Volume One: Data Representation

Chapter One: Foreword

      An introduction to this text and the purpose behind this text.

Chapter Two:Hello, World of Assembly Language

      A brief introduction to assembly language programming using the HLA language.

Chapter Three:Data Representation

      A discussion of numeric representation on the computer.

Chapter Four:More Data Representation

      Advanced numeric and non-numeric computer data representation.

Chapter Five: Questions, Projects, and Laboratory Exercises

      Test what you've learned in the previous chapters!

    These five chapters are appropriate for all courses teaching machine organization and assembly language programming.

Volume One:Data Representation

Beta Draft - Do not distribute
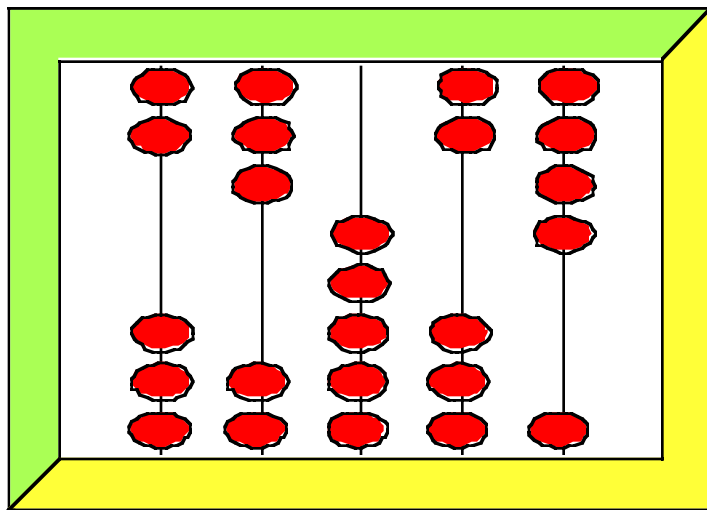
Nearly every text has a throw-away chapter as Chapter One. Here's my version. Seriously, though, some important copyright, instructional, and support information appears in this chapter. So you'll probably want to read this stuff. Instructors will definitely want to review this material.

1.1   Foreword to the HLA Version of "The Art of Assembly..."

In 1987 I began work on a text I entitled "How to Program the IBM PC, Using 8088 Assembly Language." First, the 8088 faded into history, shortly thereafter the phrase "IBM PC" and even "IBM PC Compatible" became far less dominant in the industry, so I retitled the text "The Art of Assembly Language Programming." I used this text in my courses at Cal Poly Pomona and UC Riverside for many years, getting good reviews on the text (not to mention lots of suggestions and corrections). Sometime around 1994-1995, I converted the text to HTML and posted an electronic version on the Internet. The rest, as they say is history. A week doesn't go by that I don't get several emails praising me for releasing such a fine text on the Internet. Indeed, I only hear three really big complaints about the text: (1) It's a University textbook and some people don't like to read textbooks, (2) It's 16-bit DOS-based, and (3) there isn't a print version of the text. Well, I make no apologies for complaint #1. The whole reason I wrote the text was to support my courses at Cal Poly and UC Riverside. Complaint #2 is quite valid, that's why I wrote this version of the text. As for complaint #3, it was really never cost effective to create a print version; publishers simply cannot justify printing a text 1,500 pages long with a limited market. Furthermore, having a print version would prevent me from updating the text at will for my courses.

The astute reader will note that I haven't updated the electronic version of "The Art of Assembly Language Programming" (or "AoA") since about 1996. If the whole reason for keeping the book in electronic form has been to make updating the text easy, why haven't there been any updates? Well, the story is very similar to Knuth's "The Art of Computer Programming" series: I was sidetracked by other projects[1].

The static nature of AoA over the past several years was never really intended. During the 1995-1996 time frame, I decided it was time to make a major revision to AoA. The first version of AoA was MS-DOS based and by 1995 it was clear that MS-DOS was finally becoming obsolete; almost everyone except a few die-hards had switched over to Windows. So I knew that AoA needed an update for Windows, if nothing else.

I also took some time to evaluate my curriculum to see if I couldn't improve the pedagogical (teaching) material to make it possible for my students to learn even more about 80x86 assembly language in a relatively short 10-week quarter.

One thing I've learned after teaching an assembly language course for over a decade is that support software makes all the difference in the world to students writing their first assembly language programs. When I first began teaching assembly language, my students had to write all their own I/O routines (including numeric to string conversions for numeric I/O). While one could argue that there is some value to having students write this code for themselves, I quickly discovered that they spent a large percentage of their project time over the quarter writing I/O routines. Each moment they spent writing these relatively low-level routines was one less moment available to them for learning more advanced assembly language programming techniques. While, I repeat, there is some value to learning how to write this type of code, it's not all that related to assembly language programming (after all, the same type of problem has to be solved for *any* language that allows numeric I/O). I wanted to free the students from this drudgery so they could learn more about assembly language programming. The result of this observation was "The UCR Standard Library for 80x86 Assembly Language Programmers." This is a library containing several hundred I/O and utility functions that students could use in their assembly language programs. More than nearly anything else, the UCR Standard Library improved the progress students made in my courses.

---

1. Actually, another problem is the effort needed to maintain the HTML version since it was a manual conversion from Adobe Framemaker. But that's another story...

It should come as no surprise, then, that one of my first projects when rewriting AoA was to create a new, more powerful, version of the UCR Standard Library. This effort (the UCR Stdlib v2.0) ultimately failed (although you can still download the code written for v2.0 from http://webster.cs.ucr.edu). The problem was that I was trying to get MASM to do a little bit more than it was capable of and so the project was ultimately doomed.

To condense a really long story, I decided that I needed a new assembler. One that was powerful enough to let me write the new Standard Library the way I felt it should be written. However, this new assembler should also make it much easier to learn assembly language; that is, it should relieve the students of some of the drudgery of assembly language programming just as the UCR Standard Library had. After three years of part-time effort, the end result was the "High Level Assembler," or HLA.

HLA is a radical step forward in teaching assembly language. It combines the syntax of a high level language with the low-level programming capabilities of assembly language. Together with the HLA Standard Library, it makes learning and programming assembly language almost as easy as learning and programming a High Level Language like Pascal or C++. Although HLA isn't the first attempt to create a hybrid high level/low level language, nor is it even the first attempt to create an assembly language with high level language syntax, it's certainly the first complete system (with library and operating system support) that is suitable for teaching assembly language programming. Recent experiences in my own assembly language courses show that HLA is a major improvement over MASM and other traditional assemblers when teaching machine organization and assembly language programming.

The introduction of HLA is bound to raise lots of questions about its suitability to the task of teaching assembly language programming (as well it should). Today, the primary purpose of teaching assembly language programming at the University level isn't to produce a legion of assembly language programmers; it's to teach machine organization and introduce students to machine architecture. Few instructors realistically expect more than about 5% of their students to wind up working in assembly language as their primary programming language[2]. Doesn't turning assembly language into a high level language defeat the whole purpose of the course? Well, if HLA lets you write C/C++ or Pascal programs and attempted to call these programs "assembly language" then the answer would be "Yes, this defeats the purpose of the course." However, despite the name and the high level (and very high level) features present in HLA, HLA is still assembly language. An HLA programmer still uses 80x86 machine instructions to accomplish most of the work. And those high level language statements that HLA provides are purely optional; the "purist" can use nothing but 80x86 assembly language, ignoring the high level statements that HLA provides. Those who argue that HLA is not *true* assembly language should note that Microsoft's MASM and Borland's TASM both provide many of the high level control structures found in HLA[3].

Perhaps the largest deviation from traditional assemblers that HLA makes is in the declaration of variables and data in a program. HLA uses a very Pascal-like syntax for variable, constant, type, and procedure declarations. However, this does not diminish the fact that HLA is an assembly language. After all, at the machine language (vs. assembly language) level, there is no such thing as a data declaration. Therefore, any syntax for data declaration is an abstraction of data representation in memory. I personally chose to use a syntax that would prove more familiar to my students than the traditional data declarations used by assemblers.

Indeed, perhaps the principle driving force in HLA's design has been to leverage the student's existing knowledge when teaching them assembly language. Keep in mind, when a student first learns assembly language programming, there is so much more for them to learn than a handful of 80x86 machine instructions and the machine language programming paradigm. They've got to learn assembler directives, how to declare variables, how to write and call procedures, how to comment their code, what constitutes good programming style in an assembly language program, etc. Unfortunately, with most assemblers, these concepts are completely different in assembly language than they are in a language like Pascal or C/C++. For example, the indentation techniques students master in order to write readable code in Pascal just don't apply to (traditional) assembly language programs. That's where HLA deviates from traditional assemblers. By using a

---

2. My experience suggests that only about 10-20% of my students will ever write *any* assembly language again once they graduate; less than 5% ever become regular assembly language users.

3. Indeed, in some respects the MASM and TASM HLL control structures are actually *higher* level than HLA's. I *specifically* restricted the statements in HLA because I did not want students writing "C/C++ programs with MOV instructions."

high level syntax, HLA lets students leverage their high level language knowledge to write good readable programs. HLA will not let them avoid learning machine instructions, but it doesn't force them to learn a whole new set of programming style guidelines, new ways to comment your code, new ways to create identifiers, etc. HLA lets them use the knowledge they already possess in those areas that really have little to do with assembly language programming so they can concentrate on learning the important issues in assembly language.

So let there be no question about it: HLA is an assembly language. It is not a high level language masquerading as an assembler[4]. However, it is a system that makes learning and using assembly language easier than ever before possible.

Some long-time assembly language programmers, and even many instructors, would argue that making a subject easier to learn diminishes the educational content. Students don't get as much out of a course if they don't have to work very hard at it. Certainly, students who don't apply themselves as well aren't going to learn as much from a course. I would certainly agree that if HLA's only purpose was to make it easier to learn a fixed amount of material in a course, then HLA would have the negative side-effect of reducing what the students learn in their course. However, the real purpose of HLA is to make the educational process more efficient; not so the students spend less time learning a fixed amount of material (although HLA could certainly achieve this), but to allow the students to learn the same amount of material in less time *so they can use the additional time available to them to advance their study of assembly language*. Remember what I said earlier about the UCR Standard Library- it's introduction into my course allowed me to teach even more advanced topics in my course. The same is true, even more so, for HLA. Keep in mind, I've got ten weeks in a quarter. If using HLA lets me teach the same material in seven weeks that took ten weeks with MASM, I'm not going to dismiss the course after seven weeks. Instead, I'll use this additional time to cover more advanced topics in assembly language programming. That's the real benefit to using pedagogical tools like HLA.

Of course, once I've addressed the concerns of assembly language instructors and long-time assembly language programmers, the need arises to address questions a student might have about HLA. Without question, the number one concern my students have had is "If I spend all this time learning HLA, will I be able to use this knowledge once I get out of school?" A more blunt way of putting this is "Am I wasting my time learning HLA?" Let me address these questions using three points.

First, as pointed out above, most people (instructors and experienced programmers) view learning assembly language as an educational process. Most students will probably never program full-time in assembly language, indeed, few programmers write more than a tiny fraction (less than 1%) of their code in assembly language. One of the main reasons most Universities require their students to take an assembly language course is so they will be familiar with the low-level operation of their machine and so they can appreciate what the compiler is doing for them (and help them to write better HLL code once they realize how the compiler processes HLL statements). HLA is an assembly language and learning HLA will certainly teach you the concepts of machine organization, the real purpose behind most assembly language courses.

The second point to ponder is that learning assembly language consists of two main activities; learning the assembler's syntax and learning the assembly language programming paradigm (that is, learning to *think* in assembly language). Of these two, the second activity is, by far, the more difficult. HLA, since it uses a high level language-like syntax, simplifies learning the assembly language syntax. HLA also simplifies the initial process of learning to program in assembly language by providing a crutch, the HLA high level statements, that allows students to use high level language semantics when writing their first programs. However, HLA does allow students to write "pure" assembly language programs, so a good instructor will ensure that they master the full assembly language programming paradigm before they complete the course. Once a student masters the semantics (i.e., the programming paradigm) of assembly language, learning a new syntax is relatively easy. Therefore, a typical student should be able to pick up MASM in about a week after mastering HLA[5].

As for the third and final point: to those that would argue that this is still extra effort that isn't worthwhile, I would simply point out that none of the existing assemblers have more than a cursory level of com-

---

4. The C-- language is a good example of a low-level non-assembly language, if you need a comparison.
5. This is very similar to mastering C after learning C++.

patibility. Yes, TASM can assemble most MASM programs, but the reverse is not true. And it's certainly not the case that NASM, A86, GAS, MASM, and TASM let you write interchangeable code. If you master the syntax of one of these assemblers and someone expects you to write code in a different assembler, you're still faced with the prospect of having to learn the syntax of the new assembler. And that's going to take you about a week (assuming the presence of well-written documentation). In this respect, HLA is no different than any of the other assemblers.

Having addressed these concerns you might have, it's now time to move on and start teaching assembly language programming using HLA.

1.2   Intended Audience

No single textbook can be all things to all people. This text is no exception. I've geared this text and the accompanying software to University level students who've never previously learned assembly language programming. This is not to say that others cannot benefit from this work; it simply means that as I've had to make choices about the presentation, I've made choices that should prove most comfortable for this audience I've chosen.

A secondary audience who could benefit from this presentation is any motivated person that really wants to learn assembly language. Although I assume a certain level of mathematical maturity from the reader (i.e., high school algebra), most of the "tough math" in this textbook is incidental to learning assembly language programming and you can easily skip over it without fear that you'll miss too much. High school students and those who haven't seen a school in 40 years have effectively used this text (and its DOS counterpart) to learn assembly language programming.

The organization of this text reflects the diverse audience for which it is intended. For example, in a standard textbook each chapter typically has its own set of questions, programming exercises, and laboratory exercises. Since the primary audience for this text is University students, such pedagogical material does appear within this text. However, recognizing that not everyone who reads this text wants to bother with this material (e.g., downloading it), this text moves such pedagogical material to the end of each volume in the text and places this material in a separate chapter. This is somewhat of an unusual organization, but I feel that University instructors can easily adapt to this organization and it saves burdening those who aren't interested in this material.

One audience to whom this book is specifically not directed are those persons who are already comfortable programming in 80x86 assembly language. Undoubtedly, there is a lot of material such programmers will find of use in this textbook. However, my experience suggests that those who've already learned x86 assembly language with an assembler like MASM, TASM, or NASM rebel at the thought of having to relearn basic assembly language syntax (as they would to have to learn HLA). If you fall into this category, I humbly apologize for not writing a text more to your liking. However, my goal has always been to teach those who don't already know assembly language, not extend the education of those who do. If you happen to fall into this category and you don't particularly like this text's presentation, there is some good news: there are dozens of texts on assembly language programming that use MASM and TASM out there. So you don't really need this one.

1.3   Teaching From This Text

The first thing any instructor will notice when reviewing this text is that it's far too large for any reasonable course. That's because assembly language courses generally come in two flavors: a machine organization course (more hardware oriented) and an assembly language programming course (more software oriented). No text that is "just the right size" is suitable for both types of classes. Combining the information for both courses, plus advanced information students may need after they finish the course, produces a large text, like this one.

If you're an instructor with a limited schedule for teaching this subject, you'll have to carefully select the material you choose to present over the time span of your course. To help, I've included some brief notes

at the beginning of each Volume in this text that suggests whether a chapter in that Volume is appropriate for a machine organization course, an assembly language programming course, or an advanced assembly programming course. These brief course notes can help you choose which chapters you want to cover in your course.

If you would like to offer hard copies of this text in the bookstore for your students, I will attempt to arrange with some "Custom Textbook Publishing" houses to make this material available on an "as-requested" basis. As I work out arrangements with such outfits, I'll post ordering information on Webster (http://webster.cs.ucr.edu). If your school has a printing and reprographics department, or you have a local business that handles custom publishing, you can certainly request copyright clearance to print the text locally.

If you're not taking a formal course, just keep in mind that you don't have to read this text straight through, chapter by chapter. If you want to learn assembly language programming and some of the machine organization chapters seem a little too hardware oriented for your tastes, feel free to skip those chapters and come back to them later on, when you understand the need to learn this information.

1.4   Copyright Notice

The full contents of this text is copyrighted material. Here are the rights I hereby grant concerning this material. You have the right to

- Read this text on-line from the http://webster.cs.ucr.edu web site or any other approved web site.
- Download an electronic version of this text for your own personal use and view this text on your own personal computer.
- Make a single printed copy for your own personal use.

I usually grant instructors permission to use this text in conjunction with their courses at recognized academic institutions. There are two types of reproduction I allow in this instance: electronic and printed. I grant electronic reproduction rights for one school term; after which the institution must remove the electronic copy of the text and obtain new permission to repost the electronic form (I require a new copy for each term so that corrections, changes, and additions propagate across the net). If your institution has reproduction facilities, I will grant hard copy reproduction rights for one academic year (for the same reasons as above). You may obtain copyright clearance by emailing me at

<center>rhyde@cs.ucr.edu</center>

I will respond with clearance via email. My returned email plus this page should provide sufficient acknowledgement of copyright clearance. If, for some reason, your reproduction department needs to have me physically sign a copyright clearance, I will have to charge $75.00 U.S. to cover my time and effort needed to deal with this. To obtain such clearance, please email me at the address above. Presumably, your printing and reproduction department can handle producing a master copy from PDF files. If not, I can print a master copy on a laser printer (800x400dpi), please email me for the current cost of this service.

All other rights to this text are expressly reserved by the author. In particular, it is a copyright violation to

- Post this text (or some portion thereof) on some web site without prior approval.
- Reproduce this text in printed or electronic form for non-personal (e.g., commercial) use.

The software accompanying this text is all public domain material unless an explicit copyright notice appears in the software. Feel free to use the accompanying software in any way you feel fit.

1.5   How to Get a Hard Copy of This Text

This text is distributed in electronic form only. It is not available in hard copy form nor do I personally intend to have it published. If you want a hard copy of this text, the copyright allows you to print one for yourself. The PDF distribution format makes this possible (though the length of the text will make it somewhat expensive).

If you're wondering why I don't get this text published, there's a very simple reason: it's too long. Publishing houses generally don't want to get involved with texts for specialized subjects as it is; the cost of producing this text is prohibitive given its limited market. Rather than cut it down to the 500 or so 6" x 9" pages that most publishers would accept, my decision was to stick with the full text and release the text in electronic form on the Internet. The upside is that you can get a free copy of this text; the downside is that you can't readily get a hard copy.

Note that the copyright notice forbids you from copying this text for anything other than personal use (without permission, of course). If you run a "Print to Order/Custom Textbook" publishing house and would like to make copies for people, feel free to contact me and maybe we can work out a deal for those who just have to have a hard copy of this text.

1.6   Obtaining Program Source Listings and Other Materials in This Text

All of the software appearing in this text is available from the Webster web site. The URL is

http://webster.cs.ucr.edu

The exact filename(s) of this material may change with time, and different services use different names for these files. Check on Webster for any important changes in addresses. If for some reason, Webster disappears in the future, you should use a web-based search engine like "AltaVista" and search for "Art of Assembly" to locate the current home site of this material.

1.7   Where to Get Help

If you're reading this text and you've got questions about how to do something, please post a message to one of the following Internet newsgroups:

```
comp.lang.asm.x86
alt.lang.asm
```

Hundreds of knowledgeable individuals frequent these newsgroups and as long as you're not simply asking them to do your homework assignment for you, they'll probably be more than happy to help you with any problems that you have with assembly language programming.

I certainly welcome corrections and bug reports concerning this text at my email address. However, I regret that I do not have the time to answer general assembly language programming questions via email. I do provide support in public forums (e.g., the newsgroups above and on Webster at http://webster.cs.ucr.edu) so please use those avenues rather than emailing questions directly to me. Due to the volume of email I receive daily, I regret that I cannot reply to all emails that I receive; so if you're looking for a response to a question, the newsgroup is your best bet (not to mention, others might benefit from the answer as well).

1.8   Other Materials You Will Need (Windows Version)

In addition to this text and the software I provide, you will need a machine running a 32-bit version of Windows (Windows 9x, NT, 2000, ME, etc.), a copy of Microsoft's MASM and a 32-bit linker, some sort of

text editor, and other rudimentary general-purpose software tools you normally use. MASM and MS-Link are freely available on the internet. Alas, the procedure you must follow to download these files from Microsoft seems to change on a monthly basis. However, a quick post to comp.lang.asm.x86 should turn up the current site from which you may obtain this software. Almost all the software you need to use this text is part of Windows (e.g., a simple text editor like Notepad.exe) or is freely available on the net (MASM, LINK, and HLA). You shouldn't have to purchase anything.

1.9 Other Materials You Will Need (Linux Version)

In addition to this text and the software I provide, you will need a machine running Linux (preferably Linux 2.4 or later), "as" and "ld" (if you can compile GCC programs, you've got these, they come standard with most distributions), some sort of text editor, and other rudimentary general-purpose software tools you normally use. Although not necessary, it helps if you've got superuser priviledges during installation so you can put the software in a reasonable spot.

Hello, World of Assembly Language   Chapter Two

2.1   Chapter Overview

This chapter is a "quick-start" chapter that lets you start writing basic assembly language programs right away. This chapter presents the basic syntax of an HLA (High Level Assembly) program, introduces you to the Intel CPU architecture, provides a handful of data declarations and machine instructions, describes some utility routines you can call in the HLA Standard Library, and then shows you how to write some simple assembly language programs. By the conclusion of this chapter, you should understand the basic syntax of an HLA program and be prepared to start learning new language features in subsequent chapters.

2.2   Installing the HLA Distribution Package

Before you can learn assembly language programming using HLA, you must first successfully install HLA on your system. Currently, HLA is available for the Linux and Windows operating systems. This section explains how to install HLA on these two systems. If HLA is already running on your system, you may skip to the next major section in this chapter.

The latest version of HLA is available from the Webster web server at

<div align="center">http://webster.cs.ucr.edu</div>

Go to this web site and following the HLA links to the "HLA Download" page. From here you should select the latest version of HLA for download to your computer. The HLA distribution is provided in a "Zip File" compressed format. Under Windows, you will need a decompressor program like PKUNZIP or WinZip in order to extract the HLA files from this zipped archive file; under Linux, you will use the GZIP and TAR programs to decompress and extract HLA. A detailed description of the use of these decompression products is beyond the scope of this manual, please consult the software vendor's documentation or their web page for information concerning the use of these products; this discussion will only briefly describe how to use them to extract important HLA files.

This text assumes that you will unzip the HLA distribution into the root directory of your C: drive under Windows, or to the "/usr/hla" directory under Linux. You can certainly install HLA anywhere you want, but you will have to adjust the following descriptions if you install HLA somewhere else. If possible, you should install HLA using root/administrator priviledges; regardless, you should make sure the permissions are set properly on the files so everyone has read and execute access to the HLA files; if you are unsure how to do this, please consult your operating system's documentation or consult a system administrator.

HLA is a console application. In order to run the HLA compiler you must run the command window program (this is "command.com" on Windows 95 and 98, or "cmd.exe" on Windows NT and Windows 2000; Linux users typically run "bash" or some other shell). This also means that you should be familiar with some simple "command line interface" (CLI) or "shell" commands.

Most Windows distributions let you run the command prompt windows from the Start menu or from a submenu hanging off the start menu (you may also select "RUN" from the Start menu and type "cmd" as the program name). This text assumes that you are familiar with the Windows command window and you know how to use some basic command window commands (e.g., dir, del, rename, etc.). If you have never before used the Windows command line interpreter, you should consult an appropriate text to learn a few basic commands.

Most Linux distributions run "bash" or some other shell program whenever you open up a terminal window (e.g., a GNOME or KDE terminal window or an X-TERM window). There are some minor differences between the shells running under Linux, this document assumes that you are using GNU's "bash" shell. Again, this text assumes that you are comfortable with a few commands like ls, rm, and mv. If you have never used a Unix shell program before, you should consult an appropriate text or the on-line documentation to learn a few basic commands.

Before you can actually run the HLA compiler, you must set the system execution path and set up various environment variables. The following subsections explain how to do this under Windows and then Linux.

2.2.1   Installation Under Windows

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as MASM, then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.lib) and, possibly, several operating system specific library files (e.g., kernel32.lib under Windows). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your Windows system.

First, you will need an HLA distribution for Windows. The latest version of HLA is always available on Webster at http://webster.cs.ucr.edu. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file you will need Microsoft's MASM and LINK programs, along with some Windows library files, to complete the process. The easiest way to get all the files you need is to download the "MASM32" package from http://www.pdq.com.au/home/hutch/masm.htm or any of the other places on the net where you can find the MASM32 package (Webster maintains a current link if this link is dead). Once you unzip this file, it's easy to install the MASM32 package using the install program it supplies. ***You must install MASM32 (or MASM/LINK/Win32 library files) before HLA will function properly***.

Here are the steps I went through to install MASM32 on my system:

- I downloaded masm32v6.zip from the URL above (later versions are probably okay too, although there is a slight chance that the installation will be different.
- I double-clicked on the masm32v6.zip file (which runs WinZip on my system).
- I choose to extract "install.exe". I told WinZip to extract this file to C:\.
- I double-clicked on the "install.exe" icon and selected the "C:" drive in the window that popped up. Then I hit the install button and waited while MASM32 extracted all the pertinent files. This produced a directory called "MASM32". MASM32 is a powerful assembly language development subsystem in its own right; but it uses the traditional MASM syntax rather than the HLA syntax. So we'll use MASM32 mainly for the assembler, linker, and library files. MASM32 also includes a simple editor/IDE and several other tools that may be useful to an HLA programmer. Feel free to check this software out and see if it is useful to you. For now, note that the executable files you will ultimately need are ML.EXE, ML.ERR, LINK.EXE, and a couple of DLLs. You can find them in the MASM32\BIN subdirectory. Leave them there for the time being. The MASM32\LIB directory also contains many Win32 library files you will need. Again, leave them alone for the time being.
- Next, if you haven't already done so, download the HLA executables file from Webster at http://webster.cs.ucr.edu. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).
- I downloaded the HLA1_32.zip file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my "C:\" root directory.

- After downloading HLA1_32.zip to my C: drive, I double-clicked on the icon to run WinZip. I selected "Extract" and told WinZip to extract all the files to my C:\ directory. This created an "HLA" subdirectory in my root on C: with two subdirectories (include and lib) and two EXE files (HLA.EXE and HLAPARSE.EXE. The HLA program is a "shell" program that runs the HLA compiler (HLAPARSE.EXE), MASM (ML.EXE), the linker (LINK.EXE), and other programs. You can think of HLA.EXE as the "HLA Compiler".
- Next, I created the following text file and named it "IHLA.BAT" (note that you may need to change the default drive letters if you want to install HLA on a drive other than "C:"):

```
path=c:\hla;c:\masm32\bin;%path%
set lib=c:\masm32\lib;c:\hla\hlalib;%lib%
set include=c:\hla\include;c:\masm32\include;%include%
set hlainc=c:\hla\include
set hlalib=c:\hla\hlalib\hlalib.lib
```

- Be sure you've typed all the lines exactly as written or HLA will fail to run properly. You may use any reasonable TEXT editor (e.g., NOTEPAD.EXE) to create this file. Do not use a word processing program (since they generally don't save their data as a TEXT file). Be sure the file is named "IHLA.BAT" and not "IHLA.BAT.TXT" or some other variation.
- This batch file tells the system where to find all the files you will need when running HLA. Advanced Win32 users should note that you can set all these environment variables up inside the Windows system control panel in the "Advanced->Environment Variables" area. This is far more convenient (ultimately) than using this batch file (for reasons you'll soon see). However, you can mess up you system if you don't know what you're doing when playing with the system control panel, so only advanced users who've done this stuff before should attempt this.
- HLA is a Win32 Console Window program. To run HLA you must open up a console Window. Under Windows 2000, Microsoft has hidden this away in Start->Programs->Accessories->Command Prompt. You might find it in another location. You can also start the command prompt processor by selecting Start->Run and entering "cmd".
- Once you've got the command prompt, ("C:>" or something similar), execute the IHLA.BAT file you've created by typing "IHLA" at the command line prompt. Hit the ENTER key to execute the command.
- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).
- Thus far, you've verified that HLA.EXE is operational. Now try the following command: "ML /?" This should run the Microsoft Macro Assembler (MASM) and display the help screen. You can ignore the information that appears; you will probably never need to know this stuff.
- Next, let's verify the correct operation of the linker. Type "link /?" and verify that the linker program runs. Again, you can ignore the help screen that appears. You don't need to know about this stuff.
- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we will revisit this program a little later in this chapter, don't worry about what it means just yet). Enter it into a text editor and save it using the filename "HW.HLA":

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

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

end HelloWorld;
```

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

```
HLA (High Level Assembler)
Written by Randall Hyde and released to the public domain.
Version Version 1.32 build 4904 (prototype)

Files:
1: hw.hla

Compiling "hw.hla" to "hw.asm"

Assembling hw.asm via "ml /c /coff /Cp  hw.asm"

Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997. All rights reserved.

