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 make-files 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 make-files.

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 pro-

gram 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 abbre-

viations 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. "InputLine" 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.

