression) *Expression* may be arbitrary text, but it is typically a register object.	This function returns true if the operand corresponds to an FPU register.
@isMem	@isMem(Expression) *Expression* may be arbitrary text, but it is typically a memory object (including various addressing modes).	This function returns true if the specified parameter corresponds to a legal 80x86 memory address.
@isClass	@isClass(Text) *Text* may be arbitrary text, but it is typically a class object.	This function returns true if the text parameter is a class name or a class object.
@isType	@isClass(Text) *Text* may be arbitrary text, but it is typically a type name.	This function returns true if the specified text is a type identifier.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@section	@section	This function returns a 32-bit value that specifies which portion of the program HLA is currently processing. This information is mainly useful in macros to determine where a macro expansion is taking place. This function returns the following bit values: Bit 0: Currently in the CONST section. Bit 1: Currently in the VAL section. Bit 2: Currently in the TYPE section. bit 3: Currently in the VAR section. bit 4: Currently in the STATIC section. bit 5: In the READONLY section. Bit 6: In the STORAGE section. Bit 7: Currently in the DATA section. Bits 8-11: reserved. Bit 12: Processing statements in main. Bit 13: Statements in a procedure. Bit 14: Statements in a method. Bit 15: Statements in an iterator. Bit 16: Statements in a macro. Bit 17: Statements in a keyword macro. Bit 18: In a Terminator macro. Bit 19: Statements in a Thunk. Bits 20-22: reserved. Bit 23: Processing statements in a Unit. Bit 24: Statements in a Program. Bit 25: Processing a Record declaration. Bit 26: Processing a Union declaration. Bit 27: Processing a Class declaration. Bit 28: Processing a Namespace declaration.

Table 11: HLA Information and Compile-time Symbol Table Information Functions

Function	Parameters	Description
@curLex	@curLex	Returns the current lex level of this statement within the program.
@curOffset	@curOffset	Returns the current offset into the activation record. This is the offset of the last VAR object declared in the current program/procedure.
@curDir	@curDir	Returns +1 if processing parameters, -1 otherwise. This corresponds to whether offsets are increasing or decreasing in an activation record during compilation. This function also returns +1 when processing fields in a record or class; it returns zero when processing fields of a union.
@addofs1st	@addofs1st	This function returns true when processing local variables, it returns false when processing parameters and record/class/union declarations.
@lastObject	@lastObject	This function returns a string containing the name of the last macro object processed.
@lineNumber	@lineNumber	This function returns an *uns32* value specifying the current line number in the file.

H.8 Compile-Time Variables

The HLA compile-time variables are special symbols that not only return values, but allow you to modify their internal values as well. You may use these symbols exactly like any VAL object in your program with the exception that you cannot change the type (generally *int32* or *boolean*) of these objects. By changing the values of these *pseudo-variables*, you can affect the way HLA generates code in your programs.

Table 12: HLA Compile-time Variables

Pseudo-Variable	Description
@parmOffset	This variable specifies the starting offset for parameters in a function. This should be eight for most procedures. If you change this value, HLA's automatic code generation for procedure calls may fail.
@localOffset	This variable specifies the starting offset for local variables in a procedure. This is typically zero. If you change this value, it will affect the offsets of all local symbols in the activation record, hence you should modify this value only if you really know what you're doing (and have a good reason for doing it).
@enumSize	This variable specifies the size (in bytes) of enum objects in your program. By default, this value is one. If you want word or dword sized enum objects you can change this variable to two or four. Other values may create problems for the compiler.
@minParmSize	This variable specifies the minimum number of bytes for each parameter. Under Windows, this should always be four.
@bound	This boolean variable, whose default value is true, controls the compilation of the BOUND instruction. If this variable contains false, HLA does not compile BOUND instructions. You can set this value to true or false throughout your program to control the emission of bound instructions.
@into	This boolean variable, whose default value is false, controls the compilation of the INTO instruction. If this variable contains false, HLA does not compile INTO instructions. You can set this value to true or false throughout your program to control the emission of INTO instructions.
@trace	Enables HLA's statement tracing facilities. See the separate appendix for details.
@exceptions	If true (the default) then HLA uses the full exception handling package in the HLA Standard Library. If false, HLA uses a truncated version. This allows programmers to write their own exception handling code.