 Assembling: hw.asm
Linking via "link -subsystem:console  /heap:0x1000000,0x1000000
/stack:0x1000000,0x1000000 /BASE:0x3000000 /machine:IX86  -entry:?HLAMain @hw.link
-out:hw.exe kernel32.lib user32.lib  c:\hla\hlalib\hlalib.lib hw.obj"
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/section:.text,ER
/section:readonly,R
/section:.edata,R
/section:.data,RW
/section:.bss,RW
```

- If you get all of this output, you're in business. You can run the "HW" program using the following CLI (command line interpreter) command:

```
HW
```

- One thing to remember is that unless you set the environment variables permanently in the System control panel, you will have to run the IHLA.BAT file every time you open up a new command prompt window. Since this is a pain, here are some instructions I've taken from the Internet that describe how to set up the environment variables (*DO THIS AT YOUR OWN RISK*!)

1) Open System Properties (Winkey-Break is a convenient shortcut) and go to Advanced tab, then Environment Variables. Add "c:\hla" to the Path in SYSTEM VARIABLES, not in "User variables for <your win2k login name>". Click OK, but keep the Environment Variables window open, we're not done.

2) Look at the contents of ihla.bat (ABOVE):

3) In "User Variables for <your login name>", you must end up with each of these settings. For example, to create hlainc, you click the "New..." button, type "hlainc" as the name of the variable, and type "c:\hla\include" as the Variable value (all without quotes of course). If there is already a path set, and it already has some value, add this immediately to the end: ";c:\hla;%path%" and that will preserve your existing User and System paths as well as adding c:\hla.

For example, suppose you opened up your User Variables for <login name> and it already said "C:\Private

Files\PantiePix;c:\winnt\system32;c:\winnt;c:\winnt\System32\Wbem;d:\lcc\bin;D:\PROGRA~1\ULTRAE~1;D:\4NT300;C:\msoffice\Office;c:/hla",

you would click on Edit and type "C:\Private Files\PantiePix;c:\hla;%path%"

(Same advice for preserving existing lib and include settings)

4)   Once you reboot the computer, you should be all set for "Hello world of assembly language"! (without having to run the IHLA.BAT file.)

Installing HLA is a complex and slightly involved process. Unfortunately, this is necessary because I don't have the rights to distribute MASM, LINK, and other Microsoft files. Fortunately, HUTCH has collected all of these files together so they are easy to download. If you are concerned about possible legal issues with the download, you may legally download MASM and LINK from Microsoft's site. A link on Webster (at the URL above) describes how to do this. At the time this was being written, work was progressing on HLA to produce TASM compatible output and plans were in the works to produce NASM and Gas versions as well. However, you will still have to obtain the Microsoft library files from some source if you intend to produce a Win32 application. Versions of HLA may appear for other Operating Systems as well. Check out Webster to see if any progress has been made in this direction.

The most common two problems people have running HLA involve the location of the Win32 library files and the choice of linker. During the linking phase, HLA (well, link.exe actually) requires the kernel32.lib, user32.lib, and gdi32.lib library files. These must be present in the pathname(s) specified by the LIB environment variable. If, during the linker phase, HLA complains about missing object modules, make sure that the LIB path specifies the directory containing these files. If you're a MS VC++ user, installation of VC++ should have set up the LIB path for you. If not, then locate these files (they are part of the MASM32 distribution) and copy them to the HLA\HLALIB directory (note that the ihla.bat file includes c:\hla\hlalib as part of the LIB path).

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

2.2.2 Installation Under Linux

HLA is not a stand alone program. It is a compiler that translates HLA source code into a lower-level assembly language. A separate assembler, such as Gas (as), then completes the processing of this low-level intermediate code to produce an object code file. Finally, you must link the object code output from the assembler using a linker program. Typically you will link the object code produced by one or more HLA source files with the HLA Standard Library (hlalib.a). Most of this activity takes place transparently whenever you ask HLA to compile your HLA source file(s). However, for the whole process to run smoothly, you must have installed HLA and all the support files correctly. This section will discuss how to set up HLA on your system.

First, you will need an HLA distribution for Linux. The latest version of HLA is always available on Webster at http://webster.cs.ucr.edu. You should go there and download the latest version if you do not already possess it.

As noted earlier, HLA is not a stand alone assembler. The HLA package contains the HLA compiler, the HLA Standard Library, and a set of include files for the HLA Standard Library. If you write an HLA program with just this code, HLA will produce an "ASM" file and then stop. To produce an executable file

you will need GNU's as and ld programs (these come with any Linux distribution that supports compiling C/C++ programs). Note that HLA only works with Gas v2.10 or later. The Gas assembler is part of the Binutils package. If you don't have version 2.10 or later, download an appropriate binutils package from the internet. HLA will generate errors when it attempts to assemble its output via an invocation of the as (Gas) executable if you don't have Gas v2.10 or later installed in your system.

Here are the steps I went through to install HLA on my Linux system:

- First, if you haven't already done so, download the HLA executables file from Webster at http://webster.cs.ucr.edu. On Webster you can download several different ZIP files associated with HLA from the HLA download page. The "Linux Executables" is the only one you'll absolutely need; however, you'll probably want to grab the documentation and examples files as well. If you're curious, or you want some more example code, you can download the source listings to the HLA Standard Library. If you're *really* curious (or masochistic), you can download the HLA compiler source listings to (this is *not* for casual browsing!).

- I downloaded the HLA1_39.tar.gz file while writing this. Most likely, there is a much later version available as you're reading this. Be sure to get the latest version. I chose to download this file to my root directory; you can put the file whereever you like, though this documentation assumes that all HLA files wind up in the "/usr/hla/..." directory tree. If you do not already have a "/usr/hla" subdirectory, you can create one with the "mkdir" command (it's best to do this using the "root" or "superuser" account; if you do not have superuser priviledges, you should have your system administrator do this for you.

- After downloading HLA1_39.tar.gz to my root directory, I executed the following shell command: "gzip -d HLA1_39.tar.gz". Once decompression was complete, I extracted the individual files using the command "tar xvf HLA1_39.tar". This extracted a couple of executable files ("hla" and "hlaparse") along with two subdirectories (include and hlalib). The HLA program is a "shell" program that runs the HLA compiler (hlaparse), Gas (as), the linker (ld), and other programs. You can think of "hla" as the "HLA Compiler". It would be a real good idea, at this point, to set the permissions on "hla" and "hlaparse" so that everyone can read and execute them. You should also set read and execute permissions on the two subdirectories and read permissions on all the files within the directories (if this isn't the default state). Do a "man chmod" from the Linux command-line if you don't know how to change permissions.

- Next, (logged in as a plain user rather than root or the super-user), I edited the ".bashrc" file in my home directory ("/home/rhyde" in my particular case, this will probably be different for you). I found the line that defined the "path" variable, it originally looked like this on my system

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

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

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

Without this modification, Linux will probably not find HLA when you attempt to execute it unless you type a full path (e.g., "/usr/hla/hla") when running the program. Since this is a pain, you'll definitely want to add "/usr/hla" to your path.

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

        hlalib=/usr/hla/hlalib/hlalib.a
        export hlalib
        hlainc=/usr/hla/include
        export hlainc

These four lines define (and export) environment variables that HLA needs during compilation. Without these environment variables, HLA will probably complain about not being able to find include files, or the linker (ld) will complain about strange undefined symbols when you attempt to compile your programs.

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

        source .bashrc

Note: this discussion only applies to users who run the BASH shell. If you are using a different shell (like the C-Shell or the Korn Shell), then the directions for setting the path and environment variables differs slightly. Please see the documentation for your particular shell if you don't know how to do this. Also note that Linux does not normally display files whose name begins with a period when you use the "ls" command; to see such files, use the "ls -a" shell command.

- At this point, HLA should be properly installed and ready to run. Try typing "hla -?" at the command line prompt and verify that you get the HLA help message. If not, go back and figure out what you've done wrong up to this point (it doesn't hurt to start over from the beginning if you're lost).

- Now it's time to try your hand at writing an honest to goodness HLA program and verify that the whole system is working. Here's the canonical "Hello World" program written in HLA (we'll discuss this program in detail a little later in this chapter). Enter it into a text editor and save it using the filename "hw.hla":

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

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

end HelloWorld;
```

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

```
HLA (High Level Assembler) Parser
Written by Randall Hyde and released to the public domain.
Version Version 1.39 build 6845 (prototype)
-t active
File: t.hla

Compiling "t.hla" to "t.asm"
HLA (High Level Assembler)
Copyright 1999, by Randall Hyde, all rights reserved.
Version Version 1.39 build 6845 (prototype)
ELF output
Using GAS assembler
GAS output
-test active

Files:
1: t.hla

Compiling 't.hla' to 't.asm'
using command line [hlaparse  -v -sg -test "t.hla"]

Assembling "t.asm" via [as -o t.o  "t.asm"]
Linking via [ld   -o "t"  "t.o" "/usr/hla/hlalib/hlalib.a"]
```

Installing HLA is a complex and slightly involved process; though take heart, it's a lot simpler to install HLA under Linux than Windows! (See the previous section if you need proof.) Versions of HLA may appear for other operating systems (beyond Windows and Linux) as well. Check out Webster to see if any progress has been made in this direction. Note a very unique thing about HLA: Carefully written (console) applications will compile and run on all supported operating systems without change. This is unheard of for

assembly language! So if you are using multiple operating systems supported by HLA, you'll probably want to download files for all supported OSes.

Note: to run the HelloWorld program, a Linux user would type "hw" (or possibly "./hw") at the command line prompt.

2.2.3 Installing "Art of Assembly" Related Files

Although HLA is relatively flexible about where you put it on your system, this text assumes you've installed HLA in the "hla" directory on your C: drive under a Win32 operating system or in "/usr/hla" under Linux. This text also assumes the standard directory placement for the HLA files, which has the following layout

- HLA directory
- AoA directory
- Doc directory
- Examples directory
- hlalib directory
- hlalibsrc directory
- include directory
- Tests directory

The "Art of Assembly" (AoA) software distribution has the following directory tree structure:

- AoA directory
- volume1
- ch01 directory
- ch02 directory
- etc.
- volume2
- ch01 directory
- ch02 directory
- etc.
- etc.

The main *HLA* directory contains the executable code for the compiler. This consists of two files, HLA.EXE/hla and HLAPARSE.EXE/hlaparse (Windows/Linux). These two programs must be in the current execution path in order to run the compiler. Under Windows, it wouldn't hurt to put the ml.exe, ml.err, link.exe, mspdbX0.dll (x=5, 6, or greater), and msvcrt.dll files in this directory as well. Under Linux, the "as" and "ld" programs are already in the execution path, assuming your Linux system supports C/C++ development.

The *Doc* directory contains reference material for HLA in PDF and HTML formats. If you have a copy of Adobe Acrobat Reader, you will probably want to read the PDF versions since they are much nicer than the HTML versions. These documents contain the most up-to-date information about the HLA language; you should consult them if you have a question about the HLA language or the HLA Standard Library. Generally, material in this documentation supersedes information appearing in this text since the HLA document is electronic and is probably more up to date.

The *Examples* directory contains a large set of HLA programs that demonstrate various features in the HLA language. If you have a question about an HLA feature, you can probably find an example program that demonstrates that feature in the *Examples* directory. Such examples provide invaluable insight that is often superior to a written description of the feature. Note that some of these programs may be specific to Windows or Linux, not all will compile and run under either operating system.

The *hlalib* directory contains the object code for the HLA Standard Library. As you become more competent with HLA, you may want to take a look at how HLA implements various library functions by checking out the library source code in the *hlalibsrc* subdirectory.
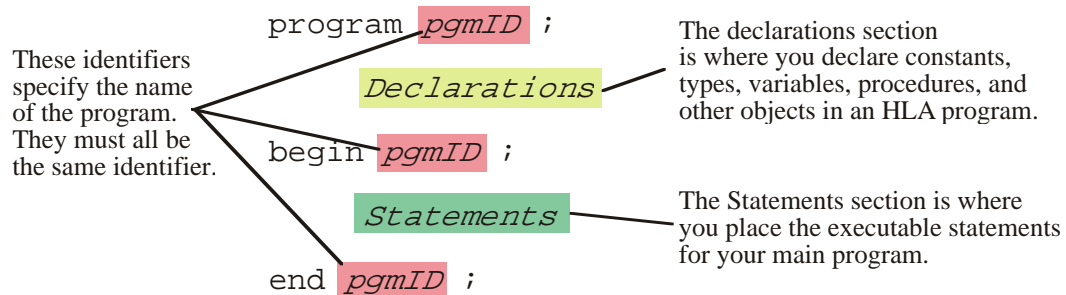
The *include* directory contains the HLA Standard Library include files. These special files (that end with a ".hhf" suffix, for "HLA Header File") are needed during assembly to provide prototype and other information to your program. The example programs in this chapter all include the HLA header file "stdlib.hhf" that, in turn, includes all the other HLA header files in the standard library.

The *Tests* directory contains various test files that test the correct operation of the HLA system. HLA includes these files as part of the distribution package because they provide additional examples of HLA coding.

The *AoA* directory contains the code specific to this textbook. This directory contains all the source code to the (complete) programs appearing in this text. It also contains the programs appearing in the Laboratory Exercises section of each chapter. Therefore, this directory is very important to you. Within this subdirectory, the information is further divided up by volume and chapter. The material for Chapter One appears in the "ch01" subdirectory of the "volume1" directory in the AoA directory tree, the material for Chapter Two appears in the "ch02" subdirectory of the "volume1" directory, etc..

2.3   The Anatomy of an HLA Program

An HLA program typically takes the following form:

These identifiers specify the name of the program. They must all be the same identifier.

```
program pgmID ;

        Declarations

begin pgmID ;

        Statements

end pgmID ;
```

The declarations section is where you declare constants, types, variables, procedures, and other objects in an HLA program.

The Statements section is where you place the executable statements for your main program.

PROGRAM, BEGIN, and END are HLA reserved words that delineate the program. Note the placement of the semicolons in this program.

Figure 2.1   Basic HLA Program Layout

The *pgmID* in the template above is a user-defined program identifier. You must pick an appropriate, descriptive, name for your program. In particular, *pgmID* would be a horrible choice for any real program. If you are writing programs as part of a course assignment, your instructor will probably give you the name to use for your main program. If you are writing your own HLA program, you will have to choose this name.

Identifiers in HLA are very similar to identifiers in most high level languages. HLA identifiers may begin with an underscore or an alphabetic character, and may be followed by zero or more alphanumeric or underscore characters. HLA's identifiers are *case neutral*. This means that the identifiers are case sensitive insofar as you must always spell an identifier exactly the same way in your program (even with respect to upper and lower case). However, unlike other case sensitive languages, like C/C++, you may not declare two identifiers in the program whose name differs only by the case of alphabetic characters appearing in an identifier. Case neutrality enforces the good programming style of always spelling your names exactly the same

way (with respect to case) and never declaring two identifiers whose only difference is the case of certain alphabetic characters.

A traditional first program people write, popularized by K&R's "The C Programming Language" is the "Hello World" program. This program makes an excellent concrete example for someone who is learning a new language. Here's what the "Hello World" program looks like in HLA:

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

begin helloWorld;

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

end helloWorld;
```

Program 2.1   The Hello World Program

The #include statement in this program tells the HLA compiler to include a set of declarations from the stdlib.hhf (standard library, HLA Header File). Among other things, this file contains the declaration of the *stdout.put* code that this program uses.

The *stdout.put* statement is the "print" statement for the HLA language. You use it to write data to the standard output device (generally the console). To anyone familiar with I/O statements in a high level language, it should be obvious that this statement prints the phrase "Hello, World of Assembly Language". The *nl* appearing at the end of this statement is a constant, also defined in "stdlib.hhf", that corresponds to the newline sequence.

Note that semicolons follow the program, BEGIN, *stdout.put*, and END statements[1]. Technically speaking, a semicolon is not necessary after the #INCLUDE statement. It is possible to create include files that generate an error if a semicolon follows the #INCLUDE statement, so you may want to get in the habit of not putting a semicolon here (note, however, that the HLA standard library include files always allow a semicolon after the corresponding #INCLUDE statement).

The #INCLUDE is your first introduction to HLA declarations. The #INCLUDE itself isn't actually a declaration, but it does tell the HLA compiler to substitute the file "stdlib.hhf" in place of the #INCLUDE directive, thus inserting several declarations at this point in your program. Most HLA programs you will write will need to include at least some of the HLA Standard Library header files ("stdlib.hhf" actually includes all the standard library definitions into your program; for more efficient compiles, you might want to be more selective about which files you include. You will see how to do this in a later chapter).

Compiling this program produces a *console* application. Running this program in a command window prints the specified string and then control returns back to the command line interpreter (or *shell* in Unix terminology).

Note that HLA is a free-format language. Therefore, you may split statement across multiple lines (just like high level languages) if this helps to make your programs more readable. For example, the *stdout.put* statement in the HelloWorld program could also be written as follows:

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

_____

1. Technically, from a language design point of view, these are not all statements. However, this chapter will not make that distinction.

Another item worth noting, since you'll see it cropping up in example code throughout this text, is that HLA automatically concatenates any adjacent string constants it finds in your source file. Therefore, the statement above is also equivalent to:

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

Indeed, "nl" (the newline) is really nothing more than a string constant, so (technically) the comma between the *nl* and the preceding string isn't necessary. You'll often see the above written as:
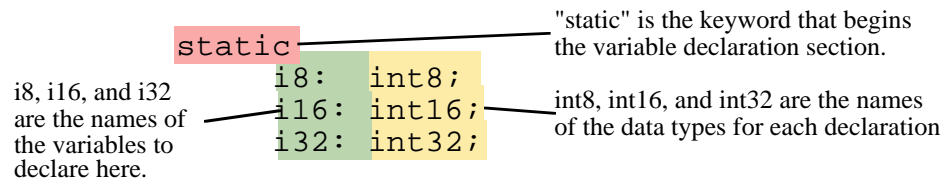
```
stdout.put( "Hello, World of Assembly Language" nl );
```

Notice the lack of a comma between the string constant and nl; this turns out to be perfectly legal in HLA, though it only applies to certain symbol string constants; you may not, in general, drop the comma. The chapter on Strings, later in this text, will explain in detail how this works. This discussion appears here because you'll probably see this "trick" employed by sample code prior to the formal discussion in the chapter on Strings.

2.4   Some Basic HLA Data Declarations

HLA provides a wide variety of constant, type, and data declaration statements. Later chapters will cover the declaration section in more detail but it's important to know how to declare a few simple variables in an HLA program.

HLA predefines three different signed integer types: *int8*, *int16*, and *int32*, corresponding to eight-bit (one byte) signed integers, 16-bit (two byte) signed integers, and 32-bit (four byte) signed integers respectively[2]. Typical variable declarations occur in the HLA *static variable section*. A typical set of variable declarations takes the following form

Figure 2.2   Static Variable Declarations

Those who are familiar with the Pascal language should be comfortable with this declaration syntax. This example demonstrates how to declare three separate integers, *i8, i16,* and *i32*. Of course, in a real program you should use variable names that are a little more descriptive. While names like "i8" and "i32" describe the type of the object, they do not describe it's purpose. Variable names should describe the purpose of the object.

In the STATIC declaration section, you can also give a variable an initial value that the operating system will assign to the variable when it loads the program into memory. The following figure demonstrates the syntax for this:

---

2. A discussion of bits and bytes will appear in the next chapter if you are unfamiliar with these terms.

```
static
        i8:   int8  := 8;
        i16:  int16 := 1600;
        i32:  int32 := -320000;
```

The constant assignment operator, ":=" tells HLA that you wish to initialize the specified variable with an initial value.

The operand after the constant assignment operator must be a constant whose type is compatible with the variable you are initializing

Figure 2.3   Static Variable Initialization

It is important to realize that the expression following the assignment operator (":=") must be a constant expression. You cannot assign the values of other variables within a STATIC variable declaration.

Those familiar with other high level languages (especially Pascal) should note that you may only declare one variable per statement. That is, HLA does not allow a comma delimited list of variable names followed by a colon and a type identifier. Each variable declaration consists of a single identifier, a colon, a type ID, and a semicolon.

Here is a simple HLA program that demonstrates the use of variables within an HLA program:

```
Program DemoVars;
#include( "stdlib.hhf" );

static
    InitDemo:       int32 := 5;
    NotInitialized: int32;

begin DemoVars;

    // Display the value of the pre-initialized variable:

    stdout.put( "InitDemo's value is ", InitDemo, nl );

    // Input an integer value from the user and display that value:

    stdout.put( "Enter an integer value: " );
    stdin.get( NotInitialized );
    stdout.put( "You entered: ", NotInitialized, nl );

end DemoVars;
```

Program 2.2   Variable Declaration and Use

In addition to STATIC variable declarations, this example introduces three new concepts. First, the *stdout.put* statement allows multiple parameters. If you specify an integer value, *stdout.put* will convert that value to the string representation of that integer's value on output. The second new feature this sample program introduces is the *stdin.get* statement. This statement reads a value from the standard input device (usually the keyboard), converts the value to an integer, and stores the integer value into the *NotInitialized* variable. Finally, this program also introduces the syntax for (one form of) HLA comments. The HLA compiler ignores all text from the "//" sequence to the end of the current line. Those familiar with C++ and Delphi should recognize these comments.

2.5   Boolean Values

HLA and the HLA Standard Library provides limited support for boolean objects. You can declare boolean variables, use boolean literal constants, use boolean variables in boolean expressions (e.g., in an IF statement), and you can print the values of boolean variables.

Boolean literal constants consist of the two predefined identifiers *true* and *false* . Internally, HLA represents the value true using the numeric value one; HLA represents false using the value zero. Most programs treat zero as false and anything else as true, so HLA's representations for *true* and *false* should prove sufficient.

To declare a boolean variable, you use the *boolean* data type. HLA uses a single byte (the least amount of memory it can allocate) to represent boolean values. The following example demonstrates some typical declarations:

```
static
    BoolVar:   boolean;
    HasClass:  boolean := false;
    IsClear:   boolean := true;
```

As you can see in this example, you may declare initialized as well as uninitialized variables.

Since boolean variables are byte objects, you can manipulate them using eight-bit registers and any instructions that operate directly on eight-bit values. Furthermore, as long as you ensure that your boolean variables only contain zero and one (for false and true, respectively), you can use the 80x86 AND, OR, XOR, and NOT instructions to manipulate these boolean values (we'll describe these instructions a little later).

You can print boolean values by making a call to the *stdout.put* routine, e.g.,

```
            stdout.put( BoolVar )
```

This routine prints the text "true" or "false" depending upon the value of the boolean parameter (zero is false, anything else is true). Note that the HLA Standard Library does not allow you to read boolean values via *stdin.get*.

2.6   Character Values

HLA lets you declare one-byte ASCII character objects using the *char* data type. You may initialize character variables with a literal character value by surrounding the character with a pair of apostrophes. The following example demonstrates how to declare and initialize character variables in HLA:

```
static
    c: char;
    LetterA: char := 'A';
```

You can print character variables using the *stdout.put* routine. We'll return to the subject of character constants a little later.

2.7   An Introduction to the Intel 80x86 CPU Family

Thus far, you've seen a couple of HLA programs that will actually compile and run. However, all the statements utilized to this point have been either data declarations or calls to HLA Standard Library routines. There hasn't been any *real* assembly language up to this point. Before we can progress any farther and learn

some real assembly language, a detour is necessary. For unless you understand the basic structure of the Intel 80x86 CPU family, the machine instructions will seem mysterious indeed.

The Intel CPU family is generally classified as a *Von Neumann Architecture Machine*. Von Neumann computer systems contain three main building blocks: the *central processing unit* (CPU), *memory*, and *input/output devices* (I/O). These three components are connected together using the *system bus*. The following block diagram shows this relationship:
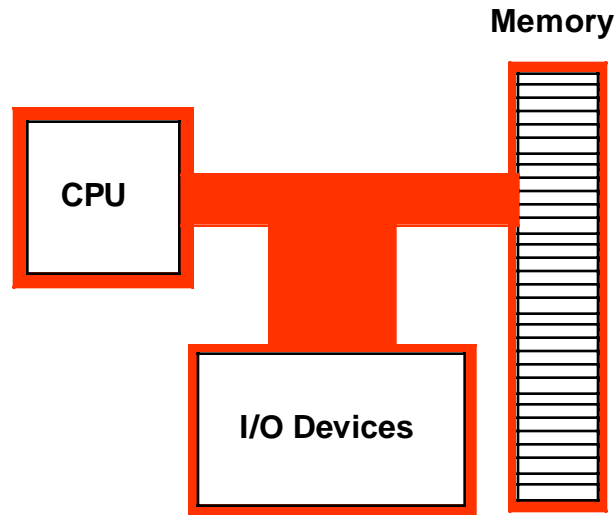

Figure 2.4   Von Neumann Computer System Block Diagram

Memory and I/O devices will be the subjects of later chapters; for now, let's take a look inside the CPU portion of the computer system, at least at the components that are visible to the assembly language programmer.

The most prominent items within the CPU are the registers. The Intel CPU registers can be broken down into four categories: general purpose registers, special purpose application accessible registers, segment registers, and special purpose kernel mode registers. This text will not consider the last two sets of registers. The segment registers are not used much in modern 32-bit operating systems (e.g., Windows, BeOS, and Linux); since this text is geared around programs written for 32-bit operating systems, there is little need to discuss the segment registers. The special purpose kernel mode registers are intended for use by people who write operating systems, debuggers, and other system level tools. Such software construction is well beyond the scope of this text, so once again there is little need to discuss the special purpose kernel mode registers.

The 80x86 (Intel family) CPUs provide several general purpose registers for application use. These include eight 32-bit registers that have the following names:

<div align="center">EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP</div>

The "E" prefix on each name stands for *extended*. This prefix differentiates the 32-bit registers from the eight 16-bit registers that have the following names:

<div align="center">AX, BX, CX, DX, SI, DI, BP, and SP</div>

Finally, the 80x86 CPUs provide eight 8-bit registers that have the following names:

<div align="center">AL, AH, BL, BH, CL, CH, DL, and DH</div>

Unfortunately, these are not all separate registers. That is, the 80x86 does not provide 24 independent registers. Instead, the 80x86 overlays the 32-bit registers with the 16-bit registers and it overlays the 16-bit registers with the 8-bit registers. The following diagram shows this relationship:
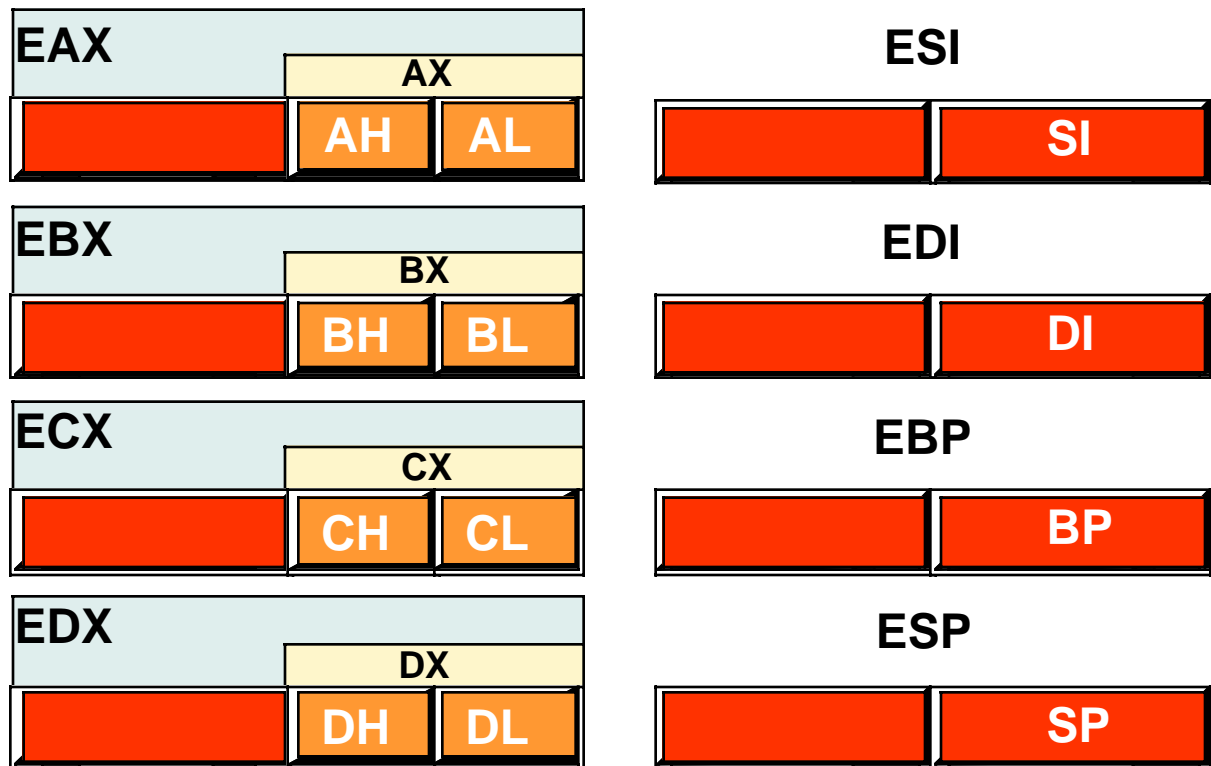
Figure 2.5   80x86 (Intel CPU) General Purpose Registers

The most important thing to note about the general purpose registers is that they are not independent. Modifying one register will modify at least one other register and may modify as many as three other registers. For example, modification of the EAX register may very well modify the AL, AH, and AX registers as well. This fact cannot be overemphasized here. A very common mistake in programs written by beginning assembly language programmers is register value corruption because the programmer did not fully understand the ramifications of the above diagram.

The EFLAGS register is a 32-bit register that encapsulates several single-bit boolean (true/false) values. Most of the bits in the EFLAGs register are either reserved for kernel mode (operating system) functions, or are of little interest to the application programmer. Eight of these bits (or *flags*) are of interest to application programmers writing assembly language programs. These are the overflow, direction, interrupt disable[3], sign, zero, auxiliary carry, parity, and carry flags. The following diagram shows their layout within the lower 16-bits of the EFLAGS register.

---

3. Application programs cannot modify the interrupt flag, but we'll look at this flag later in this text, hence the discussion of this flag here.

15   0

Overflow
Direction
Interrupt

Sign
Zero

Auxiliary Carry

Parity

Carry

Not very
interesting to
application
programmers

Figure 2.6   Layout of the FLAGS Register (Lower 16 bits of EFLAGS)
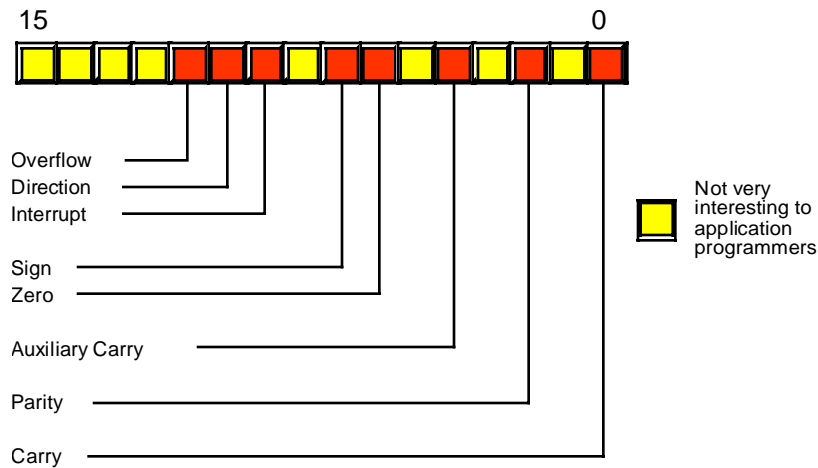
Of the eight flags that are usable by application programmers, four flags in particular are extremely valuable: the overflow, carry, sign, and zero flags. Collectively, we will call these four flags the *condition codes*[4]. The state of these flags (boolean variables) will let you test the results of previous computations and allow you to make decisions in your programs. For example, after comparing two values, the state of the condition code flags will tell you if one value is less than, equal to, or greater than a second value. The 80x86 CPUs provide special machine instructions that let you test the flags, alone or in various combinations.

The last register of interest is the EIP (instruction pointer) register. This 32-bit register contains the *memory address* of the next machine instruction to execute. Although you will manipulate this register directly in your programs, the instructions that modify its value treat this register as an implicit operand. Therefore, you will not need to remember much about this register since the 80x86 instruction set effectively hides it from you.

One important fact that comes as a surprise to those just learning assembly language is that almost all calculations on the 80x86 CPU must involve a register. For example, to add two (memory) variables together, storing the sum into a third location, you must load one of the memory operands into a register, add the second operand to the value in the register, and then store the register away in the destination memory location. Registers are a *middleman* in nearly every calculation. Therefore, registers are very important in 80x86 assembly language programs.

Another thing you should be aware of is that although the general purpose registers have the name "general purpose" you should not infer that you can use any register for any purpose. The SP/ESP register for example, has a very special purpose (it's the *stack pointer*) that effectively prevents you from using it for any other purpose. Likewise, the BP/EBP register has a special purpose that limits its usefulness as a general purpose register. All the 80x86 registers have their own special purposes that limit their use in certain contexts. For the time being, you should simply avoid the use of the ESP and EBP registers for generic calculations and keep in mind that the remaining registers are not completely interchangeable in your programs.

2.8   Some Basic Machine Instructions

The 80x86 CPUs provide just over a hundred to many thousands of different machine instructions, depending on how you define a machine instruction. Even at the low end of the count (greater than 100), it appears as though there are far too many machine instructions to learn in a short period of time. Fortunately,

4. Technically the parity flag is also a condition code, but we will not use that flag in this text.

you don't need to know all the machine instructions. In fact, most assembly language programs probably use around 30 different machine instructions[5]. Indeed, you can certainly write several meaningful programs with only a small handful of machine instructions. The purpose of this section is to provide a small handful of machine instructions so you can start writing simple HLA assembly language programs right away.

Without question, the MOV instruction is the most often-used assembly language statement. In a typical program, anywhere from 25-40% of the instructions are typically MOV instructions. As its name suggests, this instruction moves data from one location to another[6]. The HLA syntax for this instruction is

mov(*source_operand*, *destination_operand*);

The *source_operand* can be a register, a memory variable, or a constant. The *destination_operand* may be a register or a memory variable. Technically the 80x86 instruction set does not allow both operands to be memory variables; HLA, however, will automatically translate a MOV instruction with two 16- or 32-bit memory operands into a pair of instructions that will copy the data from one location to another. In a high level language like Pascal or C/C++, the MOV instruction is roughly equivalent to the following assignment statement:

destination_operand = source_operand ;

Perhaps the major restriction on the MOV instruction's operands is that they must both be the same size. That is, you can move data between two eight-bit objects, between two 16-bit objects, or between two 32-bit objects; you may not, however, mix the sizes of the operands. The following table lists all the legal combinations:

Table 1: Legal 80x86 MOV Instruction Operands

Source	Destination
Reg_8[a]	Reg_8
Reg_8	Mem_8
Mem_8	Reg_8
constant[b]	Reg_8
constant	Mem_8
Reg_{16}	Reg_{16}
Reg_{16}	Mem_{16}
Mem_{16}	Reg_{16}
constant	Reg_{16}
constant	Mem_{16}
Reg_{32}	Reg_{32}

---

5. Different programs may use a different set of 30 instructions, but few programs use more than 30 distinct instructions.
6. Technically, MOV actually copies data from one location to another. It does not destroy the original data in the source operand. Perhaps a better name for this instruction should have been COPY. Alas, it's too late to change it now.

Table 1: Legal 80x86 MOV Instruction Operands

Reg_{32}	Mem_{32}
Mem_{32}	Reg_{32}
constant	Reg_{32}
constant	Mem_{32}

a. The suffix denotes the size of the register or memory location.
b. The constant must be small enough to fit in the specified destination
operand

You should study this table carefully. Most of the general purpose 80x86 instructions use this same syntax. Note that in addition to the forms above, the HLA MOV instruction lets you specify two memory operands as the source and destination. However, this special translation that HLA provides only applies to the MOV instruction; it does not generalize to the other instructions.

The 80x86 ADD and SUB instructions let you add and subtract two operands. Their syntax is nearly identical to the MOV instruction:

add(*source_operand*, *destination_operand*);

sub(*source_operand*, *destination_operand*);

The ADD and SUB operands must take the same form as the MOV instruction, listed in the table above[7]. The ADD instruction does the following:

destination_operand = *destination_operand* + *source_operand* ;

destination_operand += *source_operand*; // For those who prefer C syntax

Similarly, the SUB instruction does the calculation:

destination_operand = *destination_operand* - *source_operand* ;

destination_operand -= *source_operand* ; // For C fans.

With nothing more than these three instructions, plus the HLA control structures that the next section discusses, you can actually write some sophisticated programs. Here's a sample HLA program that demonstrates these three instructions:

```
program DemoMOVaddSUB;

#include( "stdlib.hhf" );

static
    i8:     int8    := -8;
    i16:    int16   := -16;
    i32:    int32   := -32;

begin DemoMOVaddSUB;

    // First, print the initial values
    // of our variables.

    stdout.put
    (
        nl,
```

---

7. Remember, though, that ADD and SUB do not support memory-to-memory operations.

```
            "Initialized values: i8=", i8,
            ", i16=", i16,
            ", i32=", i32,
            nl
        );

        // Compute the absolute value of the
        // three different variables and
        // print the result.
        // Note, since all the numbers are
        // negative, we have to negate them.
        // Using only the MOV, ADD, and SUB
        // instruction, we can negate a value
        // by subtracting it from zero.

        mov( 0, al );   // Compute i8 := -i8;
        sub( i8, al );
        mov( al, i8 );

        mov( 0, ax );   // Compute i16 := -i16;
        sub( i16, ax );
        mov( ax, i16 );

        mov( 0, eax );  // Compute i32 := -i32;
        sub( i32, eax );
        mov( eax, i32 );

        // Display the absolute values:

        stdout.put
        (
            nl,
            "After negation: i8=", i8,
            ", i16=", i16,
            ", i32=", i32,
            nl
        );

        // Demonstrate ADD and constant-to-memory
        // operations:

        add( 32323200, i32 );
        stdout.put( nl, "After ADD: i32=", i32, nl );


    end DemoMOVaddSUB;
```

Program 2.3   Demonstration of MOV, ADD, and SUB Instructions

---

2.9   Some Basic HLA Control Structures

The MOV, ADD, and SUB instructions, while valuable, aren't sufficient to let you write meaningful programs. You will need to complement these instructions with the ability to make decisions and create loops in your HLA programs before you can write anything other than a trivial program. HLA provides several high level control structures that are very similar to control structures found in high level languages. These

include IF..THEN..ELSEIF..ELSE..ENDIF, WHILE..ENDWHILE, REPEAT..UNTIL, and so on. By learning these statements you will be armed and ready to write some real programs.

Before discussing these high level control structures, it's important to point out that these are not real 80x86 assembly language statements. HLA compiles these statements into a sequence of one or more real assembly language statements for you. Later in this text, you'll learn how HLA compiles the statements and you'll learn how to write pure assembly language code that doesn't use them. However, you'll need to learn many new concepts before you get to that point, so we'll stick with these high level language statements for now since you're probably already familiar with statements like these from your exposure to high level languages.

Another important fact to mention is that HLA's high level control structures are *not* as high level as they first appear. The purpose behind HLA's high level control structures is to let you start writing assembly language programs as quickly as possible, not to let you avoid the use of real assembly language altogether. You will soon discover that these statements have some severe restrictions associated with them and you will quickly outgrow their capabilities (at least the restricted forms appearing in this section). This is intentional. Once you reach a certain level of comfort with HLA's high level control structures and decide you need more power than they have to offer, it's time to move on and learn the real 80x86 instructions behind these statements.

2.9.1 Boolean Expressions in HLA Statements

Several HLA statements require a boolean (true or false) expression to control their execution. Examples include the IF, WHILE, and REPEAT..UNTIL statements. The syntax for these boolean expressions represents the greatest limitation of the HLA high level control structures. This is one area where your familiarity with a high level language will work against you – you'll want to use the same boolean expressions you use in a high level language and HLA only supports some basic forms.

HLA boolean expressions always take the following forms[8]:

<div align="center">

flag_specification

!flag_specification

register

!register

Boolean_variable

!Boolean_variable

mem_reg relop mem_reg_const

register in LowConst..HiConst

register not in LowConst..HiConst

</div>

A *flag_specification* may be one of the following symbols:

- @c   carry: True if the carry is set (1), false if the carry is clear (0).
- @nc   no carry: True if the carry is clear (0), false if the carry is set (1).
- @z   zero: True if the zero flag is set, false if it is clear.
- @nz   not zero: True if the zero flag is clear, false if it is set.
- @o   overflow: True if the overflow flag is set, false if it is clear.
- @no   no overflow: True if the overflow flag is clear, false if it is set.
- @s   sign: True if the sign flag is set, false if it is clear.
- @ns   no sign: True if the sign flag is clear, false if it is set.

---

8. Technically, there are a few more, advanced, forms, but you'll have to wait a few chapters before seeing these additional formats.

The use of the flag values in a boolean expression is somewhat advanced. You will begin to see how to use these boolean expression operands in the next chapter.

A register operand can be any of the 8-bit, 16-bit, or 32-bit general purpose registers. The expression evaluates false if the register contains a zero; it evaluates true if the register contains a non-zero value.

If you specify a boolean variable as the expression, the program tests it for zero (false) or non-zero (true). Since HLA uses the values zero and one to represent false and true, respectively, the test works in an intuitive fashion. Note that HLA requires that stand-alone variables be of type *boolean*. HLA rejects other data types. If you want to test some other type against zero/not zero, then use the general boolean expression discussed next.

The most general form of an HLA boolean expression has two operands and a relational operator. The following table lists the legal combinations:

Table 2: Legal Boolean Expressions

Left Operand	Relational Operator	Right Operand
Memory Variable or Register	= or ==	Memory Variable, Register, or Constant
	<> or !=	
	<	
	<=	
	>	
	>=	

Note that both operands cannot be memory operands. In fact, if you think of the Right Operand as the source operand and the Left Operand as the destination operand, then the two operands must be the same as those allowed for the ADD and SUB instructions.

Also like the ADD and SUB instructions, the two operands must be the same size. That is, they must both be eight-bit operands, they must both be 16-bit operands, or they must both be 32-bit operands. If the Right Operand is a constant, it's value must be in the range that is compatible with the Left Operand.

There is one other issue of which you need to be aware. If the Left Operand is a register and the Right Operand is a positive constant or another register, HLA uses an *unsigned* comparison. The next chapter will discuss the ramifications of this; for the time being, do not compare negative values in a register against a constant or another register. You may not get an intuitive result.

The IN and NOT IN operators let you test a register to see if it is within a specified range. For example, the expression "EAX in 2000..2099" evaluates true if the value in the EAX register is between 2000 and 2099 (inclusive). The NOT IN (two words) operator lets you check to see if the value in a register is outside the specified range. For example, "AL not in 'a'..'z'" evaluates true if the character in the AL register is not a lower case alphabetic character.

Here are some examples of legal boolean expressions in HLA:

@c

Bool_var

al

ESI

EAX < EBX

$$EBX > 5$$

$$i32 < -2$$

$$i8 > 128$$

$$al < i8$$

$$eax \text{ in } 1..100$$

$$ch \text{ not in } 'a'..'z'$$

2.9.2   The HLA IF..THEN..ELSEIF..ELSE..ENDIF Statement

The HLA IF statement uses the following syntax:

```
if( expression ) then

        sequence
        of one or
        more statements

elseif( expression ) then

        sequence
        of one or
        more statements

else

        sequence
        of one or
        more statements


endif;
```

The elseif clause is optional. Zero or more elseif clauses may appear in an if statement. If more than one elseif clause appears, all the elseif clauses must appear before the else clause (or before the endif if there is no else clause).

The else clause is optional. At most one else clause may appear within an if statement and it must be the last clause before the endif.

Figure 2.7   HLA IF Statement Syntax

The expressions appearing in this statement must take one of the forms from the previous section. If the associated expression is true, the code after the THEN executes, otherwise control transfers to the next ELSEIF or ELSE clause in the statement.

Since the ELSEIF and ELSE clauses are optional, an IF statement could take the form of a single IF..THEN clause, followed by a sequence of statements, and a closing ENDIF clause. The following is an example of just such a statement:

```
if( eax = 0 ) then

    stdout.put( "error: NULL value", nl );

endif;
```

If, during program execution, the expression evaluates true, then the code between the THEN and the ENDIF executes. If the expression evaluates false, then the program skips over the code between the THEN and the ENDIF.

Another common form of the IF statement has a single ELSE clause. The following is an example of an IF statement with an optional ELSE:

```
if( eax = 0 ) then

    stdout.put( "error: NULL pointer encountered", nl );

else

    stdout.put( "Pointer is valid", nl );

endif;
```

If the expression evaluates true, the code between the THEN and the ELSE executes; otherwise the code between the ELSE and the ENDIF clauses executes.

You can create sophisticated decision-making logic by incorporating the ELSEIF clause into an IF statement. For example, if the CH register contains a character value, you can select from a menu of items using code like the following:

```
if( ch = 'a' ) then

    stdout.put( "You selected the 'a' menu item", nl );

elseif( ch = 'b' ) then

    stdout.put( "You selected the 'b' menu item", nl );

elseif( ch = 'c' ) then

    stdout.put( "You selected the 'c' menu item", nl );

else

    stdout.put( "Error: illegal menu item selection", nl );

endif;
```

Although this simple example doesn't demonstrate it, HLA does not require an ELSE clause at the end of a sequence of ELSEIF clauses. However, when making multi-way decisions, it's always a good idea to provide an ELSE clause just in case an error arises. Even if you think it's impossible for the ELSE clause to execute, just keep in mind that future modifications to the code could possibly void this assertion, so it's a good idea to have error reporting statements built into your code.

2.9.3 The WHILE..ENDWHILE Statement

The WHILE statement uses the following basic syntax:

The expression in the WHILE
statement has the same
restrictions as the IF statement.

```
while( expression ) do

        sequence
        of one or              Loop Body
        more statements

endwhile;
```
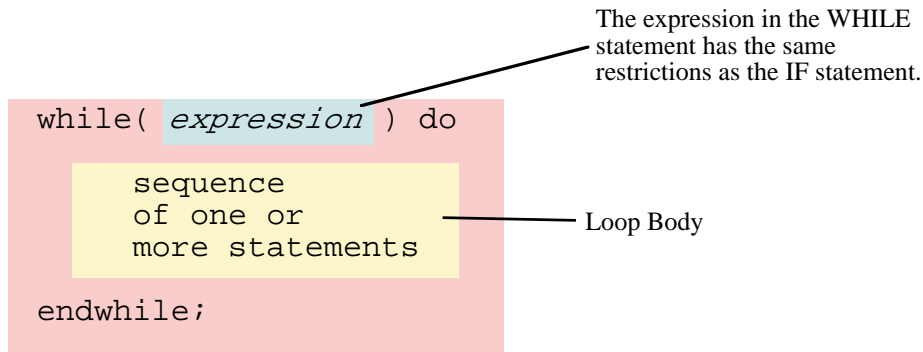
Figure 2.8   HLA While Statement Syntax

This statement evaluates the boolean expression. If it is false, control immediately transfers to the first statement following the ENDWHILE clause. If the value of the expression is true, then control falls through to the body of the loop. After the loop body executes, control transfers back to the top of the loop where the WHILE statement retests the loop control expression. This process repeats until the expression evaluates false.

Note that the WHILE loop, like its high level language siblings, tests for loop termination at the top of the loop. Therefore, it is quite possible that the statements in the body of the loop will not execute (if the expression is false when the code first executes the WHILE statement). Also note that the body of the WHILE loop must, at some point, modify the value of the boolean expression or an infinite loop will result.

```
mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );
    add( 1, i );

endwhile;
```

2.9.4 The FOR..ENDFOR Statement

The HLA FOR loop takes the following general form:

```
for( Initial_Stmt; Termination_Expression; Post_Body_Statement ) do

    << Loop Body >>

endfor;
```

This is equivalent to the following WHILE statement:

```
Initial_Stmt;
while( Termination_expression ) do

    << loop_body >>

    Post_Body_Statement;

endwhile;
```

Initial_Stmt can be any single HLA/80x86 instruction. Generally this statement initializes a register or memory location (the loop counter) with zero or some other initial value. *Termination_expression* is an

HLA boolean expression (same format that WHILE allows). This expression determines whether the loop body will execute. The *Post_Body_Statement* executes at the bottom of the loop (as shown in the WHILE example above). This is a single HLA statement. Usually it is an instruction like ADD that modifies the value of the loop control variable.

The following gives a complete example:

```
for( mov( 0, i ); i < 10; add(1, i )) do

    stdout.put( "i=", i, nl );

endfor;


// The above, rewritten as a while loop, becomes:

mov( 0, i );
while( i < 10 ) do

    stdout.put( "i=", i, nl );

    add( 1, i );

endwhile;
```

2.9.5 The REPEAT..UNTIL Statement

The HLA repeat..until statement uses the following syntax:


```
repeat

    sequence
    of one or              ———— Loop Body
    more statements

until( expression );
```

The expression in the UNTIL clause has the same restrictions as the IF statement.
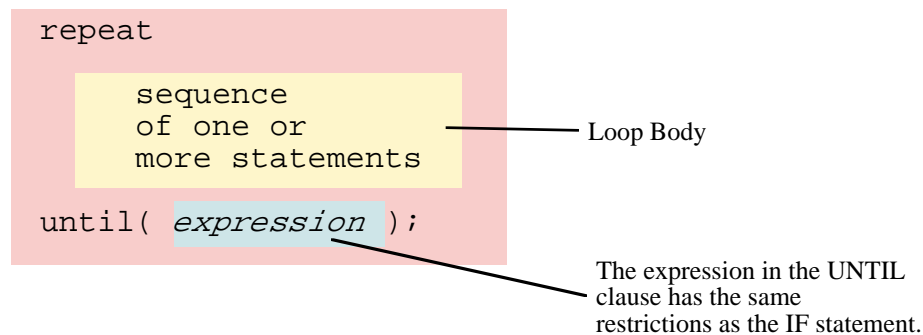
Figure 2.9   HLA Repeat..Until Statement Syntax

The HLA REPEAT..UNTIL statement tests for loop termination at the bottom of the loop. Therefore, the statements in the loop body always execute at least once. Upon encountering the UNTIL clause, the program will evaluate the expression and repeat the loop if the expression is false (that is, it repeats while false). If the expression evaluates true, the control transfers to the first statement following the UNTIL clause.

The following simple example demonstrates one use for the REPEAT..UNTIL statement:

```
mov( 10, ecx );
repeat

    stdout.put( "ecx = ", ecx, nl );
    sub( 1, ecx );

until( ecx = 0 );
```

If the loop body will always execute at least once, then it is more efficient to use a REPEAT..UNTIL loop rather than a WHILE loop.

2.9.6 The BREAK and BREAKIF Statements

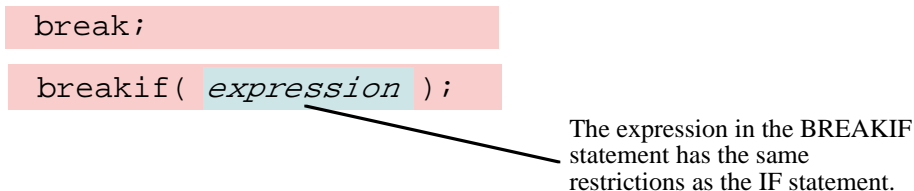The BREAK and BREAKIF statements provide the ability to prematurely exit from a loop. They use the following syntax:

```
break;
```

```
breakif( expression );
```
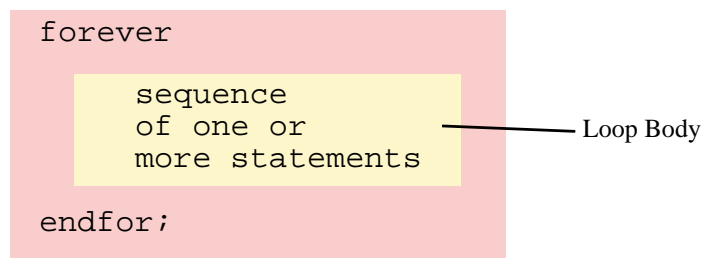
The expression in the BREAKIF statement has the same restrictions as the IF statement.

Figure 2.10   HLA Break and Breakif Syntax

The BREAK statement exits the loop that immediately contains the break; The BREAKIF statement evaluates the boolean expression and terminates the containing loop if the expression evaluates true.

2.9.7 The FOREVER..ENDFOR Statement

The FOREVER statement uses the following syntax:

```
forever

    sequence
    of one or
    more statements

endfor;
```

Loop Body

Figure 2.11   HLA Forever Loop Syntax

This statement creates an infinite loop. You may also use the BREAK and BREAKIF statements along with FOREVER..ENDFOR to create a loop that tests for loop termination in the middle of the loop. Indeed, this is probably the most common use of this loop as the following example demonstrates:

```
forever

    stdout.put( "Enter an integer less than 10: ");
    stdin.get( i );
    breakif( i < 10 );
    stdout.put( "The value needs to be less than 10!", nl );

endfor;
```

2.9.8 The TRY..EXCEPTION..ENDTRY Statement

The HLA TRY..EXCEPTION..ENDTRY statement provides very powerful *exception handling* capabilities. The syntax for this statement is the following:

```
try

        sequence
        of one or                          ———— Statements to test
        more statements

exception( exceptionID )
                                           At least one
        sequence                           exception handling
        of one or                          block.
        more statements


exception( exceptionID )                   Zero or more (optional)
                                           exception handling
        sequence                           blocks.
        of one or
        more statements


endtry;
```
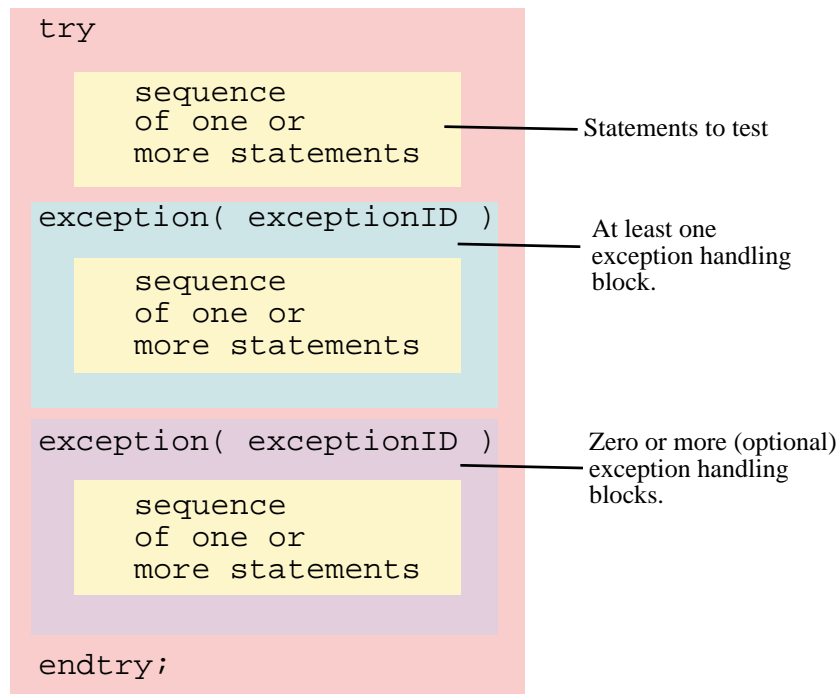
Figure 2.12   HLA Try..Except..Endtry Statement Syntax

The TRY..ENDTRY statement protects a block of statements during execution. If these statements, between the TRY clause and the first EXCEPTION clause, execute without incident, control transfers to the first statement after the ENDTRY immediately after executing the last statement in the protected block. If an error (exception) occurs, then the program interrupts control at the point of the exception (that is, the program *raises* an exception). Each exception has an unsigned integer constant associated with it, known as the exception ID. The "excepts.hhf" header file in the HLA Standard Library predefines several exception IDs, although you may create new ones for your own purposes. When an exception occurs, the system compares the exception ID against the values appearing in each of the one or more EXCEPTION clauses following the protected code. If the current exception ID matches one of the EXCEPTION values, control continues with the block of statements immediately following that EXCEPTION. After the exception handling code completes execution, control transfers to the first statement following the ENDTRY.

If an exception occurs and there is no active TRY..ENDTRY statement, or the active TRY..ENDTRY statements do not handle the specific exception, the program will abort with an error message.

The following sample program demonstrates how to use the TRY..ENDTRY statement to protect the program from bad user input:

```
repeat

    mov( false, GoodInteger );    // Note: GoodInteger must be a boolean var.
    try

        stdout.put( "Enter an integer: " );
        stdin.get( i );
        mov( true, GoodInteger );

    exception( ex.ConversionError );

        stdout.put( "Illegal numeric value, please re-enter", nl );

    exception( ex.ValueOutOfRange );

        stdout.put( "Value is out of range, please re-enter", nl );

    endtry;

until( GoodInteger );
```

The REPEAT..UNTIL loop repeats this code as long as there is an error during input. Should an exception occur, control transfers to the EXCEPTION clauses to see if a conversion error (e.g., illegal characters in the number) or a numeric overflow occurs. If either of these exceptions occur, then they print the appropriate message and control falls out of the TRY..ENDTRY statement and the REPEAT..UNTIL loop repeats since *GoodInteger* was never set to true. If a different exception occurs (one that is not handled in this code), then the program aborts with the specified error message[9].

Please see the "excepts.hhf" header file that accompanies the HLA release for a complete list of all the exception ID codes. The HLA documentation will describe the purpose of each of these exception codes.

2.10   Introduction to the HLA Standard Library

There are two reasons HLA is much easier to learn and use than standard assembly language. The first reason is HLA's high level syntax for declarations and control structures. This HLA feature leverages your high level language knowledge, reducing the need to learn arcane syntax, allowing you to learn assembly language more efficiently. The other half of the equation is the HLA Standard Library. The HLA Standard Library provides lot of commonly needed, easy to use, assembly language routines that you can call without having to write this code yourself (or even learn how to write yourself). This eliminates one of the larger stumbling blocks many people have when learning assembly language: the need for sophisticated I/O and support code in order to write basic statements. Prior to the advent of a standardized assembly language library, it often took weeks of study before a new assembly language programmer could do as much as print a string to the display. With the HLA Standard Library, this roadblock is removed and you can concentrate on learning assembly language concepts rather than learning low-level I/O details that are specific to a given operating system.

A wide variety of library routines is only part of HLA's support. After all, assembly language libraries have been around for quite some time[10]. HLA's Standard Library continues the HLA tradition by providing a high level language interface to these routines. Indeed, the HLA language itself was originally designed specifically to allow the creation of a high-level accessible set of library routines[11]. This high level interface,

---

9. An experienced programmer may wonder why this code uses a boolean variable rather than a BREAKIF statement to exit the REPEAT..UNTIL loop. There are some technical reasons for this that you will learn about later in this text.
10. E.g., the UCR Standard Library for 80x86 Assembly Language Programmers.
11. HLA was created because MASM was insufficient to support the creation of the UCR StdLib v2.0.

combined with the high level nature of many of the routines in the library, packs a surprising amount of power in an easy to use package.

The HLA Standard Library consists of several modules organized by category. The following table lists many of the modules that are available[12]:

Table 3: HLA Standard Library Modules

Name	Description
args	Command line parameter parsing support routines.
conv	Various conversions between strings and other values.
cset	Character set functions.
DateTime	Calendar, date, and time functions.
excepts	Exception handling routines.
fileio	File input and output routines
hla	Special HLA constants and other values.
Linux	Linux system calls (HLA Linux version only).
math	Transcendental and other mathematical functions.
memory	Memory allocation, deallocation, and support code.
misctypes	Miscellaneous data types.
patterns	The HLA pattern matching library.
rand	Pseudo-random number generators and support code.
stdin	User input routines
stdout	Provides user output and several other support routines.
stdlib	A special include file that links in all HLA standard library modules.
strings	HLA's powerful string library.
tables	Table (associative array) support routines.
win32	Constants used in Windows calls (HLA Win32 version, only)
x86	Constants and other items specific to the 80x86 CPU.

Later sections of this text will explain many of these modules in greater detail. This section will concentrate on the most important routines (at least to beginning HLA programmers), the *stdio* library.

_____

12. Since the HLA Standard Library is expanding, this list is probably out of date. Please see the HLA documentation for a current list of Standard Library modules.

2.10.1 Predefined Constants in the STDIO Module

Perhaps the first place to start is with a description of some common constants that the STDIO module defines for you. One constant you've seen already in code appearing in this chapter. Consider the following (typical) example:

```
stdout.put( "Hello World", nl );
```

The *nl* appearing at the end of this statement stands for *newline*. The *nl* identifier is not a special HLA reserved word, nor is it specific to the *stdout.put* statement. Instead, it's simply a predefined constant that corresponds to the string containing the standard end of line sequence (this is a carriage return/line feed pair under Windows or just a line feed under Linux).

In addition to the *nl* constant, the HLA standard I/O library module defines several other useful character constants. They are

- • stdio.bell   The ASCII bell character. Beeps the speaker when printed.
- • stdio.bs   The ASCII backspace character.
- • stdio.tab   The ASCII tab character.
- • stdio.eoln   A linefeed character (even under Windows).
- • stdio.lf   The ASCII linefeed character.
- • stdio.cr   The ASCII carriage return character.

Except for *nl*, these characters appear in the *stdio* namespace (and, therefore, require the "stdio." prefix). The placement of these ASCII constants within the *stdio* namespace is to help avoid naming conflicts with your own variables. The *nl* name does not appear within a namespace because you will use it very often and typing *stdio.nl* would get tiresome very quickly.

2.10.2 Standard In and Standard Out

Many of the HLA I/O routines have a *stdin* or *stdout* prefix. Technically, this means that the standard library defines these names in a *namespace*[13]. In practice, this prefix suggests where the input is coming from (the *standard input* device) or going to (the *standard output* device). By default, the standard input device is the system keyboard. Likewise, the default standard output device is the console display. So, in general, statements that have *stdin* or *stdout* prefixes will read and write data on the console device.

When you run a program from the command line window (or shell), you have the option of *redirecting* the standard input and/or standard output devices. A command line parameter of the form ">outfile" redirects the standard output device to the specified file (outfile). A command line parameter of the form "<infile" redirects the standard input so that its data comes from the specified input file (infile). The following examples demonstrate how to use these parameters when running a program named "testpgm" in the command window[14]:

```
testpgm <input.data
testpgm >output.txt
testpgm <in.txt >output.txt
```

---

13. Namespaces will be the subject of a later chapter.
14. Note for Linux users: depending on how your system is set up, you may need to type "./" in front of the program's name to actually execute the program, e.g., "./testpgm <input.data".

2.10.3 The stdout.newln Routine

The *stdout.newln* procedure prints a newline sequence to the standard output device. This is functionally equivalent to saying "stdout.put(nl);" Of course, the call to *stdout.newln* is sometimes a little more convenient. Example of call:

```
stdout.newln();
```

2.10.4 The stdout.puti*X* Routines

The *stdout.puti8*, *stdout.puti16*, and *stdout.puti32* library routines print a single parameter (one byte, two bytes, or four bytes, respectively) as a signed integer value. The parameter may be a constant, a register, or a memory variable, as long as the size of the actual parameter is the same as the size of the formal parameter.

These routines print the value of their specified parameter to the standard output device. These routines will print the value using the minimum number of print positions possible. If the number is negative, these routines will print a leading minus sign. Here are some examples of calls to these routines:

```
stdout.puti8( 123 );
stdout.puti16( DX );
stdout.puti32( i32Var );
```

2.10.5 The stdout.puti*X*Size Routines

The *stdout.puti8Size*, *stdout.puti16Size*, and *stdout.puti32Size* routines output signed integer values to the standard output, just like the *stdout.putiX* routines. These routines, however, provide more control over the output; they let you specify the (minimum) number of print positions the value will require on output. These routines also let you specify a padding character should the print field be larger than the minimum needed to display the value. These routines require the following parameters:

```
stdout.puti8Size( Value8, width, padchar );
stdout.puti16Size( Value16,width, padchar );
stdout.puti32Size( Value32, width, padchar );
```

The *ValueX* parameter can be a constant, a register, or a memory location of the specified size. The *width* parameter can be any signed integer constant that is between -256 and +256; this parameter may be a constant, register (32-bit), or memory location (32-bit). The *padchar* parameter should be a single character value.

Like the *stdout.putiX* routines, these routines print the specified value as a signed integer constant to the standard output device. These routines, however, let you specify the *field width* for the value. The field width is the minimum number of print positions these routines will use when printing the value. The *width* parameter specifies the minimum field width. If the number would require more print positions (e.g., if you attempt to print "1234" with a field width of two), then these routines will print however many characters are necessary to properly display the value. On the other hand, if the *width* parameter is greater than the number of character positions required to display the value, then these routines will print some extra padding characters to ensure that the output has at least *width* character positions. If the *width* value is negative, the number is left justified in the print field; if the *width* value is positive, the number is right justified in the print field.

If the absolute value of the *width* parameter is greater than the minimum number of print positions, then these *stdout.putiXSize* routines will print a padding character before or after the number. The *padchar* parameter specifies which character these routines will print. Most of the time you would specify a space as the pad character; for special cases, you might specify some other character. Remember, the *padchar* parameter is a character value; in HLA character constants are surrounded by apostrophes, not quotation marks. You may also specify an eight-bit register as this parameter.

Here is a short HLA program that demonstrates the use of the puti32Size routine to display a list of values in tabular form:

```
program NumsInColumns;

#include( "stdlib.hhf" );