H.9 Miscellaneous Compile-Time Functions

This last category contains various functions and objects that are quite useful in compile-time programs.

Table 13: Miscellaneous HLA Compile-time Functions and Objects

Function	Parameters	Description
@text	@text(str) *str* must be a string constant expression.	This function expands the string constant in-line as text, replacing this function invocation with the specified text.
@eval	@eval(text) *text* is an arbitrary sequence of textual characters.	This function is useable only within macro invocation parameter lists. It immediately evaluates the text object and passes the resulting text as the parameter to the macro. This provides eager evaluation capabilities for macro parameters.
@string:*identifier*	@string:*identifier* *identifier* must be a *text* constant.	This function returns a string constant corresponding to the string data in the specified identifer. It does not otherwise affect the value or type of *identifier.*
@toString:*identifier*	@string:*identifier* *identifier* must be a *text* constant.	This function converts the type of *identifier* from *text* to *string* and then returns the value of that string object.
@global:*identifier*	@string:*identifier* *identifier* must be an HLA identifier declared outside the current namespace.	@global is legal only within a namespace declaration. It provides access to identifiers outside the namespace.

Installing HLA on Your System Appendix I

This information has been moved to Volume One, Chapter Two. Please see the HLA on-line documentation at http://webster.cs.ucr.edu if you require additional installation assistance.

Debugging HLA Programs Appendix J

J.1 The @TRACE Pseudo-Variable

HLA v1.x has a few serious defects in its design. One major issue is debugging support. HLA v1.x emits MASM code that it runs through MASM in order to produce executable object files. Unfortunately, while this scheme makes the development, testing, and debugging of HLA easier, it effectively eliminates the possibility of using existing source level debugging tools to locate defects in an HLA program[1]. Starting with v1.24, HLA began supporting a new feature to help you debug your programs: the “@trace” pseudo-variable. This appendix will explain the purpose of this compile-time variable and how you can use it to locate defects in your programs.

By default, the compile-time variable *@trace* contains false. You can change its value with any variation of the following statement:

```
?@trace := <<boolean constant expression>>;
```

Generally, “<<boolean constant expression>>” is either *true* or *false*, but you could use any compile-time constant expression to set *@trace’s* value.

Once you set *@trace* to true, HLA begins generating extra code in your program In fact, before almost every executable statement HLA injects the following code:

```
    _traceLine_( filename, linenumber );
```

The *filename* parameter is a string specifying the name of the current source file, the *linenumber* parameter is an uns32 value that is the current line number of the file. The _traceLine_ procedure uses this information to display an appropriate trace value while the program is running.

HLA will automatically emit the above procedure call in between almost all instructions appearing in your program[2]. Assuming that the *_traceLine_* procedure simply prints the filename and line number, when you run your application it will create a log of each statement it executes.

You can control the emission of the *_traceLine_* procedure calls in your program by alternately setting *@trace* to *true* or *false* throughout your code. This lets you selectively choose which portions of your code will provide trace information during program execution. This feature is very important because if you’re displaying the trace information to the console, your program runs thousands, if not millions, of times slower during the trace operation. It wouldn’t do to have to trace through a really long loop in order to trace through following code that you’re concerned about. By setting *@trace* to *false* prior to the long loop and setting it to true immedately after the loop, you can execute the loop at full speed and then begin tracing the code you want to check once the loop completes execution.

HLA does not supply the *_traceLine_* procedure; it is your responsibility to write the code for this procedure. The following is a typical implementation:

```
procedure trace( filename:string; linenum:uns32 ); external( “_traceLine_” );
procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    pushfd();
    stdout.put( filename, “: #”, linenum, nl );
    popfd();

end trace;
```

1. Of course, you can also debug the MASM output using a source level debugger, but this is not a very pleasant experience.
2. HLA does not emit this procedure call between statements that are composed and a few other miscellaneous statements. However, your programs probably get better than 95% coverage so this will be sufficient for most purposes.