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.puti32Size( i32, 5, ' ' );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.newln();

end NumsInColumns;
```

Program 2.4   Columnar Output Demonstration Using stdio.Puti32Size

2.10.6 The stdout.put Routine

The *stdout.put* routine[15] is one of the most flexible output routines in the standard output library module. It combines most of the other output routines into a single, easy to use, procedure.

The generic form for the *stdout.put* routine is the following:

stdout.put(*list_of_values_to_output*);

The *stdout.put* parameter list consists of one or more constants, registers, or memory variables, each separated by a comma. This routine displays the value associated with each parameter appearing in the list. Since we've already been using this routine throughout this chapter, you've already seen lots of examples of this routine's basic form. It is worth pointing out that this routine has several additional features not apparent in the examples appearing in this chapter. In particular, each parameter can take one of the following two forms:

value

---

15. *Stdout.put* is actually a macro, not a procedure. The distinction between the two is beyond the scope of this chapter. However, this text will describe their differences a little later.

value:width

The *value* may be any legal constant, register, or memory variable object. In this chapter, you've seen string constants and memory variables appearing in the *stdout.put* parameter list. These parameters correspond to the first form above. The second parameter form above lets you specify a minimum field width, similar to the *stdout.putiXSize* routines[16]. The following sample program produces the same output as the previous program; however, it uses *stdout.put* rather than *stdout.puti32Size*:

```
program NumsInColumns2;

#include( "stdlib.hhf" );

var
    i32:    int32;
    ColCnt: int8;

begin NumsInColumns2;

    mov( 96, i32 );
    mov( 0, ColCnt );
    while( i32 > 0 ) do

        if( ColCnt = 8 ) then

            stdout.newln();
            mov( 0, ColCnt );

        endif;
        stdout.put( i32:5 );
        sub( 1, i32 );
        add( 1, ColCnt );

    endwhile;
    stdout.put( nl );

end NumsInColumns2;
```

Program 2.5   Demonstration of stdout.put Field Width Specification

The *stdout.put* routine is capable of much more than the few attributes this section describes. This text will introduce those additional capabilities as appropriate.

2.10.7 The stdin.getc Routine.

The *stdin.getc* routine reads the next available character from the standard input device's input buffer[17]. It returns this character in the CPU's AL register. The following example program demonstrates a simple use of this routine:

---

16. Note that you cannot specify a padding character when using the *stdout.put* routine; the padding character defaults to the space character. If you need to use a different padding character, call the *stdout.putiXSize* routines.
17. "Buffer" is just a fancy term for an array.

© 2001, By Randall Hyde

```
program charInput;

#include( "stdlib.hhf" );

var
    counter: int32;

begin charInput;

    // The following repeats as long as the user
    // confirms the repetition.

    repeat

        // Print out 14 values.

        mov( 14, counter );
        while( counter > 0 ) do

            stdout.put( counter:3 );
            sub( 1, counter );

        endwhile;

        // Wait until the user enters 'y' or 'n'.

        stdout.put( nl, nl, "Do you wish to see it again? (y/n):" );
        forever

            stdin.readLn();
            stdin.getc();
            breakif( al = 'n' );
            breakif( al = 'y' );
            stdout.put( "Error, please enter only 'y' or 'n': " );

        endfor;
        stdout.newln();

    until( al = 'n' );

end charInput;
```

Program 2.6   Demonstration of the stdin.getc() Routine

This program uses the *stdin.ReadLn* routine to force a new line of input from the user. A description of *stdin.ReadLn* appears just a little later in this chapter.

2.10.8 The stdin.geti*X* Routines

The *stdin.geti8*, *stdin.geti16*, and *stdin.geti32* routines read eight, 16, and 32-bit signed integer values from the standard input device. These routines return their values in the AL, AX, or EAX register, respectively. They provide the standard mechanism for reading signed integer values from the user in HLA.

Like the *stdin.getc* routine, these routines read a sequence of characters from the standard input buffer. They begin by skipping over any white space characters (spaces, tabs, etc.) and then convert the following stream of decimal digits (with an optional, leading, minus sign) into the corresponding integer. These rou-

tines raise an exception (that you can trap with the TRY..ENDTRY statement) if the input sequence is not a valid integer string or if the user input is too large to fit in the specified integer size. Note that values read by *stdin.geti8* must be in the range -128..+127; values read by *stdin.geti16* must be in the range -32,768..+32,767; and values read by *stdin.geti32* must be in the range -2,147,483,648..+2,147,483,647.

The following sample program demonstrates the use of these routines:

```
program intInput;

#include( "stdlib.hhf" );

var
    i8:     int8;
    i16:    int16;
    i32:    int32;

begin intInput;

    // Read integers of varying sizes from the user:

    stdout.put( "Enter a small integer between –128 and +127: " );
    stdin.geti8();
    mov( al, i8 );

    stdout.put( "Enter a small integer between –32768 and +32767: " );
    stdin.geti16();
    mov( ax, i16 );

    stdout.put( "Enter an integer between +/– 2 billion: " );
    stdin.geti32();
    mov( eax, i32 );

    // Display the input values.

    stdout.put
    (
        nl,
        "Here are the numbers you entered:", nl, nl,
        "Eight-bit integer: ", i8:12, nl,
        "16-bit integer:    ", i16:12, nl,
        "32-bit integer:    ", i32:12, nl
    );



end intInput;
```

Program 2.7   stdin.getiX Example Code

You should compile and run this program and test what happens when you enter a value that is out of range or enter an illegal string of characters.

2.10.9 The stdin.readLn and stdin.flushInput Routines

Whenever you call an input routine like *stdin.getc* or *stdin.geti32*, the program does not necessarily read the value from the user at that moment. Instead, the HLA Standard Library buffers the input by reading a whole line of text from the user. Calls to input routines will fetch data from this input buffer until the buffer is empty. While this buffering scheme is efficient and convenient, sometimes it can be confusing. Consider the following code sequence:

```
stdout.put( "Enter a small integer between -128 and +127: " );
stdin.geti8();
mov( al, i8 );

stdout.put( "Enter a small integer between -32768 and +32767: " );
stdin.geti16();
mov( ax, i16 );
```

Intuitively, you would expect the program to print the first prompt message, wait for user input, print the second prompt message, and wait for the second user input. However, this isn't exactly what happens. For example if you run this code (from the sample program in the previous section) and enter the text "123 456" in response to the first prompt, the program will not stop for additional user input at the second prompt. Instead, it will read the second integer (456) from the input buffer read during the execution of the *stdin.geti8* call.

In general, the *stdin* routines only read text from the user when the input buffer is empty. As long as the input buffer contains additional characters, the input routines will attempt to read their data from the buffer. You may take advantage of this behavior by writing code sequences such as the following:

```
stdout.put( "Enter two integer values: " );
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

This sequence allows the user to enter both values on the same line (separated by one or more white space characters) thus preserving space on the screen. So the input buffer behavior is desirable every now and then.

Unfortunately, the buffered behavior of the input routines is definitely counter-intuitive at other times. Fortunately, the HLA Standard Library provides two routines, *stdin.readLn* and *stdin.flushInput*, that let you control the standard input buffer. The *stdin.readLn* routine discards everything that is in the input buffer and immediately requires the user to enter a new line of text. The *stdin.flushInput* routine simply discards everything that is in the buffer. The next time an input routine executes, the system will require a new line of input from the user. You would typically call *stdin.readLn* immediately before some standard input routine; you would normally call *stdin.flushInput* immediately after a call to a standard input routine.

Note: If you are calling *stdin.readLn* and you find that you are having to input your data twice, this is a good indication that you should be calling *stdin.flushInput* rather than *stdin.readLn*. In general, you should always be able to call *stdin.flushInput* to flush the input buffer and read a new line of data on the next input call. The *stdin.readLn* routine is rarely necessary, so you should use *stdin.flushInput* unless you really need to immediately force the input of a new line of text.

2.10.10 The stdin.get Macro

The *stdin.get* macro combines many of the standard input routines into a single call, in much the same way that *stdout.put* combines all of the output routines into a single call. Actually, *stdin.get* is much easier to use than *stdout.put* since the only parameters to this routine are a list of variable names.

Let's rewrite the example given in the previous section:

```
stdout.put( "Enter two integer values: " );
```

```
stdin.geti32();
mov( eax, intval );
stdin.geti32();
mov( eax, AnotherIntVal );
```

Using the *stdin.get* macro, we could rewrite this code as:

```
stdout.put( "Enter two integer values: " );
stdin.get( intval, AnotherIntVal );
```

As you can see, the *stdin.get* routine is a little more convenient to use.

Note that *stdin.get* stores the input values directly into the memory variables you specify in the parameter list; it does not return the values in a register unless you actually specify a register as a parameter. The *stdin.get* parameters must all be variables or registers[18].

2.11 Putting It All Together

This chapter has covered a lot of ground! While you've still got a lot to learn about assembly language programming, this chapter, combined with your knowledge of high level languages, provides just enough information to let you start writing real assembly language programs.

In this chapter, you've seen the basic format for an HLA program. You've seen how to declare integer, character, and boolean variables. You have taken a look at the internal organization of the Intel 80x86 CPU family and learned about the MOV, ADD, and SUB instructions. You've looked at the basic HLA high level language control structures (IF, WHILE, REPEAT, FOR, BREAK, BREAKIF, FOREVER, and TRY) as well as what constitutes a legal boolean expression in these statements. Finally, this chapter has introduced several commonly-used routines in the HLA Standard Library.

You might think that knowing only three machine instructions is hardly sufficient to write meaningful programs. However, those three instructions (*mov, add,* and *sub*), combined with the HLA high level control structures and the HLA Standard Library routines are actually equivalent to knowing several dozen machine instructions. Certainly enough to write simple programs. Indeed, with only a few more arithmetic instructions plus the ability to write your own procedures, you'll be able to write almost any program. Of course, your journey into the world of assembly language has only just begun; you'll learn some more instructions, and how to use them, starting in the next chapter.

2.12 Sample Programs

This section contains several little HLA programs that demonstrate some of HLA's features appearing in this chapter. These short examples also demonstrate that it is possible to write meaningful (if simple) programs in HLA using nothing more than the information appearing in this chapter. You may find all of the sample programs appearing in this section in the "ch02" subdirectory of the "volume1" directory in the software that accompanies this text.

2.12.1 Powers of Two Table Generation

The following sample program generates a table listing all the powers of two between 2**0 and 2**30.

---

18. Note that register input is always in hexadecimal or base 16. The next chapter will discuss hexadecimal numbers.

```
        // PowersOfTwo-
        //
        //  This program generates a nicely-formatted
        //  "Powers of Two" table.  It computes the
        //  various powers of two by successively
        //  doubling the value in the pwrOf2 variable.

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

        static

            pwrOf2:     int32;
            LoopCntr:   int32;

        begin PowersOfTwo;

            // Print a start up banner.

            stdout.put( "Powers of two: ", nl, nl );

            // Initialize "pwrOf2" with 2**0 (two raised to the zero power).

            mov( 1, pwrOf2 );

            // Because of the limitations of 32-bit signed integers,
            // we can only display 2**0..2**30.

            mov( 0, LoopCntr );
            while( LoopCntr < 31 ) do

                stdout.put( "2**(", LoopCntr:2, ") = ", pwrOf2:10, nl );

                // Double the value in pwrOf2 to compute the
                // next power of two.

                mov( pwrOf2, eax );
                add( eax, eax );
                mov( eax, pwrOf2 );

                // Move on to the next loop iteration.

                inc( LoopCntr );

            endwhile;
            stdout.newln();

        end PowersOfTwo;
```

Program 2.8   Powers of Two Table Generator Program

2.12.2 Checkerboard Program

This short little program demonstrates how to generate a checkerboard pattern with HLA.

```
    // CheckerBoard-
    //
    // This program demonstrates how to draw a
    // checkerboard using a set of nested while
    // loops.

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

    static

        xCoord:     int8;   // Counts off eight squares in each row.
        yCoord:     int8;   // Counts off four pairs of squares in each column.
        ColCntr:    int8;   // Counts off four rows in each square.

    begin CheckerBoard;

        mov( 0, yCoord );
        while( yCoord < 4 ) do

            // Display a row that begins with black.

            mov( 4, ColCntr );
            repeat

                // Each square is a 4x4 group of
                // spaces (white) or asterisks (black).
                // Print out one row of asterisks/spaces
                // for the current row of squares:

                mov( 0, xCoord );
                while( xCoord < 4 ) do

                    stdout.put( "****    " );
                    add( 1, xCoord );

                endwhile;
                stdout.newln();
                sub( 1, ColCntr );

            until( ColCntr = 0 );

            // Display a row that begins with white.

            mov( 4, ColCntr );
            repeat

                // Print out a single row of
                // spaces/asterisks for this
                // row of squares:

                mov( 0, xCoord );
                while( xCoord < 4 ) do

                    stdout.put( "    ****" );
                    add( 1, xCoord );

                endwhile;
                stdout.newln();
                sub( 1, ColCntr );
```

```
            until( ColCntr = 0 );

        add( 1, yCoord );

    endwhile;

end CheckerBoard;
```

Program 2.9   Checkerboard Generation Program

2.12.3 Fibonacci Number Generation

The Fibonacci sequence is very important to certain algorithms in Computer Science and other fields. The following sample program generates a sequence of Fibonacci numbers for n=1..40.

```
// This program generates the fibonocci
// sequence for n=1..40.
//
// The fibonocci sequence is defined recursively
// for positive integers as follows:
//
//  fib(1) = 1;
//  fib(2) = 1;
//  fib( n ) = fib( n-1 ) + fib( n-2 ).
//
//  This program provides an iterative solution.


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

static

    FibCntr:    int32;
    CurFib:     int32;
    LastFib:    int32;
    TwoFibsAgo: int32;

begin fib;

    // Some simple initialization:

    mov( 1, LastFib );
    mov( 1, TwoFibsAgo );

    // Print fib(1) and fib(2) as a special case:

    stdout.put
    (
        "fib( 1 ) =          1", nl
        "fib( 2 ) =          1", nl
```

```
                    );

            // Use a loop to compute the remaining fib values:

            mov( 3, FibCntr );
            while( FibCntr <= 40 ) do

                // Get the last two computed fibonocci values
                // and add them together:

                mov( LastFib, ebx );
                mov( TwoFibsAgo, eax );
                add( ebx, eax );

                // Save the result and print it:

                mov( eax, CurFib );
                stdout.put( "fib(",FibCntr:2, ") =", CurFib:10, nl );

                // Recycle current LastFib (in ebx) as TwoFibsAgo,
                // and recycle CurFib as LastFib.

                mov( eax, LastFib );
                mov( ebx, TwoFibsAgo );

                // Bump up our loop counter:

                add( 1, FibCntr );

            endwhile;

        end fib;
```

Program 2.10   Fibonacci Sequence Generator

Data Representation   Chapter Three

A major stumbling block many beginners encounter when attempting to learn assembly language is the common use of the binary and hexadecimal numbering systems. Many programmers think that hexadecimal (or hex[1]) numbers represent absolute proof that God never intended anyone to work in assembly language. While it is true that hexadecimal numbers are a little different from what you may be used to, their advantages outweigh their disadvantages by a large margin. Nevertheless, understanding these numbering systems is important because their use simplifies other complex topics including boolean algebra and logic design, signed numeric representation, character codes, and packed data.

3.1   Chapter Overview

This chapter discusses several important concepts including the binary and hexadecimal numbering systems, binary data organization (bits, nibbles, bytes, words, and double words), signed and unsigned numbering systems, arithmetic, logical, shift, and rotate operations on binary values, bit fields and packed data. This is basic material and the remainder of this text depends upon your understanding of these concepts. If you are already familiar with these terms from other courses or study, you should at least skim this material before proceeding to the next chapter. If you are unfamiliar with this material, or only vaguely familiar with it, you should study it carefully before proceeding. *All of the material in this chapter is important!* Do not skip over any material. In addition to the basic material, this chapter also introduces some new HLA statements and HLA Standard Library routines.

3.2   Numbering Systems

Most modern computer systems do not represent numeric values using the decimal system. Instead, they typically use a binary or two's complement numbering system. To understand the limitations of computer arithmetic, you must understand how computers represent numbers.

3.2.1   A Review of the Decimal System

You've been using the decimal (base 10) numbering system for so long that you probably take it for granted. When you see a number like "123", you don't think about the value 123; rather, you generate a mental image of how many items this value represents. In reality, however, the number 123 represents:

```
1*10² + 2 * 10¹ + 3*10⁰
or
100+20+3
```
$$1*10^2 + 2 * 10^1 + 3*10^0$$
or
$$100+20+3$$

In the positional numbering system, each digit appearing to the left of the decimal point represents a value between zero and nine times an increasing power of ten. Digits appearing to the right of the decimal point represent a value between zero and nine times an increasing negative power of ten. For example, the value 123.456 means:

$$1*10^2 + 2*10^1 + 3*10^0 + 4*10^{-1} + 5*10^{-2} + 6*10^{-3}$$
or
$$100 + 20 + 3 + 0.4 + 0.05 + 0.006$$

---

1. Hexadecimal is often abbreviated as *hex* even though, technically speaking, hex means base six, not base sixteen.

3.2.2   The Binary Numbering System

Most modern computer systems (including PCs) operate using binary logic. The computer represents values using two voltage levels (usually 0v and +2.4..5v). With two such levels we can represent exactly two different values. These could be any two different values, but they typically represent the values zero and one. These two values, coincidentally, correspond to the two digits used by the binary numbering system. Since there is a correspondence between the logic levels used by the 80x86 and the two digits used in the binary numbering system, it should come as no surprise that the PC employs the binary numbering system.

The binary numbering system works just like the decimal numbering system, with two exceptions: binary only allows the digits 0 and 1 (rather than 0-9), and binary uses powers of two rather than powers of ten. Therefore, it is very easy to convert a binary number to decimal. For each "1" in the binary string, add in 2^n where "n" is the zero-based position of the binary digit. For example, the binary value 11001010_2 represents:

$$1*2^7 + 1*2^6 + 0*2^5 + 0*2^4 + 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0$$
$$=$$
$$128 + 64 + 8 + 2$$
$$=$$
$$202_{10}$$

To convert decimal to binary is slightly more difficult. You must find those powers of two which, when added together, produce the decimal result. One method is to work from a large power of two down to 2^0. Consider the decimal value 1359:

- $2^{10} = 1024$, $2^{11} = 2048$. So 1024 is the largest power of two less than 1359. Subtract 1024 from 1359 and begin the binary value on the left with a "1" digit. Binary = "1", Decimal result is 1359 - 1024 = 335.

- The next lower power of two ($2^9 = 512$) is greater than the result from above, so add a "0" to the end of the binary string. Binary = "10", Decimal result is still 335.

- The next lower power of two is 256 (2^8). Subtract this from 335 and add a "1" digit to the end of the binary number. Binary = "101", Decimal result is 79.

- 128 (2^7) is greater than 79, so tack a "0" to the end of the binary string. Binary = "1010", Decimal result remains 79.

- The next lower power of two ($2^6 = 64$) is less than 79, so subtract 64 and append a "1" to the end of the binary string. Binary = "10101", Decimal result is 15.

- 15 is less than the next power of two ($2^5 = 32$) so simply add a "0" to the end of the binary string. Binary = "101010", Decimal result is still 15.

- 16 (2^4) is greater than the remainder so far, so append a "0" to the end of the binary string. Binary = "1010100", Decimal result is 15.

- 2^3 (eight) is less than 15, so stick another "1" digit on the end of the binary string. Binary = "10101001", Decimal result is 7.

- 2^2 is less than seven, so subtract four from seven and append another one to the binary string. Binary = "101010011", decimal result is 3.

- 2^1 is less than three, so append a one to the end of the binary string and subtract two from the decimal value. Binary = "1010100111", Decimal result is now 1.

- Finally, the decimal result is one, which is 2^0, so add a final "1" to the end of the binary string. The final binary result is "10101001111"

If you actually have to convert a decimal number to binary by hand, the algorithm above probably isn't the easiest to master. A simpler solution is the "even/odd – divide by two" algorithm. This algorithm uses the following steps:

- If the number is even, emit a zero. If the number is odd, emit a one.
- Divide the number by two and throw away any fractional component or remainder.

- If the quotient is zero, the algorithm is complete.
- If the quotient is not zero and is odd, insert a one before the current string; if the number is even, prefix your binary string with zero.
- Go back to step two above and repeat.

Fortunately, you'll rarely need to convert decimal numbers directly to binary strings, so neither of these algorithms is particularly important in real life.

Binary numbers, although they have little importance in high level languages, appear everywhere in assembly language programs (even if you don't convert between decimal and binary). So you should be somewhat comfortable with them.

3.2.3 Binary Formats

In the purest sense, every binary number contains an infinite number of digits (or *bits* which is short for binary digits). For example, we can represent the number five by:

101   00000101   0000000000101   ... 000000000000101

Any number of leading zero bits may precede the binary number without changing its value.

We will adopt the convention of ignoring any leading zeros if present in a value. For example, 101_2 represents the number five but since the 80x86 works with groups of eight bits, we'll find it much easier to zero extend all binary numbers to some multiple of four or eight bits. Therefore, following this convention, we'd represent the number five as 0101_2 or 00000101_2.

In the United States, most people separate every three digits with a comma to make larger numbers easier to read. For example, 1,023,435,208 is much easier to read and comprehend than 1023435208. We'll adopt a similar convention in this text for binary numbers. We will separate each group of four binary bits with an underscore. For example, we will write the binary value 1010111110110010 as 1010_1111_1011_0010.

We often pack several values together into the same binary number. One form of the 80x86 MOV instruction uses the binary encoding 1011 0rrr dddd dddd to pack three items into 16 bits: a five-bit operation code (1_0110), a three-bit register field (rrr), and an eight-bit immediate value (dddd_dddd). For convenience, we'll assign a numeric value to each bit position. We'll number each bit as follows:

1)   The rightmost bit in a binary number is bit position zero.

2)   Each bit to the left is given the next successive bit number.

An eight-bit binary value uses bits zero through seven:

X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0

A 16-bit binary value uses bit positions zero through fifteen:

X_{15} X_{14} X_{13} X_{12} X_{11} X_{10} X_9 X_8 X_7 X_6 X_5 X_4 X_3 X_2 X_1 X_0

A 32-bit binary value uses bit positions zero through 31, etc.

Bit zero is usually referred to as the *low order* (L.O.) bit (some refer to this as the *least significant bit*). The left-most bit is typically called the *high order* (H.O.) bit (or the *most significant bit*). We'll refer to the intermediate bits by their respective bit numbers.

3.3   Data Organization

In pure mathematics a value may take an arbitrary number of bits. Computers, on the other hand, generally work with some specific number of bits. Common collections are single bits, groups of four bits (called *nibbles*), groups of eight bits (*bytes*), groups of 16 bits (*words*), groups of 32 bits (double words or *dwords*), groups of 64-bits (quad words or *qwords*), and more. The sizes are not arbitrary. There is a good reason for these particular values. This section will describe the bit groups commonly used on the Intel 80x86 chips.

3.3.1   Bits

The smallest "unit" of data on a binary computer is a single *bit*. Since a single bit is capable of representing only two different values (typically zero or one) you may get the impression that there are a very small number of items you can represent with a single bit. Not true! There are an infinite number of items you can represent with a single bit.

With a single bit, you can represent any two distinct items. Examples include zero or one, true or false, on or off, male or female, and right or wrong. However, you are *not* limited to representing binary data types (that is, those objects which have only two distinct values). You could use a single bit to represent the numbers 723 and 1,245. Or perhaps 6,254 and 5. You could also use a single bit to represent the colors red and blue. You could even represent two unrelated objects with a single bit. For example, you could represent the color red and the number 3,256 with a single bit. You can represent *any* two different values with a single bit. However, you can represent *only two* different values with a single bit.

To confuse things even more, different bits can represent different things. For example, one bit might be used to represent the values zero and one, while an adjacent bit might be used to represent the values true and false. How can you tell by looking at the bits? The answer, of course, is that you can't. But this illustrates the whole idea behind computer data structures: *data is what you define it to be*. If you use a bit to represent a boolean (true/false) value then that bit (by your definition) represents true or false. For the bit to have any real meaning, you must be consistent. That is, if you're using a bit to represent true or false at one point in your program, you shouldn't use the true/false value stored in that bit to represent red or blue later.

Since most items you'll be trying to model require more than two different values, single bit values aren't the most popular data type you'll use. However, since everything else consists of groups of bits, bits will play an important role in your programs. Of course, there are several data types that require two distinct values, so it would seem that bits are important by themselves. However, you will soon see that individual bits are difficult to manipulate, so we'll often use other data types to represent boolean values.

3.3.2   Nibbles

A *nibble* is a collection of four bits. It wouldn't be a particularly interesting data structure except for two items: BCD (*binary coded decimal*) numbers[2] and hexadecimal numbers. It takes four bits to represent a single BCD or hexadecimal digit. With a nibble, we can represent up to 16 distinct values since there are 16 unique combinations of a string of four bits:

```
0000
0001
0010
0011
0100
0101
0110
0111
1000
```

_____

2. Binary coded decimal is a numeric scheme used to represent decimal numbers using four bits for each decimal digit.

```
1001
1010
1011
1100
1101
1110
1111
```

In the case of hexadecimal numbers, the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are represented with four bits (see "The Hexadecimal Numbering System" on page 60). BCD uses ten different digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9) and requires four bits (since you can only represent eight different values with three bits). In fact, any sixteen distinct values can be represented with a nibble, but hexadecimal and BCD digits are the primary items we can represent with a single nibble.

3.3.3   Bytes

Without question, the most important data structure used by the 80x86 microprocessor is the byte. A byte consists of eight bits and is the smallest addressable datum (data item) on the 80x86 microprocessor. Main memory and I/O addresses on the 80x86 are all byte addresses. This means that the smallest item that can be individually accessed by an 80x86 program is an eight-bit value. To access anything smaller requires that you read the byte containing the data and mask out the unwanted bits. The bits in a byte are normally numbered from zero to seven as shown in Figure 3.1.
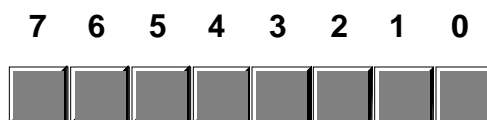
7   6   5   4   3   2   1   0

Figure 3.1   Bit Numbering

Bit 0 is the *low order bit* or *least significant bit*, bit 7 is the *high order bit* or *most significant bit* of the byte. We'll refer to all other bits by their number.

Note that a byte also contains exactly two nibbles (see Figure 3.2).

7   6   5   4   3   2   1   0
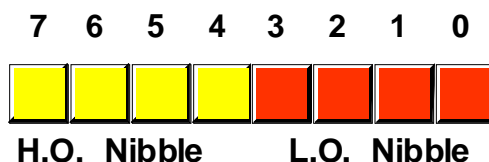
H.O. Nibble   L.O. Nibble

Figure 3.2   The Two Nibbles in a Byte

Bits 0..3 comprise the *low order nibble*, bits 4..7 form the *high order nibble*. Since a byte contains exactly two nibbles, byte values require two hexadecimal digits.

Since a byte contains eight bits, it can represent 2^8, or 256, different values. Generally, we'll use a byte to represent numeric values in the range 0..255, signed numbers in the range -128..+127 (see "Signed and Unsigned Numbers" on page 69), ASCII/IBM character codes, and other special data types requiring no more than 256 different values. Many data types have fewer than 256 items so eight bits is usually sufficient.

Since the 80x86 is a byte addressable machine (see the next volume), it turns out to be more efficient to manipulate a whole byte than an individual bit or nibble. For this reason, most programmers use a whole byte to represent data types that require no more than 256 items, even if fewer than eight bits would suffice. For example, we'll often represent the boolean values true and false by 00000001_2 and 00000000_2 (respectively).

Probably the most important use for a byte is holding a character code. Characters typed at the keyboard, displayed on the screen, and printed on the printer all have numeric values. To allow it to communicate with the rest of the world, PCs use a variant of the ASCII character set (see "The ASCII Character Encoding" on page 97). There are 128 defined codes in the ASCII character set. PCs typically use the remaining 128 possible values for extended character codes including European characters, graphic symbols, Greek letters, and math symbols.

Because bytes are the smallest unit of storage in the 80x86 memory space, bytes also happen to be the smallest variable you can create in an HLA program. As you saw in the last chapter, you can declare an eight-bit signed integer variable using the *int8* data type. Since *int8* objects are signed, you can represent values in the range -128..+127 using an *int8* variable (see "Signed and Unsigned Numbers" on page 69 for a discussion of signed number formats). You should only store signed values into *int8* variables; if you want to create an arbitrary byte variable, you should use the *byte* data type, as follows:

```
static
        byteVar: byte;
```

The *byte* data type is a partially untyped data type. The only type information associated with *byte* objects is their size (one byte). You may store any one-byte object (small signed integers, small unsigned integers, characters, etc.) into a byte variable. It is up to you to keep track of the type of object you've put into a byte variable.

3.3.4   Words

A word is a group of 16 bits. We'll number the bits in a word starting from zero on up to fifteen. The bit numbering appears in Figure 3.3.
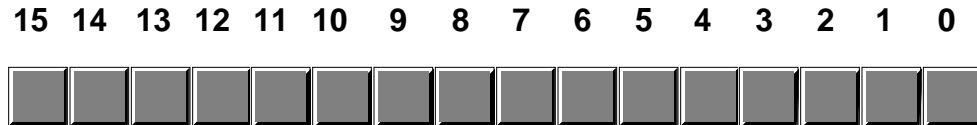

Figure 3.3   Bit Numbers in a Word

Like the byte, bit 0 is the low order bit. For words, bit 15 is the high order bit. When referencing the other bits in a word, use their bit position number.

Notice that a word contains exactly two bytes. Bits 0 through 7 form the low order byte, bits 8 through 15 form the high order byte (see Figure 3.4).
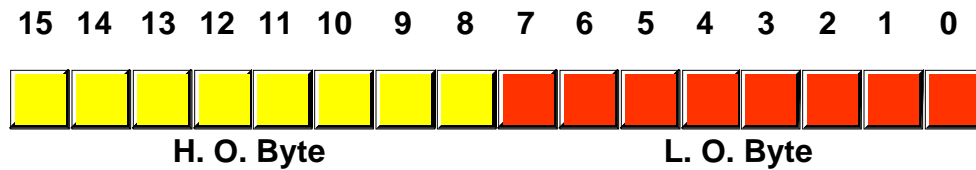
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

H. O. Byte   **L. O. Byte**

Figure 3.4   The Two Bytes in a Word

Naturally, a word may be further broken down into four nibbles as shown in Figure 3.5.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Nibble #3   Nibble #2   Nibble #1   Nibble #0

H. O. Nibble   L. O. Nibble
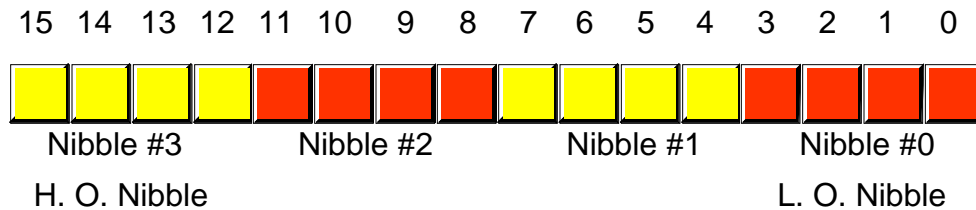
Figure 3.5   Nibbles in a Word

Nibble zero is the low order nibble in the word and nibble three is the high order nibble of the word. We'll simply refer to the other two nibbles as "nibble one" or "nibble two. "

With 16 bits, you can represent 2^{16} (65,536) different values. These could be the values in the range 0..65,535 or, as is usually the case, -32,768..+32,767, or any other data type with no more than 65,536 values. The three major uses for words are signed integer values, unsigned integer values, and UNICODE characters.

Words can represent integer values in the range 0..65,535 or -32,768..32,767. Unsigned numeric values are represented by the binary value corresponding to the bits in the word. Signed numeric values use the two's complement form for numeric values (see "Signed and Unsigned Numbers" on page 69).   As UNI-CODE characters, words can represent up to 65,536 different characters, allowing the use of non-Roman character sets in a computer program. UNICODE is an international standard, like ASCII, that allows computers to process non-Roman characters like Asian, Greek, and Russian characters.

Like bytes, you can also create word variables in an HLA program. Of course, in the last chapter you saw how to create sixteen-bit signed integer variables using the *int16* data type. To create an arbitary word variable, just use the *word* data type, as follows:

```
static
        w: word;
```

3.3.5   Double Words

A double word is exactly what its name implies, a pair of words. Therefore, a double word quantity is 32 bits long as shown in Figure 3.6.

31   23   15   7   0
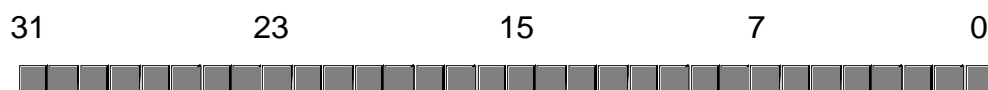

Figure 3.6   Bit Numbers in a Double Word

Naturally, this double word can be divided into a high order word and a low order word, four different bytes, or eight different nibbles (see Figure 3.7).

31   23   15   7   0

H.O. Word   L.O. Word

31   23   15   7   0

H.O. Byte   Byte # 2   Byte # 1   L.O. Byte

31   23   15   7   0

Nibble #7   #6   #5   #4   #3   #2   #1   #0
H. O.   L. O.
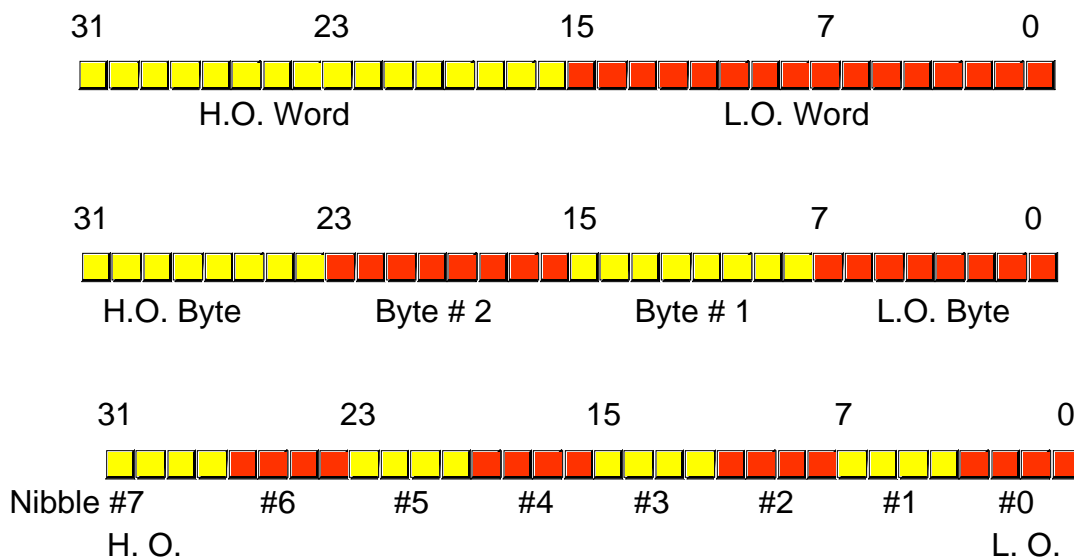
Figure 3.7   Nibbles, Bytes, and Words in a Double Word

Double words can represent all kinds of different things. A common item you will represent with a double word is a 32-bit integer value (which allows unsigned numbers in the range 0..4,294,967,295 or signed numbers in the range -2,147,483,648..2,147,483,647). 32-bit floating point values also fit into a double word. Another common use for dword objects is to store pointer variables.

In the previous chapter, you saw how to create 32-bit (dword) signed integer variables using the *int32* data type. You can also create an arbitrary double word variable using the *dword* data type as the following example demonstrates:

```
static
        d: dword;
```

3.4   The Hexadecimal Numbering System

A big problem with the binary system is verbosity. To represent the value 202_{10} requires eight binary digits. The decimal version requires only three decimal digits and, thus, represents numbers much more compactly than does the binary numbering system. This fact was not lost on the engineers who designed binary computer systems. When dealing with large values, binary numbers quickly become too unwieldy.

Unfortunately, the computer thinks in binary, so most of the time it is convenient to use the binary numbering system. Although we can convert between decimal and binary, the conversion is not a trivial task. The hexadecimal (base 16) numbering system solves these problems. Hexadecimal numbers offer the two features we're looking for: they're very compact, and it's simple to convert them to binary and vice versa. Because of this, most computer systems engineers use the hexadecimal numbering system. Since the radix (base) of a hexadecimal number is 16, each hexadecimal digit to the left of the hexadecimal point represents some value times a successive power of 16. For example, the number 1234_{16} is equal to:

```
1 * 16^3   +   2 * 16^2   +   3 * 16^1   +   4 * 16^0
```
or
```
4096 + 512 + 48 + 4 = 4660_10.
```

Each hexadecimal digit can represent one of sixteen values between 0 and 15_{10}. Since there are only ten decimal digits, we need to invent six additional digits to represent the values in the range 10_{10} through 15_{10}. Rather than create new symbols for these digits, we'll use the letters A through F. The following are all examples of valid hexadecimal numbers:

$$1234_{16} \quad DEAD_{16} \quad BEEF_{16} \quad 0AFB_{16} \quad FEED_{16} \quad DEAF_{16}$$

Since we'll often need to enter hexadecimal numbers into the computer system, we'll need a different mechanism for representing hexadecimal numbers. After all, on most computer systems you cannot enter a subscript to denote the radix of the associated value. We'll adopt the following conventions:

- All hexadecimal values begin with a "$" character, e.g., $123A4.
- All binary values begin with a percent sign ("%").
- Decimal numbers do not have a prefix character.
- If the radix is clear from the context, this text may drop the leading "$" or "%" character.

Examples of valid hexadecimal numbers:

$$\$1234 \quad \$DEAD \quad \$BEEF \quad \$AFB \quad \$FEED \quad \$DEAF$$

As you can see, hexadecimal numbers are compact and easy to read. In addition, you can easily convert between hexadecimal and binary. Consider the following table:

Table 4: Binary/Hex Conversion

Binary	Hexadecimal
%0000	$0
%0001	$1
%0010	$2
%0011	$3
%0100	$4
%0101	$5
%0110	$6
%0111	$7
%1000	$8
%1001	$9

Table 4: Binary/Hex Conversion

Binary	Hexadecimal
%1010	$A
%1011	$B
%1100	$C
%1101	$D
%1110	$E
%1111	$F

This table provides all the information you'll ever need to convert any hexadecimal number into a binary number or vice versa.

To convert a hexadecimal number into a binary number, simply substitute the corresponding four bits for each hexadecimal digit in the number. For example, to convert $ABCD into a binary value, simply convert each hexadecimal digit according to the table above:

0	A	B	C	D	Hexadecimal
0000	1010	1011	1100	1101	Binary

To convert a binary number into hexadecimal format is almost as easy. The first step is to pad the binary number with zeros to make sure that there is a multiple of four bits in the number. For example, given the binary number 1011001010, the first step would be to add two bits to the left of the number so that it contains 12 bits. The converted binary value is 001011001010. The next step is to separate the binary value into groups of four bits, e.g., 0010_1100_1010. Finally, look up these binary values in the table above and substitute the appropriate hexadecimal digits, i.e., $2CA. Contrast this with the difficulty of conversion between decimal and binary or decimal and hexadecimal!

Since converting between hexadecimal and binary is an operation you will need to perform over and over again, you should take a few minutes and memorize the table above. Even if you have a calculator that will do the conversion for you, you'll find manual conversion to be a lot faster and more convenient when converting between binary and hex.

3.5   Arithmetic Operations on Binary and Hexadecimal Numbers

There are several operations we can perform on binary and hexadecimal numbers. For example, we can add, subtract, multiply, divide, and perform other arithmetic operations. Although you needn't become an expert at it, you should be able to, in a pinch, perform these operations manually using a piece of paper and a pencil. Having just said that you should be able to perform these operations manually, the correct way to perform such arithmetic operations is to have a calculator that does them for you. There are several such calculators on the market; the following table lists some of the manufacturers who produce such devices:

Some manufacturers of Hexadecimal Calculators (circa 2002):

• Casio
• Hewlett-Packard
• Sharp
• Texas Instruments

This list is by no means exhaustive. Other calculator manufacturers probably produce these devices as well. The Hewlett-Packard devices are arguably the best of the bunch . However, they are more expensive

than the others. Sharp and Casio produce units which sell for well under $50. If you plan on doing any assembly language programming at all, owning one of these calculators is essential.

To understand why you should spend the money on a calculator, consider the following arithmetic problem:

```
  $9
+ $1
----
```

You're probably tempted to write in the answer "$10" as the solution to this problem. But that is not correct! The correct answer is ten, which is "$A", not sixteen which is "$10". A similar problem exists with the arithmetic problem:

```
  $10
-  $1
----
```

You're probably tempted to answer "$9" even though the true answer is "$F". Remember, this problem is asking "what is the difference between sixteen and one?" The answer, of course, is fifteen which is "$F".

Even if the two problems above don't bother you, in a stressful situation your brain will switch back into decimal mode while you're thinking about something else and you'll produce the incorrect result. Moral of the story – if you must do an arithmetic computation using hexadecimal numbers by hand, take your time and be careful about it. Either that, or convert the numbers to decimal, perform the operation in decimal, and convert them back to hexadecimal.

3.6   A Note About Numbers vs. Representation

Many people confuse numbers and their representation. A common question beginning assembly language students have is "I've got a binary number in the EAX register, how do I convert that to a hexadecimal number in the EAX register?" The answer is "you don't." Although a strong argument could be made that numbers in memory or in registers are represented in binary, it's best to view values in memory or in a register as *abstract numeric quantities*. Strings of symbols like 128, $80, or %1000_0000 are not different numbers; they are simply different representations for the same abstract quantity that we often refer to as "one hundred twenty-eight." Inside the computer, a number is a number regardless of representation; the only time representation matters is when you input or output the value in a human readable form.

Human readable forms of numeric quantities are always strings of characters. To print the value 128 in human readable form, you must convert the numeric value 128 to the three-character sequence '1' followed by '2' followed by '8'. This would provide the decimal representation of the numeric quantity. If you prefer, you could convert the numeric value 128 to the three character sequence "$80". It's the same number, but we've converted it to a different sequence of characters because (presumably) we wanted to view the number using hexadecimal representation rather than decimal. Likewise, if we want to see the number in binary, then we must convert this numeric value to a string containing a one followed by seven zeros.

By default, HLA displays all byte, word, and dword variables using the hexadecimal numbering system when you use the *stdout.put* routine. Likewise, HLA's *stdout.put* routine will display all register values in hex. Consider the following program that converts values input as decimal numbers to their hexadecimal equivalents:

```
program ConvertToHex;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToHex;
```

```
        stdout.put( "Input a decimal value:" );
        stdin.get( value );
        mov( value, eax );
        stdout.put( "The value ", value, " converted to hex is $", eax, nl );

    end ConvertToHex;
```

Program 3.11   Decimal to Hexadecimal Conversion Program

In a similar fashion, the default input base is also hexadecimal for registers and byte, word, or dword variables. The following program is the converse of the one above- it inputs a hexadecimal value and outputs it as decimal:

```
program ConvertToDecimal;
#include( "stdlib.hhf" );
static
    value: int32;

begin ConvertToDecimal;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );  mov( ebx, value );
    stdout.put( "The value $", ebx, " converted to decimal is ", value, nl );

end ConvertToDecimal;
```

Program 3.12   Hexadecimal to Decimal Conversion Program

Just because the HLA *stdout.put* routine chooses decimal as the default output base for *int8*, *int16*, and *int32* variables doesn't mean that these variables hold "decimal" numbers. Remember, memory and registers hold numeric values, not hexadecimal or decimal values. The *stdout.put* routine converts these numeric values to strings and prints the resulting strings. The choice of hexadecimal vs. decimal output was a design choice in the HLA language, nothing more. You could very easily modify HLA so that it outputs registers and *byte*, *word*, or *dword* variables as decimal values rather than as hexadecimal. If you need to print the value of a register or *byte*, *word*, or *dword* variable as a decimal value, simply call one of the putiX routines to do this. The *stdout.puti8* routine will output its parameter as an eight-bit signed integer. Any eight-bit parameter will work. So you could pass an eight-bit register, an *int8* variable, or a *byte* variable as the parameter to *stdout.puti8* and the result will always be decimal. The *stdout.puti16* and *stdout.puti32* provide the same capabilities for 16-bit and 32-bit objects. The following program demonstrates the decimal conversion program (Program 3.12 above) using only the EAX register (i.e., it does not use the variable *iValue*):

```
program ConvertToDecimal2;
#include( "stdlib.hhf" );
begin ConvertToDecimal2;

    stdout.put( "Input a hexadecimal value: " );
    stdin.get( ebx );
    stdout.put( "The value $", ebx, " converted to decimal is " );
    stdout.puti32( ebx );
    stdout.newln();
```

```
end ConvertToDecimal2;
```

Program 3.13   Variable-less Hexadecimal to Decimal Converter

---

Note that HLA's *stdin.get* routine uses the same default base for input as *stdout.put* uses for output. That is, if you attempt to read an *int8*, *int16*, or *int32* variable, the default input base is decimal. If you attempt to read a register or *byte*, *word*, or *dword* variable, the default input base is hexadecimal. If you want to change the default input base to decimal when reading a register or a *byte*, *word*, or *dword* variable, then you can use *stdin.geti8*, *stdin.geti16*, or *stdin.geti32*.

If you want to go in the opposite direction, that is you want to input or output an *int8*, *int16*, or *int32* variable as a hexadecimal value, you can call the *stdout.putb*, *stdout.putw*, *stdout.putd*, *stdin.getb*, *stdin.getw*, or *stdin.getd* routines. The *stdout.putb*, *stdout.putw*, and *stdout.putd* routines write eight-bit, 16-bit, or 32-bit objects as hexadecimal values. The *stdin.getb*, *stdin.getw*, and *stdin.getd* routines read eight-bit, 16-bit, and 32-bit values respectively; they return their results in the AL, AX, or EAX registers. The following program demonstrates the use of a few of these routines:

```
program HexIO;

#include( "stdlib.hhf" );

static
    i32: int32;

begin HexIO;

    stdout.put( "Enter a hexadecimal value: " );
    stdin.getd();
    mov( eax, i32 );
    stdout.put( "The value you entered was $" );
    stdout.putd( i32 );
    stdout.newln();

end HexIO;
```

Program 3.14   Demonstration of stdin.getd and stdout.putd

3.7   Logical Operations on Bits

There are four main logical operations we'll need to perform on hexadecimal and binary numbers: AND, OR, XOR (exclusive-or), and NOT. Unlike the arithmetic operations, a hexadecimal calculator isn't necessary to perform these operations. It is often easier to do them by hand than to use an electronic device

to compute them. The logical AND operation is a dyadic[3] operation (meaning it accepts exactly two oper-
ands). These operands are single binary (base 2) bits. The AND operation is:

$$0 \text{ and } 0 = 0$$

$$0 \text{ and } 1 = 0$$

$$1 \text{ and } 0 = 0$$

$$1 \text{ and } 1 = 1$$

A compact way to represent the logical AND operation is with a truth table. A truth table takes the fol-
lowing form:

Table 5: AND Truth Table

AND	0	1
0	0	0
1	0	1

This is just like the multiplication tables you encountered in elementary school. The values in the left
column correspond to the leftmost operand of the AND operation. The values in the top row correspond to
the rightmost operand of the AND operation. The value located at the intersection of the row and column
(for a particular pair of input values) is the result of logically ANDing those two values together.

In English, the logical AND operation is, "If the first operand is one and the second operand is one, the
result is one; otherwise the result is zero." We could also state this as "If either or both operands are zero, the
result is zero."

One important fact to note about the logical AND operation is that you can use it to force a zero result.
If one of the operands is zero, the result is always zero regardless of the other operand. In the truth table
above, for example, the row labelled with a zero input contains only zeros and the column labelled with a
zero only contains zero results. Conversely, if one operand contains a one, the result is exactly the value of
the second operand. These features of the AND operation are very important, particularly when we want to
force individual bits in a bit string to zero. We will investigate these uses of the logical AND operation in the
next section.

The logical OR operation is also a dyadic operation. Its definition is:

$$0 \text{ or } 0 = 0$$

$$0 \text{ or } 1 = 1$$

$$1 \text{ or } 0 = 1$$

$$1 \text{ or } 1 = 1$$

---

3. Many texts call this a binary operation. The term dyadic means the same thing and avoids the confusion with the binary
numbering system.

                                  Beta Draft - Do not distribute

The truth table for the OR operation takes the following form:

Table 6: OR Truth Table

OR	0	1
0	0	1
1	1	1

Colloquially, the logical OR operation is, "If the first operand or the second operand (or both) is one, the result is one; otherwise the result is zero." This is also known as the *inclusive-OR* operation.

If one of the operands to the logical-OR operation is a one, the result is always one regardless of the second operand's value. If one operand is zero, the result is always the value of the second operand. Like the logical AND operation, this is an important side-effect of the logical-OR operation that will prove quite useful when working with bit strings since it lets you force individual bits to one.

Note that there is a difference between this form of the inclusive logical OR operation and the standard English meaning. Consider the phrase "I am going to the store *or* I am going to the park." Such a statement implies that the speaker is going to the store or to the park but not to both places. Therefore, the English version of logical OR is slightly different than the inclusive-OR operation; indeed, it is closer to the *exclusive-OR* operation.

The logical XOR (exclusive-or) operation is also a dyadic operation. It is defined as follows:

$$0 \text{ xor } 0 = 0$$

$$0 \text{ xor } 1 = 1$$

$$1 \text{ xor } 0 = 1$$

$$1 \text{ xor } 1 = 0$$

The truth table for the XOR operation takes the following form:

Table 7: XOR Truth Table

XOR	0	1
0	0	1
1	1	0

In English, the logical XOR operation is, "If the first operand or the second operand, but not both, is one, the result is one; otherwise the result is zero." Note that the exclusive-or operation is closer to the English meaning of the word "or" than is the logical OR operation.

If one of the operands to the logical exclusive-OR operation is a one, the result is always the *inverse* of the other operand; that is, if one operand is one, the result is zero if the other operand is one and the result is one if the other operand is zero. If the first operand contains a zero, then the result is exactly the value of the second operand. This feature lets you selectively invert bits in a bit string.

The logical NOT operation is a monadic operation (meaning it accepts only one operand). It is:

$$NOT\ 0 = 1$$

$$NOT\ 1 = 0$$

The truth table for the NOT operation takes the following form:

Table 8: NOT Truth Table

NOT	0	1
	1	0

3.8   Logical Operations on Binary Numbers and Bit Strings

As described in the previous section, the logical functions work only with single bit operands. Since the 80x86 uses groups of eight, sixteen, or thirty-two bits, we need to extend the definition of these functions to deal with more than two bits. Logical functions on the 80x86 operate on a *bit-by-bit* (or *bitwise*) basis. Given two values, these functions operate on bit zero producing bit zero of the result. They operate on bit one of the input values producing bit one of the result, etc. For example, if you want to compute the logical AND of the following two eight-bit numbers, you would perform the logical AND operation on each column independently of the others:

```
%1011_0101
%1110_1110
----------
%1010_0100
```

This bit-by-bit form of execution can be easily applied to the other logical operations as well.

Since we've defined logical operations in terms of binary values, you'll find it much easier to perform logical operations on binary values than on values in other bases. Therefore, if you want to perform a logical operation on two hexadecimal numbers, you should convert them to binary first. This applies to most of the basic logical operations on binary numbers (e.g., AND, OR, XOR, etc.).

The ability to force bits to zero or one using the logical AND/OR operations and the ability to invert bits using the logical XOR operation is very important when working with strings of bits (e.g., binary numbers). These operations let you selectively manipulate certain bits within some value while leaving other bits unaffected. For example, if you have an eight-bit binary value X and you want to guarantee that bits four through seven contain zeros, you could logically AND the value X with the binary value %0000_1111. This bitwise logical AND operation would force the H.O. four bits to zero and pass the L.O. four bits of X through unchanged. Likewise, you could force the L.O. bit of X to one and invert bit number two of X by logically ORing X with %0000_0001 and logically exclusive-ORing X with %0000_0100, respectively. Using the logical AND, OR, and XOR operations to manipulate bit strings in this fashion is known as *masking* bit strings. We use the term *masking* because we can use certain values (one for AND, zero for OR/XOR) to 'mask out' or 'mask in' certain bits from the operation when forcing bits to zero, one, or their inverse.

The 80x86 CPUs support four instructions that apply these bitwise logical operations to their operands. The instructions are AND, OR, XOR, and NOT. The AND, OR, and XOR instructions use the same syntax as the ADD and SUB instructions, that is,

```
and( source, dest );
 or( source, dest );
xor( source, dest );
```

These operands have the same limitations as the ADD operands. Specifically, the *source* operand has to be a constant, memory, or register operand and the *dest* operand must be a memory or register operand. Also, the operands must be the same size and they cannot both be memory operands. These instructions compute the obvious bitwise logical operation via the equation:

$$dest = dest \text{ operator } source$$

The 80x86 logical NOT instruction, since it has only a single operand, uses a slightly different syntax. This instruction takes the following form:

```
not( dest );
```

Note that this instruction has a single operand. It computes the following result:

$$dest = \text{NOT}(\ dest\ )$$

The *dest* operand (for *not*) must be a register or memory operand. This instruction inverts all the bits in the specified destination operand.

The following program inputs two hexadecimal values from the user and calculates their logical AND, OR, XOR, and NOT:

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

    stdout.put( "Input left operand: " );
    stdin.get( eax );
    stdout.put( "Input right operand: " );
    stdin.get( ebx );

    mov( eax, ecx );
    and( ebx, ecx );
    stdout.put( "$", eax, " AND $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    or( ebx, ecx );
    stdout.put( "$", eax, " OR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    xor( ebx, ecx );
    stdout.put( "$", eax, " XOR $", ebx, " = $", ecx, nl );

    mov( eax, ecx );
    not( ecx );
    stdout.put( "NOT $", eax, " = $", ecx, nl );

    mov( ebx, ecx );
    not( ecx );
    stdout.put( "NOT $", ebx, " = $", ecx, nl );

end LogicalOp;
```

Program 3.15   AND, OR, XOR, and NOT Example

3.9   Signed and Unsigned Numbers

So far, we've treated binary numbers as unsigned values. The binary number ...00000 represents zero, ...00001 represents one, ...00010 represents two, and so on toward infinity. What about negative numbers? Signed values have been tossed around in previous sections and we've mentioned the two's complement

numbering system, but we haven't discussed how to represent negative numbers using the binary numbering system. That is what this section is all about!

To represent signed numbers using the binary numbering system we have to place a restriction on our numbers: they must have a finite and fixed number of bits. For our purposes, we're going to severely limit the number of bits to eight, 16, 32, or some other small number of bits.

With a fixed number of bits we can only represent a certain number of objects. For example, with eight bits we can only represent 256 different values. Negative values are objects in their own right, just like positive numbers; therefore, we'll have to use some of the 256 different eight-bit values to represent negative numbers. In other words, we've got to use up some of the (unsigned) positive numbers to represent negative numbers. To make things fair, we'll assign half of the possible combinations to the negative values and half to the positive values and zero. So we can represent the negative values -128..-1 and the non-negative values 0..127 with a single eight bit byte. With a 16-bit word we can represent values in the range -32,768..+32,767. With a 32-bit double word we can represent values in the range -2,147,483,648..+2,147,483,647. In general, with n bits we can represent the signed values in the range -2^{n-1} to $+2^{n-1}-1$.

Okay, so we can represent negative values. Exactly how do we do it? Well, there are many ways, but the 80x86 microprocessor uses the two's complement notation. In the two's complement system, the H.O. bit of a number is a *sign bit*. If the H.O. bit is zero, the number is positive; if the H.O. bit is one, the number is negative. Examples:

For 16-bit numbers:

$8000 is negative because the H.O. bit is one.

$100 is positive because the H.O. bit is zero.

$7FFF is positive.

$FFFF is negative.

$FFF is positive.

If the H.O. bit is zero, then the number is positive and is stored as a standard binary value. If the H.O. bit is one, then the number is negative and is stored in the two's complement form. To convert a positive number to its negative, two's complement form, you use the following algorithm:

1)   Invert all the bits in the number, i.e., apply the logical NOT function.

2)   Add one to the inverted result.

For example, to compute the eight-bit equivalent of -5:

```
%0000_0101     Five (in binary).
%1111_1010     Invert all the bits.
%1111_1011     Add one to obtain result.
```

If we take minus five and perform the two's complement operation on it, we get our original value, %0000_0101, back again, just as we expect:

```
%1111_1011     Two's complement for -5.
%0000_0100     Invert all the bits.
%0000_0101     Add one to obtain result (+5).
```

The following examples provide some positive and negative 16-bit signed values:

$7FFF: +32767, the largest 16-bit positive number.

$8000: -32768, the smallest 16-bit negative number.

$4000: +16,384.

To convert the numbers above to their negative counterpart (i.e., to negate them), do the following:

```
$7FFF:      %0111_1111_1111_1111    +32,767
            %1000_0000_0000_0000    Invert all the bits (8000h)
            %1000_0000_0000_0001    Add one (8001h or -32,767)

4000h:      %0100_0000_0000_0000    16,384
            %1011_1111_1111_1111    Invert all the bits ($BFFF)
            %1100_0000_0000_0000    Add one ($C000 or -16,384)

$8000:      %1000_0000_0000_0000    -32,768
            %0111_1111_1111_1111    Invert all the bits ($7FFF)
            %1000_0000_0000_0000    Add one (8000h or -32768)
```

$8000 inverted becomes $7FFF. After adding one we obtain $8000! Wait, what's going on here? -(-32,768) is -32,768? Of course not. But the value +32,768 cannot be represented with a 16-bit signed number, so we cannot negate the smallest negative value.

Why bother with such a miserable numbering system? Why not use the H.O. bit as a sign flag, storing the positive equivalent of the number in the remaining bits? The answer lies in the hardware. As it turns out, negating values is the only tedious job. With the two's complement system, most other operations are as easy as the binary system. For example, suppose you were to perform the addition 5+(-5). The result is zero. Consider what happens when we add these two values in the two's complement system:

```
%   0000_0101
%   1111_1011
   ------------
%1_0000_0000
```

We end up with a carry into the ninth bit and all other bits are zero. As it turns out, if we ignore the carry out of the H.O. bit, adding two signed values always produces the correct result when using the two's complement numbering system. This means we can use the same hardware for signed and unsigned addition and subtraction. This wouldn't be the case with some other numbering systems.

Except for the questions associated with this chapter, you will not need to perform the two's complement operation by hand. The 80x86 microprocessor provides an instruction, NEG (negate), that performs this operation for you. Furthermore, all the hexadecimal calculators will perform this operation by pressing the change sign key (+/- or CHS). Nevertheless, performing a two's complement by hand is easy, and you should know how to do it.

Once again, you should note that the data represented by a set of binary bits depends entirely on the context. The eight bit binary value %1100_0000 could represent an IBM/ASCII character, it could represent the unsigned decimal value 192, or it could represent the signed decimal value -64. As the programmer, it is your responsibility to use this data consistently.

The 80x86 negate instruction, NEG, uses the same syntax as the NOT instruction; that is, it takes a single destination operand:

```
neg( dest );
```

This instruction computes "dest = -dest;" and the operand has the same limitations as for NOT (it must be a memory location or a register). NEG operates on byte, word, and dword-sized objects. Of course, since this is a signed integer operation, it only makes sense to operate on signed integer values. The following program demonstrates the two's complement operation by using the NEG instruction:

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

static
```

```
       PosValue:    int8;
       NegValue:    int8;

   begin twosComplement;

       stdout.put( "Enter an integer between 0 and 127: " );
       stdin.get( PosValue );

       stdout.put( nl, "Value in hexadecimal: $" );
       stdout.putb( PosValue );

       mov( PosValue, al );
       not( al );
       stdout.put( nl, "Invert all the bits: $", al, nl );
       add( 1, al );
       stdout.put( "Add one: $", al, nl );
       mov( al, NegValue );
       stdout.put( "Result in decimal: ", NegValue, nl );

       stdout.put
       (
           nl,
           "Now do the same thing with the NEG instruction: ",
           nl
       );
       mov( PosValue, al );
       neg( al );
       mov( al, NegValue );
       stdout.put( "Hex result = $", al, nl );
       stdout.put( "Decimal result = ", NegValue, nl );

   end twosComplement;
```

Program 3.16   The Two's Complement Operation

As you saw in the previous chapters, you use the *int8*, *int16*, and *int32* data types to reserve storage for signed integer variables. Those chapters also introduced routines like *stdout.puti8* and *stdin.geti32* that read and write signed integer values. Since this section has made it abundantly clear that you must differentiate signed and unsigned calculations in your programs, you should probably be asking yourself about now "how do I declare and use unsigned integer variables?"

The first part of the question, "how do you declare unsigned integer variables," is the easiest to answer. You simply use the *uns8*, *uns16*, and *uns32* data types when declaring the variables, for example:

```
static
    u8:     uns8;
    u16:    uns16;
    u32:    uns32;
```

As for using these unsigned variables, the HLA Standard Library provides a complementary set of input/output routines for reading and displaying unsigned variables. As you can probably guess, these routines include *stdout.putu8*, *stdout.putu16*, *stdout.putu32*, *stdout.putu8Size*, *stdout.putu16Size*, *stdout.putu32Size*, *stdin.getu8, stdin.getu16,* and *stdin.getu32*. You use these routines just as you would use their signed integer counterparts except, of course, you get to use the full range of the unsigned values with these routines. The following source code demonstrates unsigned I/O as well as demonstrating what can happen if you mix signed and unsigned operations in the same calculation:

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

static
    UnsValue:   uns16;

begin UnsExample;

    stdout.put( "Enter an integer between 32,768 and 65,535: " );
    stdin.getu16();
    mov( ax, UnsValue );

    stdout.put
    (
        "You entered ",
        UnsValue,
        ".  If you treat this as a signed integer, it is "
    );
    stdout.puti16( UnsValue );
    stdout.newln();

end UnsExample;
```

Program 3.17   Unsigned I/O

3.10   Sign Extension, Zero Extension, Contraction, and Saturation

Since two's complement format integers have a fixed length, a small problem develops. What happens if you need to convert an eight bit two's complement value to 16 bits? This problem, and its converse (converting a 16 bit value to eight bits) can be accomplished via *sign extension* and *contraction* operations. Likewise, the 80x86 works with fixed length values, even when processing unsigned binary numbers. *Zero extension* lets you convert small unsigned values to larger unsigned values.

Consider the value "-64". The eight bit two's complement value for this number is $C0. The 16-bit equivalent of this number is $FFC0. Now consider the value "+64". The eight and 16 bit versions of this value are $40 and $0040, respectively. The difference between the eight and 16 bit numbers can be described by the rule: "If the number is negative, the H.O. byte of the 16 bit number contains $FF; if the number is positive, the H.O. byte of the 16 bit quantity is zero."

To sign extend a value from some number of bits to a greater number of bits is easy, just copy the sign bit into all the additional bits in the new format. For example, to sign extend an eight bit number to a 16 bit number, simply copy bit seven of the eight bit number into bits 8..15 of the 16 bit number. To sign extend a 16 bit number to a double word, simply copy bit 15 into bits 16..31 of the double word.

You must use sign extension when manipulating signed values of varying lengths. Often you'll need to add a byte quantity to a word quantity. You must sign extend the byte quantity to a word before the operation takes place. Other operations (multiplication and division, in particular) may require a sign extension to 32-bits. You must not sign extend unsigned values.