This particular procedure simply prints the filename and line number each time HLA calls it. Therefore, you'll get a trace sent to the standard output device that looks something like the following:

```
t.hla: #19
t.hla: #20
t.hla: #21
t.hla: #22
etc.
```

A very important thing to note about this sample implementation of *_traceLine_* is that it preserves the FLAGs register. The *_traceLine_* procedure must preserve all registers including the flags. The example code above only preserves the FLAGS because the *stdout.put* macro always preserves all the registers. However, were this code to modify other registers or call some procedure that modifies some registers, you would want to preserve those register values as well. Remember, HLA calls this procedure between most instructions, if you do not preserve the registers and flags within this procedure, it will adversely affect the running of your program.

This is very important: the *@trace* variable must contain false while HLA is compiling your *_traceLine_* procedure. If HLA emits trace code inside the trace procedure, this will create an infinite loop via infinite recursion which will crash your program. Always make sure *@trace* is false across the compilation of your *_traceLine_* procedure.

Here's a sample program using the *@trace* variable and sample output for the program:

```
program t;
#includeOnce( "stdlib.hhf" )


procedure trace( filename:string; linenum:uns32 ); external( "_traceLine_" );

?@trace := false; // Must be false for TRACE routine.

procedure trace( filename:string; linenum:uns32 ); nodisplay;
begin trace;

    stdout.put( filename, ": #", linenum, nl );

end trace;



?@trace := true;

begin t;


    mov( 0, ecx );              // Line 22
    if( ecx == 0 ) then         // Line 23

        mov( 10, ecx );         // Line 25

    else

        mov( 5, ecx );

    endif;
    while( ecx > 0 ) do         // Line 32

        dec( ecx );             // Line 34

    endwhile;
    for( mov( 0, ecx ); ecx < 5; inc( ecx )) do   // Line 37
```

```
        add( 1, eax );                          // Line 39

    endfor;


end t;

Sample Output:

t.hla: #22
t.hla: #23
t.hla: #25
t.hla: #32
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #34
t.hla: #37
t.hla: #39
t.hla: #39
t.hla: #39
t.hla: #39
t.hla: #39
```

Although the *_traceLine_* procedure in this example is very simple, you can write as sophisticated a procedure as you like. However, do keep in mind that this code executes between almost every pair of statements your program has, so you should attempt to keep this procedure as short and as fast as possible. However, if desired you can stop the program, request user input, and let the user specify how program control is to proceed.

One very useful purpose for HLA's trace capabilities is tracking down bad pointer references in your program. If you program aborts with an "Illegal memory access" exception, you can pinpoint the offending instruction by turning on the trace option and letting the program run until the exception occurs. The last line number in the trace will be (approximately, within one or two statements) the line number of the offending instruction.

You can also display (global) values within the *_traceLine_* procedure. Printing local values, unfortunately, is problematic since external procedures must appear at lex level one in your program (i.e., you cannot nest the function within some other procedure). It is possible to set up a pointer to a local procedure and call that procedure indirectly from within *_traceLine_*, but this effort is worth it only on rare occasions. Usually it's just easier to stick a "stdout.put" statement directly in the code where you wish to view the output. However, if some variable is being overwritten by some unknown statement in your program, and you don't know where the program is modifying the variable, printing the variables value from the *_traceLine_* procedure can help you pinpoint the statement that overwrites the variable.

Of course, another way to debug HLA programs is to stick a whole bunch of "print" statements in your code. The problem with such statements, however, is that you've got to remember to remove them before you ship your final version of the program. There are few things more embarrasing than having your customer ask you why your program is printing debug messages in the middle of the reports your program is producing for them. You can use conditional compilation (#IF statements) to control the compilation of such statements, but conditional compilation statements tend to clutter up your source code (even more so than the debugging print statements). One solution to this problem is to create a macro, let's call it *myDebugMsg*, that hides all the nasty details. Consider the following code:

```
#macro myDebugMsg( runTimeFlag, msg );
```

```
    #if( @defined( DEBUG ))   // If DEBUG is not defined, then don't emit code.

        #if( DEBUG )           // DEBUG has to be a boolean const equal to true.

            #if( @isConst( runTimeFlag ))

                #if( runTimeFlag )

                    stdout.put( msg, nl );

                #endif

            #else  // runTimeFlag is a run-time variable

                if( runTimeFlag ) then

                    stdout.put( msg, nl );

                endif;

            #endif  // runTimeFlag is/is not Constant

        #endif // DEBUG

    #endif // DEBUG is defined

#endmacro
```

With this macro defined, you can write a statement like the following to print/not print a message at run-time:

```
    myDebugMsg( BooleanValue, "This is a debug message" );
```

The *BooleanValue* expression is either a boolean constant expression, a boolean variable, or a register. If this value is true, then the program will print the message; if this value is false, the program does not print the message. Since this is a variable, you can control debugging output at run-time by changing the value of the *BooleanValue* parameter.