```
Sign Extension:
Eight Bits  Sixteen Bits  Thirty-two Bits

 $80        $FF80         $FFFF_FF80
 $28        $0028         $0000_0028
 $9A        $FF9A         $FFFF_FF9A
```

```
$7F          $007F            $0000_007F
---          $1020            $0000_1020
---          $8086            $FFFF_8086
```

To extend an unsigned byte you must zero extend the value. Zero extension is very easy – just store a zero into the H.O. byte(s) of the larger operand. For example, to zero extend the value $82 to 16-bits you simply add a zero to the H.O. byte yielding $0082.

```
Zero Extension:
Eight Bits     Sixteen Bits          Thirty-two Bits

  $80            $0080                 $0000_0080
  $28            $0028                 $0000_0028
  $9A            $009A                 $0000_009A
  $7F            $007F                 $0000_007F
  ---            $1020                 $0000_1020
  ---            $8086                 $0000_8086
```

The 80x86 provides several instructions that will let you sign or zero extend a smaller number to a larger number. The first group of instructions we will look at will sign extend the AL, AX, or EAX register. These instructions are

- cbw();   // Converts the byte in AL to a word in AX via sign extension.
- cwd();   // Converts the word in AX to a double word in DX:AX
- cdq();   // Converts the double word in EAX to the quad word in EDX:EAX
- cwde();   // Converts the word in AX to a doubleword in EAX.

Note that the CWD (convert word to doubleword) instruction does not sign extend the word in AX to the doubleword in EAX. Instead, it stores the H.O. doubleword of the sign extension into the DX register (the notation "DX:AX" tells you that you have a double word value with DX containing the upper 16 bits and AX containing the lower 16 bits of the value). If you want the sign extension of AX to go into EAX, you should use the CWDE (convert word to doubleword, extended) instruction.

The four instructions above are unusual in the sense that these are the first instructions you've seen that do not have any operands. These instructions' operands are *implied* by the instructions themselves.

Within a few chapters you will discover just how important these instructions are, and why the CWD and CDQ instructions involve the DX and EDX registers. However, for simple sign extension operations, these instructions have a few major drawbacks - you do not get to specify the source and destination operands and the operands must be registers.

For general sign extension operations, the 80x86 provides an extension of the MOV instruction, MOVSX (move with sign extension), that copies data and sign extends the data while copying it. The MOVSX instruction's syntax is very similar to the MOV instruction:

<p style="text-align:center">movsx(source, dest);</p>

The big difference in syntax between this instruction and the MOV instruction is the fact that the destination operand must be larger than the source operand. That is, if the source operand is a byte, the destination operand must be a word or a double word. Likewise, if the source operand is a word, the destination operand must be a double word. Another difference is that the destination operand has to be a register; the source operand, however, can be a memory location[4].

To zero extend a value, you can use the MOVZX instruction. It has the same syntax and restrictions as the MOVSX instruction. Zero extending certain eight-bit registers (AL, BL, CL, and DL) into their corresponding 16-bit registers is easily accomplished without using MOVZX by loading the complementary H.O.

---

4. This doesn't turn out to be much of a limitation because sign extension almost always precedes an arithmetic operation which must take place in a register.

register (AH, BH, CH, or DH) with zero. Obviously, to zero extend AX into DX:AX or EAX into EDX:EAX, all you need to do is load DX or EDX with zero[5].

The following sample program demonstrates the use of the sign extension instructions:

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

static
    i8:     int8;
    i16:    int16;
    i32:    int32;

begin signExtension;

    stdout.put( "Enter a small negative number: " );
    stdin.get( i8 );

    stdout.put( nl, "Sign extension using CBW and CWDE:", nl, nl );

    mov( i8, al );
    stdout.put( "You entered ", i8, " ($", al, ")", nl );

    cbw();
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    cwde();
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

    stdout.put( nl, "Sign extension using MOVSX:", nl, nl );

    movsx( i8, ax );
    mov( ax, i16 );
    stdout.put( "16-bit sign extension: ", i16, " ($", ax, ")", nl );

    movsx( i8, eax );
    mov( eax, i32 );
    stdout.put( "32-bit sign extension: ", i32, " ($", eax, ")", nl );

end signExtension;
```

Program 3.18   Sign Extension Instructions

Sign contraction, converting a value with some number of bits to the identical value with a fewer number of bits, is a little more troublesome. Sign extension never fails. Given an *m*-bit signed value you can always convert it to an *n*-bit number (where $n > m$) using sign extension. Unfortunately, given an *n*-bit number, you cannot always convert it to an *m*-bit number if $m < n$. For example, consider the value -448. As a 16-bit hexadecimal number, its representation is $FE40. Unfortunately, the magnitude of this number is too large to fit into an eight bit value, so you cannot sign contract it to eight bits. This is an example of an overflow condition that occurs upon conversion.

---

5. Zero extending into DX:AX or EDX:EAX is just as necessary as the CWD and CDQ instructions, as you will eventually see.

To properly sign contract one value to another, you must look at the H.O. byte(s) that you want to discard. The H.O. bytes you wish to remove must all contain either zero or $FF. If you encounter any other values, you cannot contract it without overflow. Finally, the H.O. bit of your resulting value must match *every* bit you've removed from the number. Examples (16 bits to eight bits):

```
$FF80 can be sign contracted to $80.
$0040 can be sign contracted to $40.
$FE40 cannot be sign contracted to 8 bits.
$0100 cannot be sign contracted to 8 bits.
```

Another way to reduce the size of an integer is through saturation. Saturation is useful in situations where you must convert a larger object to a smaller object and you're willing to live with possible loss of precision. To convert a value via saturation you simply copy the larger value to the smaller value if it is not outside the range of the smaller object. If the larger value is outside the range of the smaller value, then you *clip* the value by setting it to the largest (or smallest) value within the range of the smaller object.

For example, when converting a 16-bit signed integer to an eight-bit signed integer, if the 16-bit value is in the range -128..+127 you simply copy the L.O. byte of the 16-bit object to the eight-bit object. If the 16-bit signed value is greater than +127, then you clip the value to +127 and store +127 into the eight-bit object. Likewise, if the value is less than -128, you clip the final eight bit object to -128. Saturation works the same way when clipping 32-bit values to smaller values. If the larger value is outside the range of the smaller value, then you simply set the smaller value to the value closest to the out of range value that you can represent with the smaller value.

Obviously, if the larger value is outside the range of the smaller value, then there will be a loss of precision during the conversion. While clipping the value to the limits the smaller object imposes is never desirable, sometimes this is acceptable as the alternative is to raise an exception or otherwise reject the calculation. For many applications, such as audio or video processing, the clipped result is still recognizable, so this is a reasonable conversion to use.

3.11   Shifts and Rotates

Another set of logical operations which apply to bit strings are the *shift* and *rotate* operations. These two categories can be further broken down into *left shifts, left rotates, right shifts,* and *right rotates.* These operations turn out to be extremely useful to assembly language programmers.

The left shift operation moves each bit in a bit string one position to the left (see Figure 3.8).
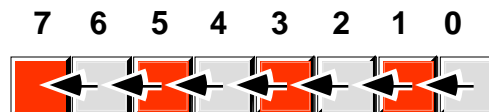

Figure 3.8   Shift Left Operation

Bit zero moves into bit position one, the previous value in bit position one moves into bit position two, etc. There are, of course, two questions that naturally arise: "What goes into bit zero?" and "Where does bit seven wind up?"   We'll shift a zero into bit zero and the previous value of bit seven will be the *carry* out of this operation.

The 80x86 provides a shift left instruction, SHL, that performs this useful operation. The syntax for the SHL instruction is the following:

```
shl( count, dest );
```

The count operand is either "CL" or a constant in the range 0..n, where n is one less than the number of bits in the destination operand (i.e., n=7 for eight-bit operands, n=15 for 16-bit operands, and n=31 for 32-bit operands). The dest operand is a typical dest operand, it can be either a memory location or a register.

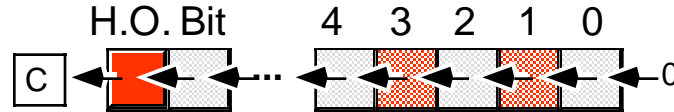When the count operand is the constant one, the SHL instruction does the following:

Figure 3.9   Operation of the SHL(1, Dest) Instruction

In Figure 3.9, the "C" represents the carry flag. That is, the bit shifted out of the H.O. bit of the operand is moved into the carry flag. Therefore, you can test for overflow after a SHL(1, dest) instruction by testing the carry flag immediately after executing the instruction (e.g., by using "if(@c) then..." or "if(@nc) then...").

Intel's literature suggests that the state of the carry flag is undefined if the shift count is a value other than one. Usually, the carry flag contains the last bit shifted out of the destination operand, but Intel doesn't seem to guarantee this. If you need to shift more than one bit out of an operand and you need to capture all the bits you shift out, you should take a look at the SHLD and SHRD instructions in the appendicies.

Note that shifting a value to the left is the same thing as multiplying it by its radix. For example, shifting a decimal number one position to the left (adding a zero to the right of the number) effectively multiplies it by ten (the radix):

```
1234 shl 1 = 12340  (shl 1 means shift one digit position to the left)
```

Since the radix of a binary number is two, shifting it left multiplies it by two. If you shift a binary value to the left twice, you multiply it by two twice (i.e., you multiply it by four). If you shift a binary value to the left three times, you multiply it by eight (2*2*2). In general, if you shift a value to the left n times, you multiply that value by 2^n.

A right shift operation works the same way, except we're moving the data in the opposite direction. Bit seven moves into bit six, bit six moves into bit five, bit five moves into bit four, etc. During a right shift, we'll move a zero into bit seven, and bit zero will be the carry out of the operation (see Figure 3.10).
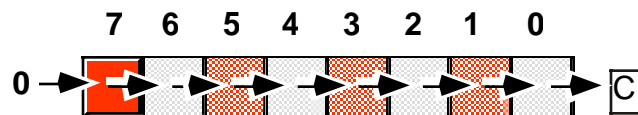

Figure 3.10   Shift Right Operation

As you would probably expect by now, the 80x86 provides a SHR instruction that will shift the bits to the right in a destination operand. The syntax is the same as the SHL instruction except, of course, you specify SHR rather than SHL:

```
SHR( count, dest );
```

This instruction shifts a zero into the H.O. bit of the destination operand, it shifts all the other bits one place to the right (that is, from a higher bit number to a lower bit number). Finally, bit zero is shifted into the carry flag. If you specify a count of one, the SHR instruction does the following:
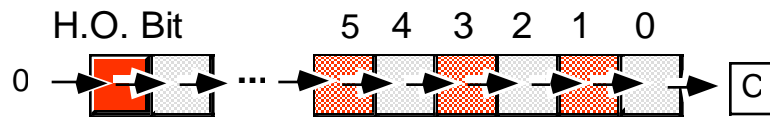

Figure 3.11   SHR(1, Dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

Since a left shift is equivalent to a multiplication by two, it should come as no surprise that a right shift is roughly comparable to a division by two (or, in general, a division by the radix of the number). If you perform n right shifts, you will divide that number by 2^n.

There is one problem with shift rights with respect to division: as described above a shift right is only equivalent to an *unsigned* division by two. For example, if you shift the unsigned representation of 254 (0FEh) one place to the right, you get 127 (07Fh), exactly what you would expect. However, if you shift the binary representation of -2 (0FEh) to the right one position, you get 127 (07Fh), which is *not* correct. This problem occurs because we're shifting a zero into bit seven. If bit seven previously contained a one, we're changing it from a negative to a positive number. Not a good thing when dividing by two.

To use the shift right as a division operator, we must define a third shift operation: *arithmetic shift right*[6]. An arithmetic shift right works just like the normal shift right operation (a *logical shift right*) with one exception: instead of shifting a zero into bit seven, an arithmetic shift right operation leaves bit seven alone, that is, during the shift operation it does not modify the value of bit seven as Figure 3.12 shows.
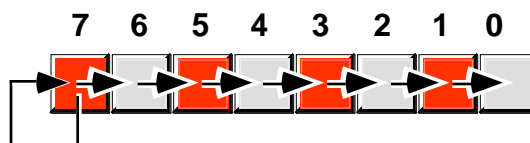

Figure 3.12   Arithmetic Shift Right Operation

This generally produces the result you expect. For example, if you perform the arithmetic shift right operation on -2 (0FEh) you get -1 (0FFh). Keep one thing in mind about arithmetic shift right, however. This operation always rounds the numbers to the closest integer *which is less than or equal to the actual result*. Based on experiences with high level programming languages and the standard rules of integer truncation, most people assume this means that a division always truncates towards zero. But this simply isn't the case. For example, if you apply the arithmetic shift right operation on -1 (0FFh), the result is -1, not zero. -1 is less than zero so the arithmetic shift right operation rounds towards minus one. This is not a "bug" in the arithmetic shift right operation, it's just uses a diffferent (though valid) definition of integer division.

_____

6. There is no need for an arithmetic shift left. The standard shift left operation works for both signed and unsigned numbers, assuming no overflow occurs.

The 80x86 provides an arithmetic shift right instruction, SAR (shift arithmetic right). This instruction's syntax is nearly identical to SHL and SHR. The syntax is

```
SAR( count, dest );
```

The usual limitations on the count and destination operands apply. This instruction does the following if the count is one:

Figure 3.13   SAR(1, dest) Operation

Once again, Intel's documents suggest that shifts of more than one bit leave the carry in an undefined state.

Another pair of useful operations are *rotate left* and *rotate right*. These operations behave like the shift left and shift right operations with one major difference: the bit shifted out from one end is shifted back in at the other end.

Figure 3.14   Rotate Left Operation

Figure 3.15   Rotate Right Operation

The 80x86 provides ROL (rotate left) and ROR (rotate right) instructions that do these basic operations on their operands. The syntax for these two instructions is similar to the shift instructions:

```
rol( count, dest );
ror( count, dest );
```

Once again, this instructions provide a special behavior if the shift count is one. Under this condition these two instructions also copy the bit shifted out of the destination operand into the carry flag as the following two figures show:

H.O. Bit   5   4   3   2   1   0
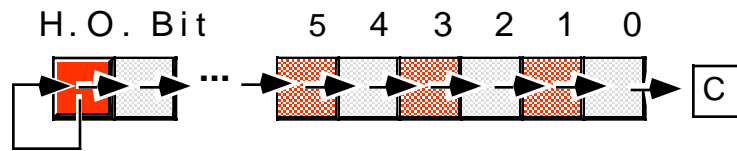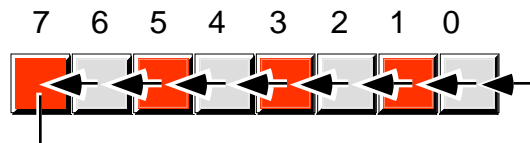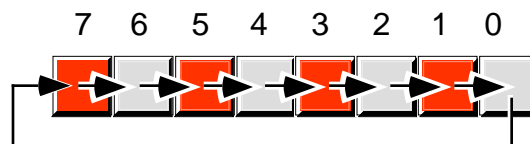
Figure 3.16   ROL(1, Dest) Operation

Note that, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

H.O. Bit   5   4   3   2   1   0
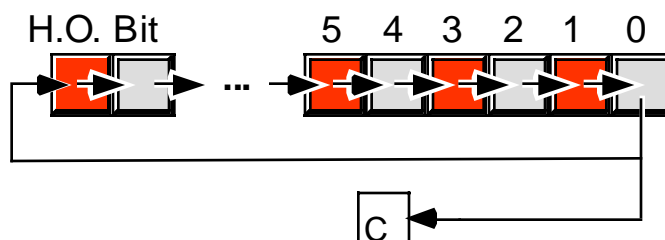
Figure 3.17   ROR(1, Dest) Operation

It will turn out that it is often more convenient for the rotate operation to shift the output bit through the carry and shift the previous carry value back into the input bit of the shift operation. The 80x86 RCL (rotate through carry left) and RCR (rotate through carry right) instructions achieve this for you. These instructions use the following syntax:

```
RCL( count, dest );
RCR( count, dest );
```

As is true for the other shift and rotate instructions, the count operand is either a constant or the CL register and the destination operand is a memory location or register. The count operand must be a value that is less than the number of bits in the destination operand. For a count value of one, these two instructions do the following:

H.O. Bit   5   4   3   2   1   0

C
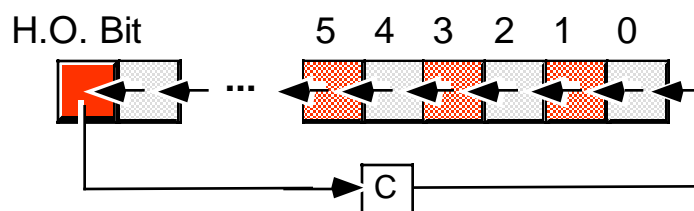
Figure 3.18   RCL(1, Dest) Operation
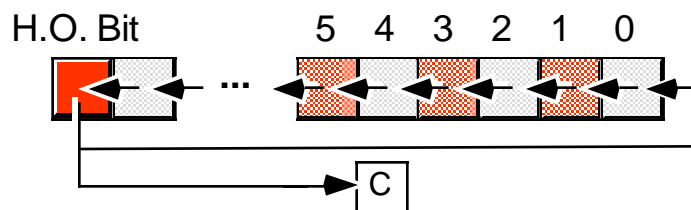
H.O. Bit   5   4   3   2   1   0

C
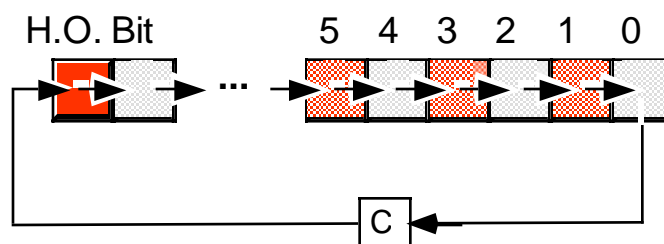
Figure 3.19   RCR(1, Dest) Operation

Again, Intel's documents suggest that rotates of more than one bit leave the carry in an undefined state.

3.12 Bit Fields and Packed Data

Although the 80x86 operates most efficiently on byte, word, and double word data types, occasionally you'll need to work with a data type that uses some number of bits other than eight, 16, or 32. For example, consider a date of the form "04/02/01". It takes three numeric values to represent this date: a month, day, and year value. Months, of course, take on the values 1..12. It will require at least four bits (maximum of sixteen different values) to represent the month. Days range between 1..31. So it will take five bits (maximum of 32 different values) to represent the day entry. The year value, assuming that we're working with values in the range 0..99, requires seven bits (which can be used to represent up to 128 different values). Four plus five plus seven is 16 bits, or two bytes. In other words, we can pack our date data into two bytes rather than the three that would be required if we used a separate byte for each of the month, day, and year values. This saves one byte of memory for each date stored, which could be a substantial saving if you need to store a lot of dates. The bits could be arranged as shown in the following figure:

15 14 13 12 11 10 9   8   7   6   5   4   3   2   1   0

M M M M D D D D D Y Y Y Y Y Y Y

Figure 3.20   Short Packed Date Format (Two Bytes)

MMMM represents the four bits making up the month value, DDDDD represents the five bits making up the day, and YYYYYYY is the seven bits comprising the year. Each collection of bits representing a data item is a *bit field*. April 2nd, 2001 would be represented as $4101:

```
0100    00010 0000001 = %0100_0001_0000_0001 or $4101
 4       2       01
```

Although packed values are *space efficient* (that is, very efficient in terms of memory usage), they are computationally *inefficient* (slow!). The reason? It takes extra instructions to unpack the data packed into the various bit fields. These extra instructions take additional time to execute (and additional bytes to hold the instructions); hence, you must carefully consider whether packed data fields will save you anything. The following sample program demonstrates the effort that must go into packing and unpacking this 16-bit date format:

```
program dateDemo;

#include( "stdlib.hhf" );

static
    day:        uns8;
    month:      uns8;
    year:       uns8;

    packedDate: word;

begin dateDemo;

    stdout.put( "Enter the current month, day, and year: " );
    stdin.get( month, day, year );

    // Pack the data into the following bits:
    //
    // 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
    //  m  m  m  m  d  d  d  d  d  y  y  y  y  y  y  y

    mov( 0, ax );
    mov( ax, packedDate );  //Just in case there is an error.
    if( month > 12 ) then

        stdout.put( "Month value is too large", nl );

    elseif( month = 0 ) then

        stdout.put( "Month value must be in the range 1..12", nl );

    elseif( day > 31 ) then

        stdout.put( "Day value is too large", nl );
```

```
        elseif( day = 0 ) then

            stdout.put( "Day value must be in the range 1..31", nl );

        elseif( year > 99 ) then

            stdout.put( "Year value must be in the range 0..99", nl );

        else

            mov( month, al );
            shl( 5, ax );
            or( day, al );
            shl( 7, ax );
            or( year, al );
            mov( ax, packedDate );

        endif;

        // Okay, display the packed value:

        stdout.put( "Packed data = $", packedDate, nl );



        // Unpack the date:

        mov( packedDate, ax );
        and( $7f, al );          // Retrieve the year value.
        mov( al, year );

        mov( packedDate, ax );  // Retrieve the day value.
        shr( 7, ax );
        and( %1_1111, al );
        mov( al, day );

        mov( packedDate, ax );  // Retrive the month value.
        rol( 4, ax );
        and( %1111, al );
        mov( al, month );

        stdout.put( "The date is ", month, "/", day, "/", year, nl );



    end dateDemo;
```

Program 3.19   Packing and Unpacking Date Data

Of course, having gone through the problems with Y2K, using a date format that limits you to 100 years (or even 127 years) would be quite foolish at this time. If you're concerned about your software running 100 years from now, perhaps it would be wise to use a three-byte date format rather than a two-byte format. As you will see in the chapter on arrays, however, you should always try to create data objects whose length is an even power of two (one byte, two bytes, four bytes, eight bytes, etc.) or you will pay a performance penalty. Hence, it is probably wise to go ahead and use four bytes and pack this data into a dword variable. Figure 3.21 shows a possible data organization for a four-byte date.

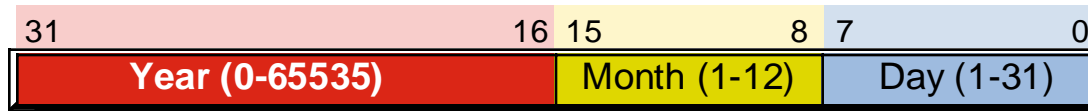| 31	16	15	8	7	0
Year (0-65535)		Month (1-12)		Day (1-31)	

Figure 3.21   Long Packed Date Format (Four Bytes)

In this long packed data format several changes were made beyond simply extending the number of bits associated with the year. First, since there are lots of extra bits in a 32-bit dword variable, this format allots extra bits to the month and day fields. Since these two fields consist of eight bits each, they can be easily extracted as a byte object from the dword. This leaves fewer bits for the year, but 65,536 years is probably sufficient; you can probably assume without too much concern that your software will not still be in use 63 thousand years from now when this date format will wrap around.

Of course, you could argue that this is no longer a packed date format. After all, we needed three numeric values, two of which fit just nicely into one byte each and one that should probably have at least two bytes. Since this "packed" date format consumes the same four bytes as the unpacked version, what is so special about this format? Well, another difference you will note between this long packed date format and the short date format appearing in Figure 3.20 is the fact that this long date format rearranges the bits so the *Year* is in the H.O. bit positions, the *Month* field is in the middle bit positions, and the *Day* field is in the L.O. bit positions. This is important because it allows you to very easily compare two dates to see if one date is less than, equal to, or greater than another date. Consider the following code:

```
mov( Date1, eax );        // Assume Date1 and Date2 are dword variables
if( eax > Date2 ) then    //  using the Long Packed Date format.

    << do something if Date1 > Date2 >>

endif;
```

Had you kept the different date fields in separate variables, or organized the fields differently, you would not have been able to compare *Date1* and *Date2* in such a straight-forward fashion. Therefore, this example demonstrates another reason for packing data even if you don't realize any space savings- it can make certain computations more convenient or even more efficient (contrary to what normally happens when you pack data).

Examples of practical packed data types abound. You could pack eight boolean values into a single byte, you could pack two BCD digits into a byte, etc. Of course, a classic example of packed data is the FLAGs register (see Figure 3.22). This register packs nine important boolean objects (along with seven important system flags) into a single 16-bit register. You will commonly need to access many of these flags. For this reason, the 80x86 instruction set provides many ways to manipulate the individual bits in the FLAGs register. Of course, you can test many of the condition code flags using the HLA @c, @nc, @z, @nz, etc., pseudo-boolean variables in an IF statement or other statement using a boolean expression.

In addition to the condition codes, the 80x86 provides instructions that directly affect certain flags. These instructions include the following:

- cld();   Clears (sets to zero) the direction flag.
- std();   Sets (to one) the direction flag.
- cli();   Clears the interrupt disable flag.
- sti();   Sets the interrupt disable flag.
- clc();   Clears the carry flag.
- stc();   Sets the carry flag.
- cmc();   Complements (inverts) the carry flag.
- sahf();   Stores the AH register into the L.O. eight bits of the FLAGs register.
- lahf();   Loads AH from the L.O. eight bits of the FLAGs register.

                                                 Beta Draft - Do not distribute

There are other instructions that affect the FLAGs register as well; these, however, demonstrate how to access several of the packed boolean values in the FLAGs register. The LAHF and SAHF instructions, in particular, provide a convenient way to access the L.O. eight bits of the FLAGs register as an eight-bit byte (rather than as eight separate one-bit values).

Overflow
Direction
Interrupt
Trace
Sign
Zero

Auxiliary Carry

Parity

Carry

Reserved for System Purposes
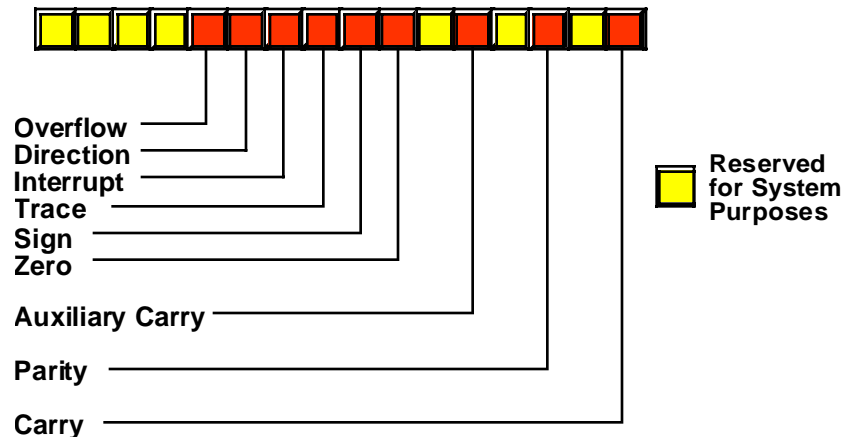
Figure 3.22   The FLAGs Register as a Packed Data Type

The LAHF (load AH with the L.O. eight bits of the FLAGs register) and the SAHF (store AH into the L.O. byte of the FLAGs register) use the following syntax:

```
lahf();
sahf();
```

3.13   Putting It All Together

In this chapter you've seen how we represent numeric values inside the computer. You've seen how to represent values using the decimal, binary, and hexadecimal numbering systems as well as the difference between signed and unsigned numeric representation. Since we represent nearly everything else inside a computer using numeric values, the material in this chapter is very important. Along with the base representation of numeric values, this chapter discusses the finite bit-string organization of data on typical computer systems, specfically bytes, words, and doublewords. Next, this chapter discusses arithmetic and logical operations on the numbers and presents some new 80x86 instructions to apply these operations to values inside the CPU. Finally, this chapter concludes by showing how you can pack several different numeric values into a fixed-length object (like a byte, word, or doubleword).

Absent from this chapter is any discussion of non-integer data. For example, how do we represent real numbers as well as integers? How do we represent characters, strings, and other non-numeric data? Well, that's the subject of the next chapter, so keep on reading...

More Data Representation   Chapter Four

4.1   Chapter Overview

Although the basic machine data objects (bytes, words, and double words) appear to represent nothing more than signed or unsigned numeric values, we can employ these data types to represent many other types of objects. This chapter discusses some of the other objects and their internal computer representation.

This chapter begins by discussing the floating point (real) numeric format. After integer representation, floating point representation is the second most popular numeric format in use on modern computer systems[1]. Although the floating point format is somewhat complex, the necessity to handle non-integer calculations in modern programs requires that you understand this numeric format and its limitations.

Binary Coded Decimal (BCD) is another numeric data representation that is useful in certain contexts. Although BCD is not suitable for general purpose arithmetic, it is useful in some embedded applications. The principle benefit of the BCD format is the ease with which you can convert between string and BCD format. When we look at the BCD format a little later in this chapter, you'll see why this is the case.

Computers can represent all kinds of different objects, not just numeric values. Characters are, unquestionably, one of the more popular data types a computer manipulates. In this chapter you will take a look at a couple of different ways we can represent individual characters on a computer system. This chapter discusses two of the more common character sets in use today: the ASCII character set and the Unicode character set.

This chapter concludes by discussing some common non-numeric data types like pixel colors on a video display, audio data, video data, and so on. Of course, there are lots of different representations for any kind of standard data you could envision; there is no way two chapters in a textbook can cover them all. (And that's not even considering specialized data types you could create). Nevertheless, this chapter (and the last) should give you the basic idea behind representing data on a computer system.

4.2   An Introduction to Floating Point Arithmetic

Integer arithmetic does not let you represent fractional numeric values. Therefore, modern CPUs support an approximation of *real* arithmetic: floating point arithmetic. A big problem with floating point arithmetic is that it does not follow the standard rules of algebra. Nevertheless, many programmers apply normal algebraic rules when using floating point arithmetic. This is a source of defects in many programs. One of the primary goals of this section is to describe the limitations of floating point arithmetic so you will understand how to use it properly.

Normal algebraic rules apply only to *infinite precision* arithmetic. Consider the simple statement "x:=x+1," x is an integer. On any modern computer this statement follows the normal rules of algebra *as long as overflow does not occur.* That is, this statement is valid only for certain values of x (*minint <= x < maxint*). Most programmers do not have a problem with this because they are well aware of the fact that integers in a program do not follow the standard algebraic rules (e.g., $5/2 \neq 2.5$).

Integers do not follow the standard rules of algebra because the computer represents them with a finite number of bits. You cannot represent any of the (integer) values above the maximum integer or below the minimum integer. Floating point values suffer from this same problem, only worse. After all, the integers are a subset of the real numbers. Therefore, the floating point values must represent the same infinite set of integers. However, there are an infinite number of values between any two real values, so this problem is infinitely worse. Therefore, as well as having to limit your values between a maximum and minimum range, you cannot represent all the values between those two ranges, either.

_____

1. There are other numeric formats, such as fixed point formats and binary coded decimal format.

To represent real numbers, most floating point formats employ scientific notation and use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*. The end result is that floating point numbers can only represent numbers with a specific number of *significant* digits. This has a big impact on how floating point arithmetic operates. To easily see the impact of limited precision arithmetic, we will adopt a simplified decimal floating point format for our examples. Our floating point format will provide a mantissa with three significant digits and a decimal exponent with two digits. The mantissa and exponents are both signed values as shown in Figure 4.1
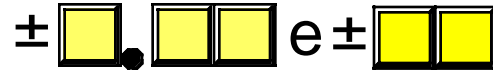

Figure 4.1   Simple Floating Point Format

When adding and subtracting two numbers in scientific notation, you must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, you must adjust the values so they have the same exponent. One way to do this is to convert 4.56e0 to 0.456e1 and then add. This produces 1.686e1. Unfortunately, the result does not fit into three significant digits, so we must either *round* or *truncate* the result to three significant digits. Rounding generally produces the most accurate result, so let's round the result to obtain 1.69e1. As you can see, the lack of *precision* (the number of digits or bits we maintain in a computation) affects the accuracy (the correctness of the computation).

In the previous example, we were able to round the result because we maintained *four* significant digits *during* the calculation. If our floating point calculation is limited to three significant digits *during* computation, we would have had to truncate the last digit of the smaller number, obtaining 1.68e1 which is even less correct. To improve the accuracy of floating point calculations, it is necessary to add extra digits for use during the calculation. Extra digits available during a computation are known as *guard digits* (or *guard bits* in the case of a binary format). They greatly enhance accuracy during a long chain of computations.

The accuracy loss during a single computation usually isn't enough to worry about unless you are greatly concerned about the accuracy of your computations. However, if you compute a value which is the result of a sequence of floating point operations, the error can *accumulate* and greatly affect the computation itself. For example, suppose we were to add 1.23e3 with 1.00e0. Adjusting the numbers so their exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these two values, even after rounding, is 1.23e3. This might seem perfectly reasonable to you; after all, we can only maintain three significant digits, adding in a small value shouldn't affect the result at all. However, suppose we were to add 1.00e0 to 1.23e3 *ten times*. The first time we add 1.00e0 to 1.23e3 we get 1.23e3. Likewise, we get this same result the second, third, fourth, ..., and tenth time we add 1.00e0 to 1.23e3. On the other hand, had we added 1.00e0 to itself ten times, then added the result (1.00e1) to 1.23e3, we would have gotten a different result, 1.24e3. This is an important thing to know about limited precision arithmetic:

> ❑   The order of evaluation can effect the accuracy of the result.

You will get more accurate results if the relative magnitudes (that is, the exponents) are close to one another. If you are performing a chain calculation involving addition and subtraction, you should attempt to group the values appropriately.

Another problem with addition and subtraction is that you can wind up with *false precision*. Consider the computation 1.23e0 - 1.22 e0. This produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this latter form suggests that the last two digits are exactly zero. Unfortunately, we've only got a single significant digit at this time. Indeed, some FPUs or floating point software packages might actually insert random digits (or bits) into the L.O. positions. This brings up a second important rule concerning limited precision arithmetic:

> ❑   Whenever subtracting two numbers with the same signs or adding two numbers with
>   different signs, the accuracy of the result may be less than the precision available in
>   the floating point format.

Multiplication and division do not suffer from the same problems as addition and subtraction since you do not have to adjust the exponents before the operation; all you need to do is add the exponents and multiply the mantissas (or subtract the exponents and divide the mantissas). By themselves, multiplication and division do not produce particularly poor results. However, they tend to multiply any error that already exists in a value. For example, if you multiply 1.23e0 by two, when you should be multiplying 1.24e0 by two, the result is even less accurate. This brings up a third important rule when working with limited precision arithmetic:

> ❑ When performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first.

Often, by applying normal algebraic transformations, you can arrange a calculation so the multiply and divide operations occur first. For example, suppose you want to compute x*(y+z). Normally you would add y and z together and multiply their sum by x. However, you will get a little more accuracy if you transform x*(y+z) to get x*y+x*z and compute the result by performing the multiplications first.

Multiplication and division are not without their own problems. When multiplying two very large or very small numbers, it is quite possible for *overflow* or *underflow* to occur. The same situation occurs when dividing a small number by a large number or dividing a large number by a small number. This brings up a fourth rule you should attempt to follow when multiplying or dividing values:

> ❑ When multiplying and dividing sets of numbers, try to arrange the multiplications so that they multiply large and small numbers together; likewise, try to divide numbers that have the same relative magnitudes.

Comparing floating point numbers is very dangerous. Given the inaccuracies present in any computation (including converting an input string to a floating point value), you should *never* compare two floating point values to see if they are equal. In a binary floating point format, different computations which produce the same (mathematical) result may differ in their least significant bits. For example, adding 1.31e0+1.69e0 should produce 3.00e0. Likewise, adding 1.50e0+1.50e0 should produce 3.00e0. However, were you to compare (1.31e0+1.69e0) against (1.50e0+1.50e0) you might find out that these sums are *not* equal to one another. The test for equality succeeds if and only if all bits (or digits) in the two operands are exactly the same. Since this is not necessarily true after two different floating point computations which should produce the same result, a straight test for equality may not work.

The standard way to test for equality between floating point numbers is to determine how much error (or tolerance) you will allow in a comparison and check to see if one value is within this error range of the other. The straight-forward way to do this is to use a test like the following:

```
if Value1 >= (Value2-error) and Value1 <= (Value2+error) then …
```

Another common way to handle this same comparison is to use a statement of the form:

```
if abs(Value1-Value2) <= error then …
```

Most texts, when discussing floating point comparisons, stop immediately after discussing the problem with floating point equality, assuming that other forms of comparison are perfectly okay with floating point numbers. This isn't true! If we are assuming that $x=y$ if x is within $y\pm error$, then a simple bitwise comparison of x and y will claim that $x<y$ if y is greater than x but less than $y+error$. However, in such a case x should really be treated as equal to y, not less than y. Therefore, we must always compare two floating point numbers using ranges, regardless of the actual comparison we want to perform. Trying to compare two floating point numbers directly can lead to an error. To compare two floating point numbers, x and y, against one another, you should use one of the following forms:

```
=    if abs(x-y) <= error then …
≠    if abs(x-y) > error then …
<    if (x-y) < -error then …
≤    if (x-y) <= error then …
>    if (x-y) > error then …
≥    if (x-y) >= -error then …
```

You must exercise care when choosing the value for *error*. This should be a value slightly greater than the largest amount of error which will creep into your computations. The exact value will depend upon the

particular floating point format you use, but more on that a little later. The final rule we will state in this section is

❑   When comparing two floating point numbers, always compare one value to see if it is in the range given by the second value plus or minus some small error value.

There are many other little problems that can occur when using floating point values. This text can only point out some of the major problems and make you aware of the fact that you cannot treat floating point arithmetic like real arithmetic – the inaccuracies present in limited precision arithmetic can get you into trouble if you are not careful. A good text on numerical analysis or even scientific computing can help fill in the details that are beyond the scope of this text. If you are going to be working with floating point arithmetic, *in any language*, you should take the time to study the effects of limited precision arithmetic on your computations.

HLA's IF statement does not support boolean expressions involving floating point operands. Therefore, you cannot use statements like "IF(x < 3.141) THEN..." in your programs. In a later chapter that discusses floating point operations on the 80x86 you'll learn how to do floating point comparisons.

4.2.1 IEEE Floating Point Formats

When Intel planned to introduce a floating point coprocessor for their new 8086 microprocessor, they were smart enough to realize that the electrical engineers and solid-state physicists who design chips were, perhaps, not the best people to do the necessary numerical analysis to pick the best possible binary representation for a floating point format. So Intel went out and hired the best numerical analyst they could find to design a floating point format for their 8087 FPU. That person then hired two other experts in the field and the three of them (Kahn, Coonan, and Stone) designed Intel's floating point format. They did such a good job designing the KCS Floating Point Standard that the IEEE organization adopted this format for the IEEE floating point format[2].

To handle a wide range of performance and accuracy requirements, Intel actually introduced *three* floating point formats: single precision, double precision, and extended precision. The single and double precision formats corresponded to C's float and double types or FORTRAN's real and double precision types. Intel intended to use extended precision for long chains of computations. Extended precision contains 16 extra bits that the calculations could use as guard bits before rounding down to a double precision value when storing the result.

The single precision format uses a one's complement 24 bit mantissa and an eight bit excess-127 exponent. The mantissa usually represents a value between 1.0 to just under 2.0. The H.O. bit of the mantissa is always assumed to be one and represents a value just to the left of the *binary point*[3]. The remaining 23 mantissa bits appear to the right of the binary point. Therefore, the mantissa represents the value:

```
1.mmmmmmmm mmmmmmmmm mmmmmmmmm
```

The "mmmm…" characters represent the 23 bits of the mantissa. Keep in mind that we are working with binary numbers here. Therefore, each position to the right of the binary point represents a value (zero or one) times a successive negative power of two. The implied one bit is always multiplied by 2^0, which is one. This is why the mantissa is always greater than or equal to one. Even if the other mantissa bits are all zero, the implied one bit always gives us the value one[4]. Of course, even if we had an almost infinite number of one bits after the binary point, they still would not add up to two. This is why the mantissa can represent values in the range one to just under two.

Although there are an infinite number of values between one and two, we can only represent eight million of them because we use a 23 bit mantissa (the 24th bit is always one). This is the reason for inaccuracy

---

2. There were some minor changes to the way certain degenerate operations were handled, but the bit representation remained essentially unchanged.

3. The binary point is the same thing as the decimal point except it appears in binary numbers rather than decimal numbers.

4. Actually, this isn't necessarily true. The IEEE floating point format supports *denormalized* values where the H.O. bit is not zero. However, we will ignore denormalized values in our discussion.

in floating point arithmetic – we are limited to 23 bits of precision in computations involving single precision floating point values.

The mantissa uses a *one's complement* format rather than two's complement. This means that the 24 bit value of the mantissa is simply an unsigned binary number and the sign bit determines whether that value is positive or negative. One's complement numbers have the unusual property that there are two representations for zero (with the sign bit set or clear). Generally, this is important only to the person designing the floating point software or hardware system. We will assume that the value zero always has the sign bit clear.

To represent values outside the range 1.0 to just under 2.0, the exponent portion of the floating point format comes into play. The floating point format raises two to the power specified by the exponent and then multiplies the mantissa by this value. The exponent is eight bits and is stored in an *excess-127* format. In excess-127 format, the exponent 2^0 is represented by the value 127 ($7f). Therefore, to convert an exponent to excess-127 format simply add 127 to the exponent value. The use of excess-127 format makes it easier to compare floating point values. The single precision floating point format takes the form shown in Figure 4.2.
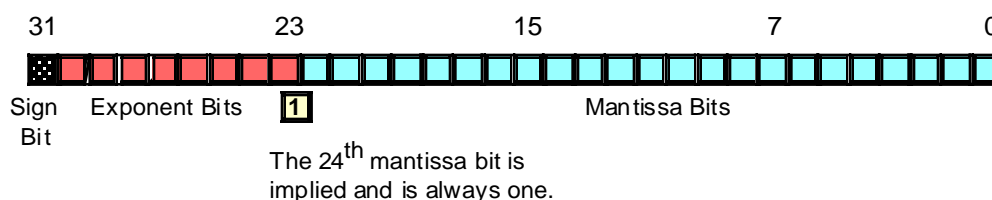
Figure 4.2   Single Precision (32-bit) Floating Point Format

With a 24 bit mantissa, you will get approximately 6-$^1/_2$ digits of precision (one half digit of precision means that the first six digits can all be in the range 0..9 but the seventh digit can only be in the range 0..x where x<9 and is generally close to five). With an eight bit excess-127 exponent, the dynamic range of single precision floating point numbers is approximately $2^{\pm128}$ or about $10^{\pm38}$.

Although single precision floating point numbers are perfectly suitable for many applications, the dynamic range is somewhat limited for many scientific applications and the very limited precision is unsuitable for many financial, scientific, and other applications. Furthermore, in long chains of computations, the limited precision of the single precision format may introduce serious error.

The double precision format helps overcome the problems of single precision floating point. Using twice the space, the double precision format has an 11-bit excess-1023 exponent and a 53 bit mantissa (with an implied H.O. bit of one) plus a sign bit. This provides a dynamic range of about $10^{\pm308}$and 14-$^1/_2$ digits of precision, sufficient for most applications. Double precision floating point values take the form shown in Figure 4.3.

Figure 4.3   64-Bit Double Precision Floating Point Format

In order to help ensure accuracy during long chains of computations involving double precision floating point numbers, Intel designed the extended precision format. The extended precision format uses 80 bits. Twelve of the additional 16 bits are appended to the mantissa, four of the additional bits are appended to the

end of the exponent. Unlike the single and double precision values, the extended precision format's mantissa does not have an implied H.O. bit which is always one. Therefore, the extended precision format provides a 64 bit mantissa, a 15 bit excess-16383 exponent, and a one bit sign. The format for the extended precision floating point value is shown in Figure 4.4:
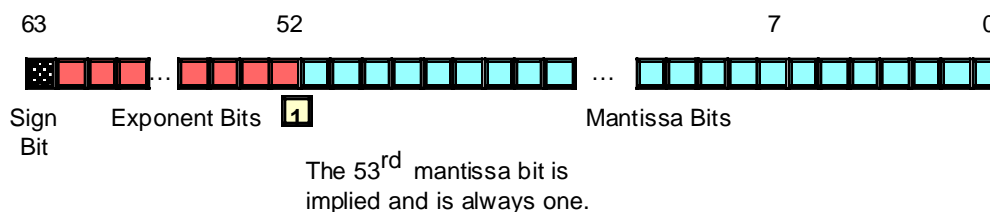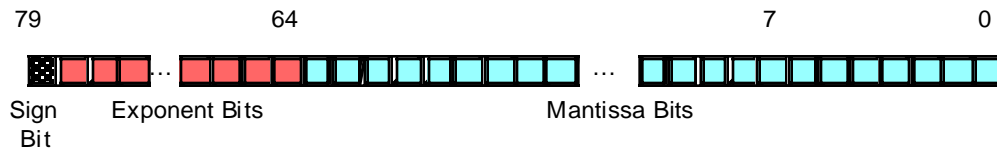

Figure 4.4   80-bit Extended Precision Floating Point Format

On the FPUs all computations are done using the extended precision form. Whenever you load a single or double precision value, the FPU automatically converts it to an extended precision value. Likewise, when you store a single or double precision value to memory, the FPU automatically rounds the value down to the appropriate size before storing it. By always working with the extended precision format, Intel guarantees a large number of guard bits are present to ensure the accuracy of your computations. Some texts erroneously claim that you should never use the extended precision format in your own programs, because Intel only guarantees accurate computations when using the single or double precision formats. This is foolish. By performing all computations using 80 bits, Intel helps ensure (but not guarantee) that you will get full 32 or 64 bit accuracy in your computations. Since the FPUs do not provide a large number of guard bits in 80 bit computations, some error will inevitably creep into the L.O. bits of an extended precision computation. However, if your computation is correct to 64 bits, the 80 bit computation will always provide *at least* 64 accurate bits. Most of the time you will get even more. While you cannot assume that you get an accurate 80 bit computation, you can usually do better than 64 when using the extended precision format.

To maintain maximum precision during computation, most computations use *normalized* values. A normalized floating point value is one whose H.O. mantissa bit contains one. Almost any non-normalized value can be normalized by shifting the mantissa bits to the left and decrementing the exponent until a one appears in the H.O. bit of the mantissa. Remember, the exponent is a binary exponent. Each time you increment the exponent, you multiply the floating point value by two. Likewise, whenever you decrement the exponent, you divide the floating point value by two. By the same token, shifting the mantissa to the left one bit position multiplies the floating point value by two; likewise, shifting the mantissa to the right divides the floating point value by two. Therefore, shifting the mantissa to the left one position *and* decrementing the exponent does not change the value of the floating point number at all.

Keeping floating point numbers normalized is beneficial because it maintains the maximum number of bits of precision for a computation. If the H.O. bits of the mantissa are all zero, the mantissa has that many fewer bits of precision available for computation. Therefore, a floating point computation will be more accurate if it involves only normalized values.

There are two important cases where a floating point number cannot be normalized. The value 0.0 is a special case. Obviously it cannot be normalized because the floating point representation for zero has no one bits in the mantissa. This, however, is not a problem since we can exactly represent the value zero with only a single bit.

The second case is when we have some H.O. bits in the mantissa which are zero but the biased exponent is also zero (and we cannot decrement it to normalize the mantissa). Rather than disallow certain small values, whose H.O. mantissa bits and biased exponent are zero (the most negative exponent possible), the IEEE standard allows special *denormalized* values to represent these smaller values[5]. Although the use of denormalized values allows IEEE floating point computations to produce better results than if underflow occurred, keep in mind that denormalized values offer less bits of precision.

_____

5. The alternative would be to underflow the values to zero.

Since the FPU always converts single and double precision values to extended precision, extended precision arithmetic is actually *faster* than single or double precision. Therefore, the expected performance benefit of using the smaller formats is not present on these chips. However, when designing the Pentium/586 CPU, Intel redesigned the built-in floating point unit to better compete with RISC chips. Most RISC chips support a native 64 bit double precision format which is faster than Intel's extended precision format. Therefore, Intel provided native 64 bit operations on the Pentium to better compete against the RISC chips. Therefore, the double precision format is the fastest on the Pentium and later chips.

4.2.2 HLA Support for Floating Point Values

HLA provides several data types and library routines to support the use of floating point data in your assembly language programs. These include built-in types to declare floating point variables as well as routines that provide floating point input, output, and conversion.

Perhaps the best place to start when discussing HLA's floating point facilities is with a description of floating point literal constants. HLA floating point constants allow the following syntax:

- An optional "+" or "-" symbol, denoting the sign of the mantissa (if this is not present, HLA assumes that the mantissa is positive),
- Followed by one or more decimal digits,
- Optionally followed by a decimal point and one or more decimal digits,
- Optionally followed by an "e" or "E", optionally followed by a sign ("+" or "-") and one or more decimal digits.

Note: the decimal point or the "e"/"E" must be present in order to differentiate this value from an integer or unsigned literal constant. Here are some examples of legal literal floating point constants:

```
1.234    3.75e2    -1.0   1.1e-1    1e+4    0.1    -123.456e+789    +25e0
```

Notice that a floating point literal constant cannot begin with a decimal point; it must begin with a decimal digit so you must use "0.1" to represent ".1" in your programs.

HLA also allows you to place an underscore character ("_") between any two consecutive decimal digits in a floating point literal constant. You may use the underscore character in place of a comma (or other language-specific separator character) to help make your large floating point numbers easier to read. Here are some examples:

```
       1_234_837.25      1_000.00      789_934.99       9_999.99
```

To declare a floating point variable you use the *real32*, *real64*, or *real80* data types. Like their integer and unsigned brethren, the number at the end of these data type declarations specifies the number of bits used for each type's binary representation. Therefore, you use *real32* to declare single precision real values, *real64* to declare double precision floating point values, and *real80* to declare extended precision floating point values. Other than the fact that you use these types to declare floating point variables rather than integers, their use is nearly identical to that for *int8, int16, int32,* etc. The following examples demonstrate these declarations and their syntax:

```
static

        fltVar1:   real32;
        fltVar1a:  real32 := 2.7;
        pi:        real32 := 3.14159;
        DblVar:    real64;
        DblVar2:   real64 := 1.23456789e+10;
        XPVar:     real80;
        XPVar2:    real80 := -1.0e-104;
```

To output a floating point variable in ASCII form, you would use one of the *stdout.putr32, std-out.putr64,* or *stdout.putr80* routines. These procedures display a number in decimal notation, that is, a string of digits, an optional decimal point and a closing string of digits. Other than their names, these three routines use exactly the same calling sequence. Here are the calls and parameters for each of these routines:

```
stdout.putr80( r:real80; width:uns32; decpts:uns32 );
stdout.putr64( r:real64; width:uns32; decpts:uns32 );
stdout.putr32( r:real32; width:uns32; decpts:uns32 );
```

The first parameter to these procedures is the floating point value you wish to print. The size of this parameter must match the procedure's name (e.g., the *r* parameter must be an 80-bit extended precision floating point variable when calling the *stdout.putr80* routine). The second parameter specifies the field width for the output text; this is the number of print positions the number will require when the procedure displays it. Note that this width must include print positions for the sign of the number and the decimal point. The third parameter specifies the number of print positions after the decimal point. For example,

```
                    stdout.putr32( pi, 10, 4 );
```

displays the value

```
                    _ _ _ _ 3.1416
```

(the underscores represent leading spaces in this example).

Of course, if the number is very large or very small, you will want to use scientific notation rather than decimal notation for your floating point numeric output. The HLA Standard Library *stdout.pute32, std-out.pute64*, and *stdout.pute80* routines provide this facility. These routines use the following procedure prototypes:

```
stdout.pute80( r:real80; width:uns32 );
stdout.pute64( r:real64; width:uns32 );
stdout.pute32( r:real32; width:uns32 );
```

Unlike the decimal output routines, these scientific notation output routines do not require a third parameter specifying the number of digits after the decimal point to display. The width parameter, indirectly, specifies this value since all but one of the mantissa digits always appears to the right of the decimal point. These routines output their values in decimal notation, similar to the following:

```
1.23456789e+10     -1.0e-104      1e+2
```

You can also output floating point values using the HLA Standard Library *stdout.put* routine. If you specify the name of a floating point variable in the *stdout.put* parameter list, the *stdout.put* code will output the value using scientific notation. The actual field width varies depending on the size of the floating point variable (the *stdout.put* routine attempts to output as many significant digits as possible, in this case). Example:

```
            stdout.put( "XPVar2 = ", XPVar2 );
```

If you specify a field width specification, by using a colon followed by a signed integer value, then the *stdout.put* routine will use the appropriate *stdout.puteXX* routine to display the value. That is, the number will still appear in scientific notation, but you get to control the field width of the output value. Like the field width for integer and unsigned values, a positive field width right justifies the number in the specified field, a negative number left justifies the value. Here is an example that prints the *XPVar2* variable using ten print positions:

```
            stdout.put( "XPVar2 = ", XPVar2:10 );
```

If you wish to use *stdout.put* to print a floating point value in decimal notation, you need to use the following syntax:

```
            Variable_Name : Width : DecPts
```

Note that the *DecPts* field must be a non-negative integer value.

When *stdout.put* contains a parameter of this form, it calls the corresponding *stdout.putrXX* routine to display the specified floating point value. As an example, consider the following call:

```
stdout.put( "Pi = ", pi:5:3 );
```

The corresponding output is

```
3.142
```

The HLA Standard Library provides several other useful routines you can use when outputting floating point values. Consult the HLA Standard Library reference manual for more information on these routines.

The HLA Standard Library provides several routines to let you display floating point values in a wide variety of formats. In contrast, the HLA Standard Library only provides two routines to support floating point input: *stdin.getf()* and *stdin.get()*. The *stdin.getf()* routine requires the use of the 80x86 FPU stack, a hardware component that this chapter is not going to cover. Therefore, this chapter will defer the discussion of the *stdin.getf()* routine until the chapter on arithmetic, later in this text. Since the *stdin.get()* routine provides all the capabilities of the *stdin.getf()* routine, this deference will not prove to be a problem.

You've already seen the syntax for the *stdin.get()* routine; its parameter list simply contains a list of variable names. *Stdin.get()* reads appropriate values for the user for each of the variables appearing in the parameter list. If you specify the name of a floating point variable, the *stdin.get()* routine automatically reads a floating point value from the user and stores the result into the specified variable. The following example demonstrates the use of this routine:

```
stdout.put( "Input a double precision floating point value: " );
stdin.get( DblVar );
```

> **Warning**: This section has discussed how you would declare floating point variables and how you would input and output them. It did not discuss arithmetic. Floating point arithmetic is different than integer arithmetic; you cannot use the 80x86 ADD and SUB instructions to operate on floating point values. Floating point arithmetic will be the subject of a later chapter in this text.

4.3   Binary Coded Decimal (BCD) Representation

Although the integer and floating point formats cover most of the numeric needs of an average program, there are some special cases where other numeric representations are convenient. In this section we'll discuss the Binary Coded Decimal (BCD) format since the 80x86 CPU provides a small amount of hardware support for this data representation.

BCD values are a sequence of nibbles with each nibble representing a value in the range zero through nine. Of course you can represent values in the range 0..15 using a nibble; the BCD format, however, uses only 10 of the possible 16 different values for each nibble.

Each nibble in a BCD value represents a single decimal digit. Therefore, with a single byte (i.e., two digits) we can represent values containing two decimal digits, or values in the range 0..99. With a word, we can represent values having four decimal digits, or values in the range 0..9999. Likewise, with a double word we can represent values with up to eight decimal digits (since there are eight nibbles in a double word value).

7 6 5 4 3 2 1 0

H.O. Nibble
(H.O. Digit)

L.O. Nibble
(L.O. Digit)

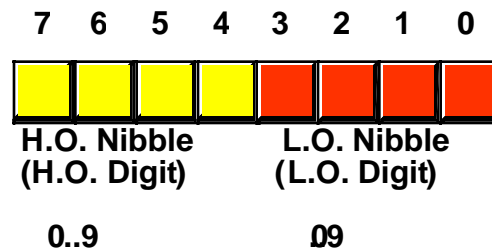0..9   0..9

Figure 4.5   BCD Data Representation in Memory

As you can see, BCD storage isn't particularly memory efficient. For example, an eight-bit BCD variable can represent values in the range 0..99 while that same eight bits, when holding a binary value, can represent values in the range 0..255. Likewise, a 16-bit binary value can represent values in the range 0..65535 while a 16-bit BCD value can only represent about $^1/_6$ of those values (0..9999). Inefficient storage isn't the only problem. BCD calculations tend to be slower than binary calculations.

At this point, you're probably wondering why anyone would ever use the BCD format. The BCD format does have two saving graces: it's very easy to convert BCD values between the internal numeric representation and their string representation; also, its very easy to encode multi-digit decimal values in hardware (e.g., using a "thumb wheel" or dial) using BCD than it is using binary. For these two reasons, you're likely to see people using BCD in embedded systems (e.g., toaster ovens and alarm clocks) but rarely in general purpose computer software.

A few decades ago people mistakenly thought that calculations involving BCD (or just 'decimal') arithmetic was more accurate than binary calculations. Therefore, they would often perform 'important' calculations, like those involving dollars and cents (or other monetary units) using decimal-based arithmetic. While it is true that certain calculations can produce more accurate results in BCD, this statement is not true in general. Indeed, for most calculations (even those involving fixed point decimal arithmetic), the binary representation is more accurate. For this reason, most modern computer programs represent all values in a binary form. For example, the Intel x86 floating point unit (FPU) supports a pair of instructions for loading and storing BCD values. Internally, however, the FPU converts these BCD values to binary and performs all calculations in binary. It only uses BCD as an external data format (external to the FPU, that is). This generally produces more accurate results and requires far less silicon than having a separate coprocessor that supports decimal arithmetic.

This text will take up the subject of BCD arithmetic in a later chapter. Until then, you can safely ignore BCD unless you find yourself converting a COBOL program to assembly language (which is quite unlikely).

4.4   Characters

Perhaps the most important data type on a personal computer is the character data type. The term "character" refers to a human or machine readable symbol that is typically a non-numeric entity. In general, the term "character" refers to any symbol that you can normally type on a keyboard (including some symbols that may require multiple key presses to produce) or display on a video display. Many beginners often confuse the terms "character" and "alphabetic character." These terms are not the same. Punctuation symbols, numeric digits, spaces, tabs, carriage returns (enter), other control characters, and other special symbols are also characters. When this text uses the term "character" it refers to any of these characters, not just the alphabetic characters. When this text refers to alphabetic characters, it will use phrases like "alphabetic characters," "upper case characters," or "lower case characters."[6].

Another common problem beginners have when they first encounter the character data type is differentiating between numeric characters and numbers. The character '1' is distinct and different from the value one. The computer (generally) uses two different internal, binary, representations for numeric characters ('0', '1', ..., '9') versus the numeric values zero through nine. You must take care not to confuse the two.

Most computer systems use a one or two byte sequence to encode the various characters in binary form. Windows and Linux certainly fall into this category, using either the ASCII or Unicode encodings for characters. This section will discuss the ASCII character set and the character declaration facilities that HLA provides.

4.4.1 The ASCII Character Encoding

The ASCII (American Standard Code for Information Interchange) Character set maps 128 textual characters to the unsigned integer values 0..127 ($0..$7F). Internally, of course, the computer represents everything using binary numbers; so it should come as no surprise that the computer also uses binary values to represent non-numeric entities such as characters. Although the exact mapping of characters to numeric values is arbitrary and unimportant, it is important to use a standardized code for this mapping since you will need to communicate with other programs and peripheral devices and you need to talk the same "language" as these other programs and devices. This is where the ASCII code comes into play; it is a standardized code that nearly everyone has agreed upon. Therefore, if you use the ASCII code 65 to represent the character "A" then you know that some peripheral device (such as a printer) will correctly interpret this value as the character "A" whenever you transmit data to that device.

You should not get the impression that ASCII is the only character set in use on computer systems. IBM uses the EBCDIC character set family on many of its mainframe computer systems. Another common character set in use is the Unicode character set. Unicode is an extension to the ASCII character set that uses 16 bits rather than seven to represent characters. This allows the use of 65,536 different characters in the character set, allowing the inclusion of most symbols in the world's different languages into a single unified character set.

Since the ASCII character set provides only 128 different characters and a byte can represent 256 different values, an interesting question arises: "what do we do with the values 128..255 that one could store into a byte value when working with character data?" One answer is to ignore those extra values. That will be the primary approach of this text. Another possibility is to extend the ASCII character set and add an additional 128 characters to the character set. Of course, this would tend to defeat the whole purpose of having a standardized character set unless you could get everyone to agree upon the extensions. That is a difficult task.

When IBM first created their IBM-PC, they defined these extra 128 character codes to contain various non-English alphabetic characters, some line drawing graphics characters, some mathematical symbols, and several other special characters. Since IBM's PC was the foundation for what we typically call a PC today, that character set has become a pseudo-standard on all IBM-PC compatible machines. Even on modern machines, which are not IBM-PC compatible and cannot run early PC software, the IBM extended character set still survives. Note, however, that this PC character set (an extension of the ASCII character set) is not universal. Most printers will not print the extended characters when using native fonts and many programs (particularly in non-English countries) do not use those characters for the upper 128 codes in an eight-bit value. For these reasons, this text will generally stick to the standard 128 character ASCII character set. However, a few examples and programs in this text will use the IBM PC extended character set, particularly the line drawing graphic characters (see Appendix B).

Should you need to exchange data with other machines which are not PC-compatible, you have only two alternatives: stick to standard ASCII or ensure that the target machine supports the extended IBM-PC character set. Some machines, like the Apple Macintosh, do not provide native support for the extended IBM-PC character set; however you may obtain a PC font which lets you display the extended character set.

---

6. Upper and lower case characters are always alphabetic characters within this text.

Other machines have similar capabilities. However, the 128 characters in the standard ASCII character set are the only ones you should count on transferring from system to system.

Despite the fact that it is a "standard", simply encoding your data using standard ASCII characters does not guarantee compatibility across systems. While it's true that an "A" on one machine is most likely an "A" on another machine, there is very little standardization across machines with respect to the use of the control characters. Indeed, of the 32 control codes plus delete, there are only four control codes commonly supported – backspace (BS), tab, carriage return (CR), and line feed (LF). Worse still, different machines often use these control codes in different ways. End of line is a particularly troublesome example. Windows, MS-DOS, CP/M, and other systems mark end of line by the two-character sequence CR/LF. Apple Macintosh, and many other systems mark the end of line by a single CR character. Linux, BeOS, and other UNIX systems mark the end of a line with a single LF character. Needless to say, attempting to exchange simple text files between such systems can be an experience in frustration. Even if you use standard ASCII characters in all your files on these systems, you will still need to convert the data when exchanging files between them. Fortunately, such conversions are rather simple.

Despite some major shortcomings, ASCII data is *the* standard for data interchange across computer systems and programs. Most programs can accept ASCII data; likewise most programs can produce ASCII data. Since you will be dealing with ASCII characters in assembly language, it would be wise to study the layout of the character set and memorize a few key ASCII codes (e.g., "0", "A", "a", etc.).

The ASCII character set (excluding the extended characters defined by IBM) is divided into four groups of 32 characters. The first 32 characters, ASCII codes 0 through $1F (31), form a special set of non-printing characters called the control characters. We call them control characters because they perform various printer/display control operations rather than displaying symbols. Examples include *carriage return*, which positions the cursor to the left side of the current line of characters[7], line feed (which moves the cursor down one line on the output device), and back space (which moves the cursor back one position to the left). Unfortunately, different control characters perform different operations on different output devices. There is very little standardization among output devices. To find out exactly how a control character affects a particular device, you will need to consult its manual.