As *myDebugMsg* is written, you must define the boolean constant *DEBUG* and set it to true or the program will not compile the debugging statements. If DEBUG is a VAL object, you can actually

You can certainly expand upon this macro by providing support for a variable number of parameters. This would allow you to specify an list of values to display in the *stdout.put* macro invocation. See the chapter on Macros for more information about variable parameter lists in macros if you want to do this.

J.2 The Assert Macro

Another tool you may use to help locate defects in your programs is the assert macro. This macro is available in the "excepts.hhf" header file (included by "stdlib.hhf"). An invocation of this macro takes the following form:

```
        assert( boolean_expression );
```

Note that you do not preface the macro with "except." since the macro declaration appears outside the "except" namespace in the excepts.hhf header file. The boolean_expression component is any expression that is legal within an HLA HLL control statement like IF, WHILE, or REPEAT..UNTIL.

The assert macro evaluates the expression and simply returns if the expression is true. If the expression evaluates false, then this macro invocation will raise an exception (ex.AssertionFailed). By liberally sprinkling assert invocations through your code, you can test the behavior of your program and stop execution if an assertion fails (thus helping you to pinpoint problems in your code).

One problem with placing a lot of asserts throughout your code is that each assert takes up a small amount of space and execution time. Fortunately, the HLA Standard Library provides a mechanism by which you may control code generation of the assert macros. The excepts.hhf header files defines a VAL object, ex.NDEBUG, that is initialized with the value false. When this compile-time variable is false, HLA will emit the code for the assert macro; however, if you set this constant to true, then HLA will not emit any code for the assert macro. Therefore, you may liberally place assert macro invocations throughout your code and not worry about their effect on the final version of your program you ship; you can easily remove the impact of all assert macros in your program by sticking in a statement of the form “?ex.NDEBUG:=true;” in your source file.

Note that you may selectively turn asserts on or off by alternately placing “ex.NDEBUG:=false;” and “?ex.NDEBUG:=true;” throughout your code. This allows you to leave some important assertions active in your code, even when you ship the final version.

Since assert raises an exception, you may use the HLA try..exception..endtry statement to catch any exceptions that fail. If you do not handle a specific assertion failure, HLA will abort the program with an appropriate message that tells you the (source) file and line number where the assertion failed.

(More to come someday...)

Comparing HLA and MASM Appendix K

To be written... Until then, check the output HLA proceduces to see the code that HLA emits for various statements.

HLA Code Generation for HLL Statements Appendix L

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 chosing 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 appendix 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.

L.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.

L.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 a (MASM) file similar to the following[1]:

```
            if      @Version lt 612
            .586
            else
            .686
            .mmx
            .xmm
```

1. Because the code generator in HLA is changing all the time, this file may not reflect an accurate compilation of the above HLA code. However, the concepts will be the same.

```
                endif
                .model  flat, syscall
offset32        equu     <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        endp
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.

L.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.

L.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)
<
<=
>
>=
```

In the expressions above operand$_1$ and operand$_2$ 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 that 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
                jb      ?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:

reg$_8$ in *CSet_Constant*
reg$_8$ in *CSet_Variable*

reg$_8$ not in *CSet_Constant*
reg$_8$ 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 cannot 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.

L.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 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:
       cmp eax, 10
       ja ?4_label

?4_label:

       cmp ebx, 10
       jne ?2_lbl2
```

L.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 and 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;
    }#) then

        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.

L.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.

L.8 repeat..until

L.9 for..endfor

L.10 forever..endfor

L.11 break, breakif

L.12 continue, continueif

L.13 begin..end, exit, exitif

L.14 foreach..endfor

L.15 try..unprotect..exception..anyexception..endtry, raise

Editorial Note: This document is a work in progress. At some future date I will finish the sections above. Until then, use the HLA “-s” compiler option to emit MASM code and study the MASM output as described in this appendix.