The second group of 32 ASCII character codes comprise various punctuation symbols, special characters, and the numeric digits. The most notable characters in this group include the space character (ASCII code $20) and the numeric digits (ASCII codes $30..$39). Note that the numeric digits differ from their numeric values only in the H.O. nibble. By subtracting $30 from the ASCII code for any particular digit you can obtain the numeric equivalent of that digit.

The third group of 32 ASCII characters contains the upper case alphabetic characters. The ASCII codes for the characters "A".."Z" lie in the range $41..$5A (65..90). Since there are only 26 different alphabetic characters, the remaining six codes hold various special symbols.

The fourth, and final, group of 32 ASCII character codes represent the lower case alphabetic symbols, five additional special symbols, and another control character (delete). Note that the lower case character symbols use the ASCII codes $61..$7A. If you convert the codes for the upper and lower case characters to binary, you will notice that the upper case symbols differ from their lower case equivalents in exactly one bit position. For example, consider the character code for "E" and "e" in the following figure:

_____

7. Historically, carriage return refers to the *paper carriage* used on typewriters. A carriage return consisted of physically moving the carriage all the way to the right so that the next character typed would appear at the left hand side of the paper.
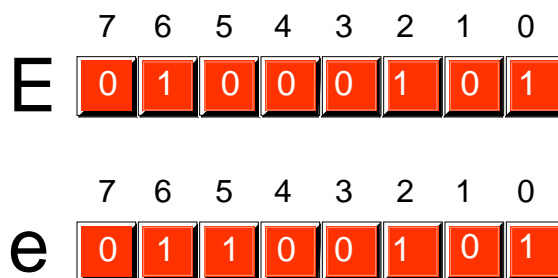
Figure 4.6   ASCII Codes for "E" and "e"

The only place these two codes differ is in bit five. Upper case characters always contain a zero in bit five; lower case alphabetic characters always contain a one in bit five. You can use this fact to quickly convert between upper and lower case. If you have an upper case character you can force it to lower case by setting bit five to one. If you have a lower case character and you wish to force it to upper case, you can do so by setting bit five to zero. You can toggle an alphabetic character between upper and lower case by simply inverting bit five.

Indeed, bits five and six determine which of the four groups in the ASCII character set you're in:

Table 9: ASCII Groups

Bit 6	Bit 5	Group
0	0	Control Characters
0	1	Digits & Punctuation
1	0	Upper Case & Special
1	1	Lower Case & Special

So you could, for instance, convert any upper or lower case (or corresponding special) character to its equivalent control character by setting bits five and six to zero.

Consider, for a moment, the ASCII codes of the numeric digit characters:

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
"0"	48	$30
"1"	49	$31
"2"	50	$32
"3"	51	$33

Table 10: ASCII Codes for Numeric Digits

Character	Decimal	Hexadecimal
"4"	52	$34
"5"	53	$35
"6"	54	$36
"7"	55	$37
"8"	56	$38
"9"	57	$39

The decimal representations of these ASCII codes are not very enlightening. However, the hexadecimal representation of these ASCII codes reveals something very important – the L.O. nibble of the ASCII code is the binary equivalent of the represented number. By stripping away (i.e., setting to zero) the H.O. nibble of a numeric character, you can convert that character code to the corresponding binary representation. Conversely, you can convert a binary value in the range 0..9 to its ASCII character representation by simply setting the H.O. nibble to three. Note that you can use the logical-AND operation to force the H.O. bits to zero; likewise, you can use the logical-OR operation to force the H.O. bits to %0011 (three).

Note that you *cannot* convert a string of numeric characters to their equivalent binary representation by simply stripping the H.O. nibble from each digit in the string. Converting 123 ($31 $32 $33) in this fashion yields three bytes: $010203, not the correct value which is $7B. Converting a string of digits to an integer requires more sophistication than this; the conversion above works only for single digits.

4.4.2 HLA Support for ASCII Characters

Although you could easily store character values in *byte* variables and use the corresponding numeric equivalent ASCII code when using a character literal in your program, such agony is unnecessary - HLA provides good support for character variables and literals in your assembly language programs.

Character literal constants in HLA take one of two forms: a single character surrounded by apostrophes or a pound symbol ("#") followed by a numeric constant in the range 0..127 specifying the ASCII code of the character. Here are some examples:

```
        'A'        #65        #$41        #%0100_0001
```

Note that these examples all represent the same character ('A') since the ASCII code of 'A' is 65.

With a single exception, only a single character may appear between the apostrophes in a literal character constant. That single exception is the apostrophe character itself. If you wish to create an apostrophe literal constant, place four apostrophes in a row (i.e., double up the apostrophe inside the surrounding apostrophes), i.e.,

```
                                    ''''
```

The pound sign operator ("#") must precede a legal HLA numeric constant (either decimal, hexadecimal or binary as the examples above indicate). In particular, the pound sign is not a generic character conversion function; it cannot precede registers or variable names, only constants. As a general rule, you should always use the apostrophe form of the character literal constant for graphic characters (that is, those that are printable or displayable). Use the pound sign form for control characters (that are invisible, or do funny things when you print them) or for extended ASCII characters that may not display or print properly within your source code.

Notice the difference between a character literal constant and a string literal constant in your programs. Strings are sequences of zero or more characters surrounded by quotation marks, characters are surrounded by apostrophes. It is especially important to realize that

<p style="text-align:center">'A' ≠ "A"</p>

The character constant 'A' and the string containing the single character "A" have two completely different internal representations. If you attempt to use a string containing a single character where HLA expects a character constant, HLA will report an error. Strings and string constants will be the subject of a later chapter.

To declare a character variable in an HLA program, you use the *char* data type. The following declaration, for example, demonstrates how to declare a variable named *UserInput*:

```
static
    UserInput:      char;
```

This declaration reserves one byte of storage that you could use to store any character value (including eight-bit extended ASCII characters). You can also initialize character variables as the following example demonstrates:

```
static

    TheCharA:       char := 'A';
    ExtendedChar    char := #128;
```

Since character variables are eight-bit objects, you can manipulate them using eight-bit registers. You can move character variables into eight-bit registers and you can store the value of an eight-bit register into a character variable.

The HLA Standard Library provides a handful of routines that you can use for character I/O and manipulation; these include *stdout.putc*, *stdout.putcSize*, *stdout.put, stdin.getc,* and *stdin.get*.

The *stdout.putc* routine uses the following calling sequence:

```
stdout.putc( chvar );
```

This procedure outputs the single character parameter passed to it as a character to the standard output device. The parameter may be any *char* constant or variable, or a *byte* variable or register[8].

The *stdout.putcSize* routine provides output width control when displaying character variables. The calling sequence for this procedure is

```
stdout.putcSize( charvar, widthInt32, fillchar );
```

This routine prints the specified character (parameter *c*) using at least *width* print positions[9]. If the absolute value of *width* is greater than one, then *stdout.putcSize* prints the *fill* character as padding. If the value of *width* is positive, then *stdout.putcSize* prints the character right justified in the print field; if *width* is negative, then *stdout.putcSize* prints the character left justified in the print field. Since character output is usually left justified in a field, the *width* value will normally be negative for this call. The space character is the most common *fill* value.

You can also print character values using the generic *stdout.put* routine. If a character variable appears in the *stdout.put* parameter list, then *stdout.put* will automatically print it as a character value, e.g.,

```
stdout.put( "Character c = '", c, "'", nl );
```

You can read characters from the standard input using the *stdin.getc* and *stdin.get* routines. The *stdin.getc* routine does not have any parameters. It reads a single character from the standard input buffer and returns this character in the AL register. You may then store the character value away or otherwise

---

8. If you specify a byte variable or a byte-sized register as the parameter, the *stdout.putc* routine will output the character whose ASCII code appears in the variable or register.

9. The only time *stdout.putcSize* uses more print positions than you specify is when you specify zero as the width; then this routine uses exactly one print position.

manipulate the character in the AL register. The following program reads a single character from the user, converts it to upper case if it is a lower case character, and then displays the character:

```
program charInputDemo;
#include( "stdlib.hhf" );
static
    c:char;

begin charInputDemo;

    stdout.put( "Enter a character: " );
    stdin.getc();
    if( al >= 'a' ) then

        if( al <= 'z' ) then

            and( $5f, al );

        endif;

    endif;
    stdout.put
    (
        "The character you entered, possibly ", nl,
        "converted to upper case, was '"
    );
    stdout.putc( al );
    stdout.put( "'", nl );

end charInputDemo;
```

Program 4.1   Character Input Sample

You can also use the generic *stdin.get* routine to read character variables from the user. If a *stdin.get* parameter is a character variable, then the *stdin.get* routine will read a character from the user and store the character value into the specified variable. Here is the program above rewritten to use the *stdin.get* routine:

```
program charInputDemo2;
#include( "stdlib.hhf" );
static
    c:char;

begin charInputDemo2;

    stdout.put( "Enter a character: " );
    stdin.get(c);
    if( c >= 'a' ) then

        if( c <= 'z' ) then

            and( $5f, c );

        endif;
```

                           Beta Draft - Do not distribute

```
        endif;
        stdout.put
        (
            "The character you entered, possibly ", nl,
            "converted to upper case, was '",
            c,
            "'", nl
        );

    end charInputDemo2;
```

Program 4.2   Stdin.get Character Input Sample

As you may recall from the last chapter, the HLA Standard Library buffers its input. Whenever you read a character from the standard input using *stdin.getc* or *stdin.get*, the library routines read the next available character from the buffer; if the buffer is empty, then the program reads a new line of text from the user and returns the first character from that line. If you want to guarantee that the program reads a new line of text from the user when you read a character variable, you should call the *stdin.flushInput* routine before attempting to read the character. This will flush the current input buffer and force the input of a new line of text on the next input (which should be your *stdin.getc* or *stdin.get* call).

The end of line is problematic. Different operating systems handle the end of line differently on output versus input. From the console device, pressing the ENTER key signals the end of a line; however, when reading data from a file you get an end of line sequence which is typically a line feed or a carriage return/line feed pair. To help solve this problem, HLA's Standard Library provides an "end of line" function. This procedure returns true (one) in the AL register if all the current input characters have been exhausted, it returns false (zero) otherwise. The following sample program demonstrates the use of the *stdin.eoln* function.

```
program eolnDemo2;
#include( "stdlib.hhf" );
begin eolnDemo2;

    stdout.put( "Enter a short line of text: " );
    stdin.flushInput();
    repeat

        stdin.getc();
        stdout.putc( al );
        stdout.put( "=$", al, nl );

    until( stdin.eoln() );

end eolnDemo2;
```

Program 4.3   Testing for End of Line Using Stdin.eoln

The HLA language and the HLA Standard Library provide many other procedures and additional support for character objects. Later chapters in this textbook, as well as the HLA reference documentation, describe how to use these features.

4.4.3 The ASCII Character Set

The following table lists the binary, hexadecimal, and decimal representations for each of the 128 ASCII character codes.

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0000_0000	00	0	NULL
0000_0001	01	1	ctrl A
0000_0010	02	2	ctrl B
0000_0011	03	3	ctrl C
0000_0100	04	4	ctrl D
0000_0101	05	5	ctrl E
0000_0110	06	6	ctrl F
0000_0111	07	7	bell
0000_1000	08	8	backspace
0000_1001	09	9	tab
0000_1010	0A	10	line feed
0000_1011	0B	11	ctrl K
0000_1100	0C	12	form feed
0000_1101	0D	13	return
0000_1110	0E	14	ctrl N
0000_1111	0F	15	ctrl O
0001_0000	10	16	ctrl P
0001_0001	11	17	ctrl Q
0001_0010	12	18	ctrl R
0001_0011	13	19	ctrl S
0001_0100	14	20	ctrl T
0001_0101	15	21	ctrl U
0001_0110	16	22	ctrl V
0001_0111	17	23	ctrl W

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0001_1000	18	24	ctrl X
0001_1001	19	25	ctrl Y
0001_1010	1A	26	ctrl Z
0001_1011	1B	27	ctrl [
0001_1100	1C	28	ctrl \
0001_1101	1D	29	Esc
0001_1110	1E	30	ctrl ^
0001_1111	1F	31	ctrl _
0010_0000	20	32	space
0010_0001	21	33	!
0010_0010	22	34	"
0010_0011	23	35	#
0010_0100	24	36	$
0010_0101	25	37	%
0010_0110	26	38	&
0010_0111	27	39	'
0010_1000	28	40	(
0010_1001	29	41	)
0010_1010	2A	42	*
0010_1011	2B	43	+
0010_1100	2C	44	,
0010_1101	2D	45	-
0010_1110	2E	46	.
0010_1111	2F	47	/
0011_0000	30	48	0
0011_0001	31	49	1
0011_0010	32	50	2
0011_0011	33	51	3

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0011_0100	34	52	4
0011_0101	35	53	5
0011_0110	36	54	6
0011_0111	37	55	7
0011_1000	38	56	8
0011_1001	39	57	9
0011_1010	3A	58	:
0011_1011	3B	59	;
0011_1100	3C	60	<
0011_1101	3D	61	=
0011_1110	3E	62	>
0011_1111	3F	63	?
0100_0000	40	64	@
0100_0001	41	65	A
0100_0010	42	66	B
0100_0011	43	67	C
0100_0100	44	68	D
0100_0101	45	69	E
0100_0110	46	70	F
0100_0111	47	71	G
0100_1000	48	72	H
0100_1001	49	73	I
0100_1010	4A	74	J
0100_1011	4B	75	K
0100_1100	4C	76	L
0100_1101	4D	77	M
0100_1110	4E	78	N
0100_1111	4F	79	O

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0101_0000	50	80	P
0101_0001	51	81	Q
0101_0010	52	82	R
0101_0011	53	83	S
0101_0100	54	84	T
0101_0101	55	85	U
0101_0110	56	86	V
0101_0111	57	87	W
0101_1000	58	88	X
0101_1001	59	89	Y
0101_1010	5A	90	Z
0101_1011	5B	91	[
0101_1100	5C	92	\
0101_1101	5D	93	]
0101_1110	5E	94	^
0101_1111	5F	95	_
0110_0000	60	96	`
0110_0001	61	97	a
0110_0010	62	98	b
0110_0011	63	99	c
0110_0100	64	100	d
0110_0101	65	101	e
0110_0110	66	102	f
0110_0111	67	103	g
0110_1000	68	104	h
0110_1001	69	105	i
0110_1010	6A	106	j
0110_1011	6B	107	k

Table 11: ASCII Character Set

Binary	Hex	Decimal	Character
0110_1100	6C	108	l
0110_1101	6D	109	m
0110_1110	6E	110	n
0110_1111	6F	111	o
0111_0000	70	112	p
0111_0001	71	113	q
0111_0010	72	114	r
0111_0011	73	115	s
0111_0100	74	116	t
0111_0101	75	117	u
0111_0110	76	118	v
0111_0111	77	119	w
0111_1000	78	120	x
0111_1001	79	121	y
0111_1010	7A	122	z
0111_1011	7B	123	{
0111_1100	7C	124	|
0111_1101	7D	125	}
0111_1110	7E	126	~
0111_1111	7F	127	

4.5   The UNICODE Character Set

Although the ASCII character set is, unquestionably, the most popular character representation on computers, it is certainly not the only format around. For example, IBM uses the EBCDIC code on many of its mainframe and minicomputer lines. Since EBCDIC appears mainly on IBM's big iron and you'll rarely encounter it on personal computer systems, we will not consider that character set in this text. Another character representation that is becoming popular on small computer systems (and large ones, for that matter) is the Unicode character set. Unicode overcomes two of ASCII's greatest limitations: the limited character space (i.e., a maximum of 128/256 characters in an eight-bit byte) and the lack of international (beyond the USA) characters.

Unicode uses a 16-bit word to represent a single character. Therefore, Unicode supports up to 65,536 different character codes. This is obviously a huge advance over the 256 possible codes we can represent with an eight-bit byte. Unicode is upwards compatible from ASCII. Specifically, if the H.O. 17 bits of a

Unicode character contain zero, then the L.O. seven bits represent the same character as the ASCII character with the same character code. If the H.O. 17 bits contain some non-zero value, then the character represents some other value. If you're wondering why so many different character codes are necessary, simply note that certain Asian character sets contain 4096 characters (at least, their Unicode subset).

This text will stick to the ASCII character set except for a few brief mentions of Unicode here and there. Eventually, this text may have to eliminate the discussion of ASCII in favor of Unicode since many new operating systems are using Unicode internally (and convert to ASCII as necessary). Unfortunately, many string algorithms are not as conveniently written for Unicode as for ASCII (especially character set functions) so we'll stick with ASCII in this text as long as possible.

4.6   Other Data Representations

Of course, we can represent many different objects other than numbers and characters in a computer system. The following subsections provide a brief description of the different real-world data types you might encounter.

4.6.1 Representing Colors on a Video Display

As you're probably aware, color images on a computer display are made up of a series of dots known as *pixels* (which is short for "picture elements."). Different display modes (depending on the capability of the display adapter) use different data representations for each of these pixels. The one thing in common between these data types is that they control the mixture of the three additive primary colors (red, green, and blue) to form a specific color on the display. The question, of course, is how much of each of these colors do they mix together?

Color depth is the term video card manufacturers use to describe how much red, green, and blue they mix together for each pixel. Modern video cards generally provide three color depths of eight, sixteen, or twenty-four bits, allowing 256, 65536, or over 16 million colors per pixel on the display. This produces images that are somewhat coarse and grainy (eight-bit images) to "Polaroid quality" (16-bit images), on up to "photographic quality" (24-bit images)[10].

One problem with these color depths is that two of the three formats do not contain a number of bits that is evenly divisible by three. Therefore, in each of these formats at least one of the three primary colors will have fewer bits than the others. For example, with an eight-bit color depth, two of the colors can have three bits (or eight different shades) associated with them while one of the colors must have only two bits (or four shades). Therefore, when distributing the bits there are three formats possible: 2-3-3 (two bits red, three bits green, and three bits blue), 3-2-3, or 3-3-2. Likewise, with a 16 bit color depth, two of the three colors can have five bits while the third color can have six bits. This lets us generate three different palettes using the bit values 5-5-6, 5-6-5, or 6-5-5. For 24-bit displays, each primary color can have eight bits, so there is an even distribution of the colors for each pixel.

A 24-bit display produces amazingly good results. A 16-bit display produces okay images. Eight-bit displays, to put it bluntly, produce horrible photographic images (they do produce good synthetic images like those you would manipulate with a draw program). To produce better images when using an eight-bit display, most cards provide a hardware *palette*. A palette is nothing more than an array of 24-bit values containing 256 elements[11]. The system uses the eight-bit pixel value as an index into this array of 256 values and displays the color associated with the 24-bit entry in the palette table. Although the display can still dis-

---

10. Some graphic artists would argue that 24 bit images are not of a sufficient quality. There are some display/printer./scanner devices capable of working with 32-bit, 36-bit, and even 48-bit images; if, of course, you're willing to pay for them.

11. Actually, the color depth of each palette entry is not necessarily fixed at 24 bits. Some display devices, for example, use 18-bit entries in their palette.

play only 256 different colors at one time, the palette mechanism lets users select exactly which colors they want to display. For example, they could display 250 shades of blue and six shades of purple if such a mixture produces a better image for them.

Eight-bit pixel value provide an index into a table of 256 24-bit values. The value of the selected element specifies the 24-bit color to display.

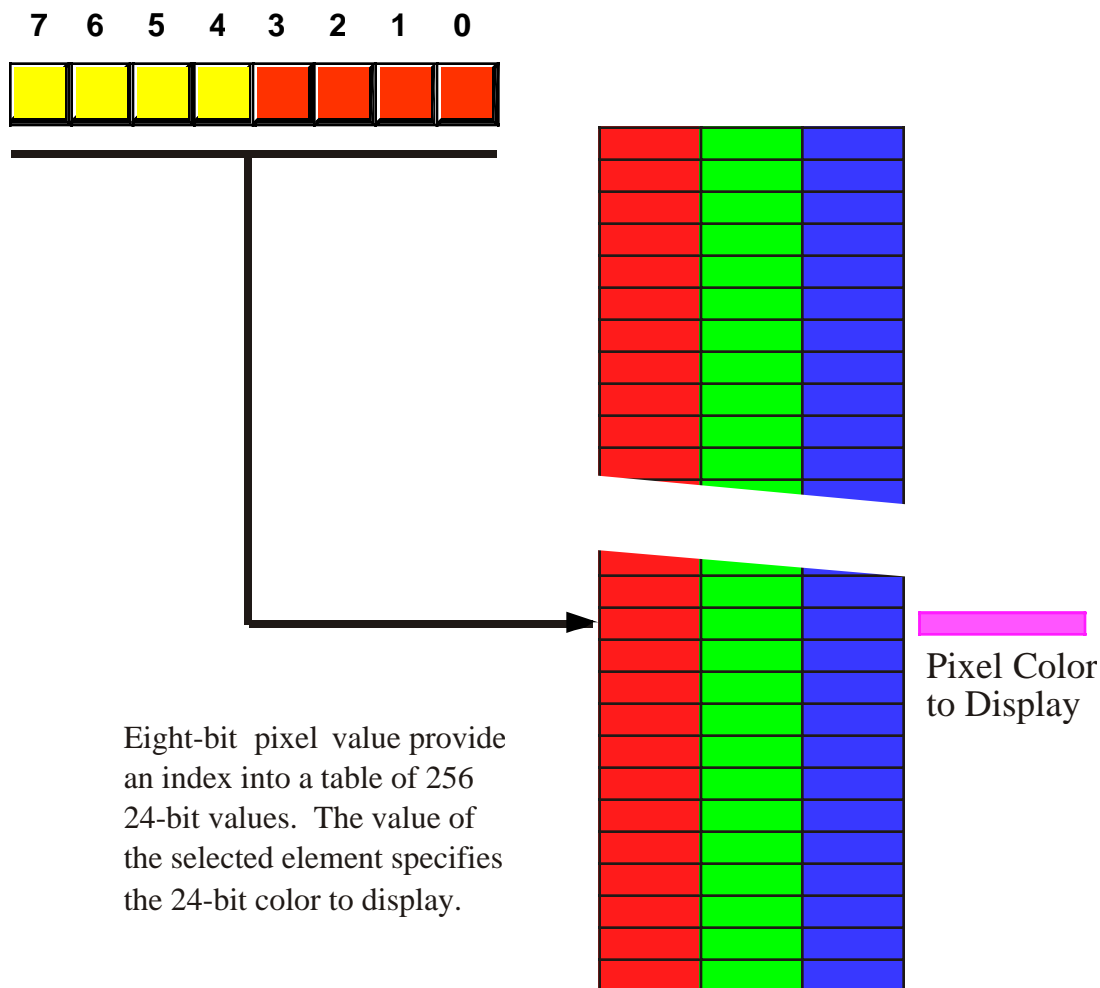Pixel Color to Display

Figure 4.7   Extending the Number of Colors Using a Palette

Unfortunately, the palette scheme only works for displays with minimal color depths. For example, attempting to use a palette with 16-bit images would require a lookup table with 65,536 different three-byte entries – a bit much for today's operating systems (since they may have to reload the palette every time you select a window on the display). Fortunately, the higher bit depths don't require the palette concept as much as the eight-bit color depth.

Obviously, we could dream up other schemes for representing pixel color on the display. Some display systems, for example, use the subtractive primary colors (Cyan, Yellow, and Magenta, plus Black, the so-called CYMK color space). Other display system use fewer or more bits to represent the values. Some distribute the bits between various shades. Monochrome displays typically use one, four, or eight bit pixels to display various gray scales (e.g., two, sixteen, or 256 shades of gray). However, the bit organizations of this section are among the more popular in use by display adapters.

4.6.2 Representing Audio Information

Another real-world quantity you'll often find in digital form on a computer is audio information. WAV files, MP3 files, and other audio formats are quite popular on personal computers. An interesting question is "how do we represent audio information inside the computer?" While many sound formats are far too complex to discuss here (e.g., the MP3 format), it is relatively easy to represent sound using a simple sound data format (something similar to the WAV file format). In this section we'll explore a couple of possible ways to represent audio information; but before we take a look at the digital format, perhaps it's a wise idea to study the analog format first.
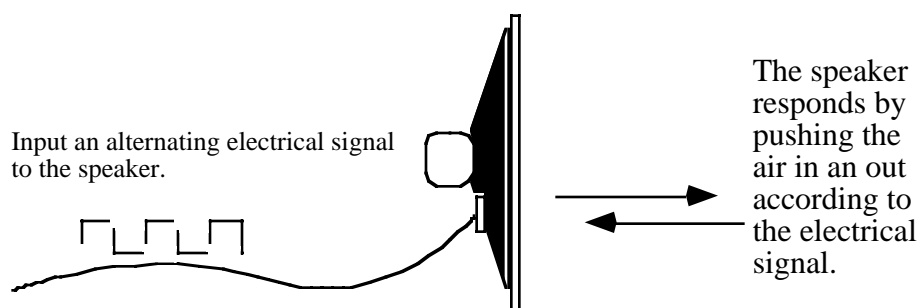

Figure 4.8   Operation of a Speaker

Sounds you hear are the result of vibrating air molecules. When air molecules quickly vibrate back and forth between 20 and 20,000 times per second, we interpret this as some sort of sound. A speaker (see Figure 4.8) is a device which vibrates air in response to an electrical signal. That is, it converts an electric signal which alternates between 20 and 20,000 times per second (Hz) to an audible tone. Alternating a signal is very easy on a computer, all you have to do is apply a logic one to an output port for some period of time and then write a logic zero to the output port for a short period. Then repeat this over and over again. A plot of this activity over time appears in Figure 4.9.
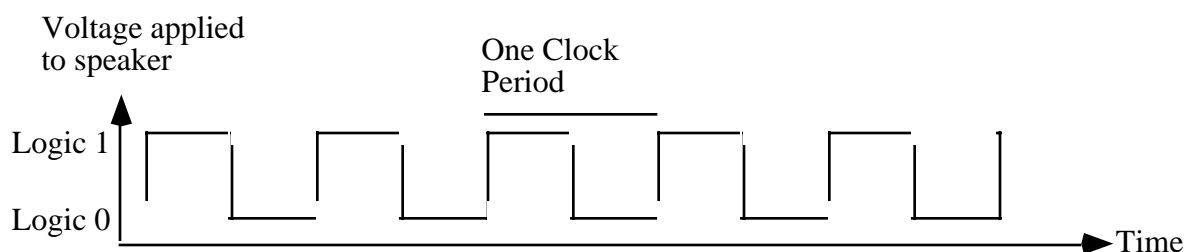
Note: Frequency is equal to the reciprical of the clock period.   Audible sounds are between 20 and 20,000 Hz.

Figure 4.9   An Audible Sound Wave

Although many humans are capable of hearing tones in the range 20-20Khz, the PC's speaker is not capable of faithfully reproducing the tones in this range. It works pretty good for sounds in the range 100-10Khz, but the volume drops off dramatically outside this range. Fortunately, most modern PCs contain a sound card that is quite capable (with appropriate external speakers) of faithfully representing "CD-Quality" sound.   Of course, a good question might be "what is CD-Quality sound, anyway?"   Well, to answer

that question, we've got to decide how we're going to represent sound information in a binary format (see "What is "Digital Audio" Anyway?" on page 112).

Take another look at Figure 4.9. This is a graph of amplitude (volume level) over time. If logic one corresponds to a fully extended speaker cone and logic zero corresponds to a fully retracted speaker cone, then the graph in Figure 4.9 suggests that we are constantly pushing the speaker cone in an out as time progresses. This analog data, by the way, produces what is known as a "square wave" which tends to be a very bright sound at high frequencies and a very buzzy sound at low frequencies. One advantage of a square wave tone is that we only need to alternate a single bit of data over time in order to produce a tone. This is very easy to do and very inexpensive. These two reasons are why the PC's built-in speaker (not the sound card) uses exactly this technique for producing beeps and squawks.

To produce different tones with a square wave sound system is very easy. All you've got to do is write a one and a zero to some bit connected to the speaker somewhere between 20 and 20,000 times per second. You can even produce "warbling" sounds by varying the frequency at which you write those zeros and ones to the speaker.

One easy data format we can develop to represent digitized (or, should we say, "binarized") audio data is to create a stream of bits that we feed to the speaker every $^1/_{40,000}$ seconds. By alternating ones and zeros in this bit stream, we get a 20 KHz tone (remember, it takes a high and a low section to give us one clock period, hence it will take two bits to produce a single cycle on the output). To get a 20 Hz tone, you would create a bit stream that alternates between 1,000 zeros and 1,000 ones. With 1,000 zeros, the speaker will remain in the retracted position for $^1/_{40}$ seconds, following that with 1,000 ones leaves the speaker in the fully extended position for $^1/_{40}$ seconds. The end result is that the speaker moves in and out 20 times a second (giving us our 20 Hz frequency). Of course, you don't have to emit a regular pattern of zeros and ones. By varying the positions of the ones and zeros in your data stream you can dramatically affect the type of sound the system will produce.

The length of your data stream will determine how long the sound plays. With 40,000 bits, the sound will play for one second (assuming each bit's duration is $^1/_{40,000}$ seconds). As you can see, this sound format will consume 5,000 bytes per second. This may seem like a lot, but it's relatively modest by digital audio standards.

Unfortunately, square waves are very limited with respect to the sounds they can produce and are not very high fidelity (certainly not "CD-Quality"). Real analog audio signals are much more complex and you cannot represent them with two different voltage levels on a speaker. Figure 4.10 provides a typical example

What is "Digital Audio" Anyway?

"Digital Audio" or "digitized audio" is the conventional term the consumer electronics industry uses to describe audio information encoded for use on a computer. What exactly does the term "digital" mean in this case. Historically, the term "digit" refers to a finger. A digital numbering system is one based on counting one's fingers. Traditionally, then, a "digital number" was a base ten number (since the numbering system we most commonly use is based on the ten digits with which God endowed us). In the early days of computer systems the terms "digital computer" and "binary computer" were quite prevalent, with digital computers describing decimal computer systems (i.e., BCD-based systems). Binary computers, of course, were those based on the binary numbering system. Although BCD computers are mainly an artifact in the historical dust bin, the name "digital computer" lives on and is the common term to describe all computer systems, binary or otherwise. Therefore, when people talk about the logic gates computer designers use to create computer systems, they call them "digital logic." Likewise, when they refer to computerized data (like audio data), they refer to it as "digital." Technically, the term "digital" should mean base ten, not base two. Therefore, we should really refer to "digital audio" as "binary audio" to be technically correct. However, it's a little late in the game to change this term, so "digital XXXXX" lives on. Just keep in mind that the two terms "digital audio" and "binary audio" really do mean the same thing, even though they shouldn't.

of an audio waveform. Notice that the frequency and the amplitude (the height of the signal) varies considerably over time. To capture the height of the waveform at any given point in time we will need more than two values; hence, we'll need more than a single bit.

Voltage applied
to speaker
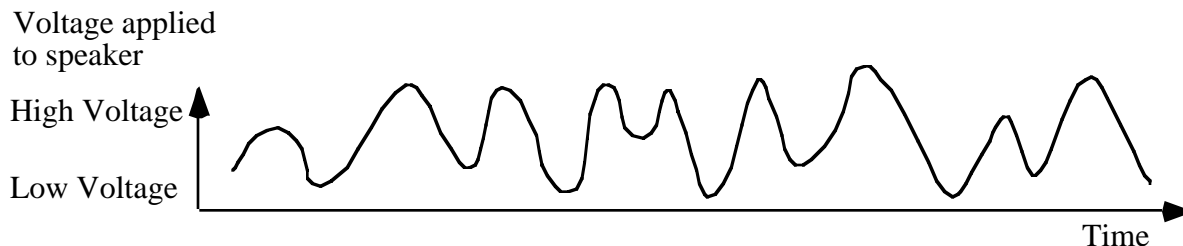
High Voltage

Low Voltage

Time

Figure 4.10   A Typical Audio Waveform

An obvious first approximation is to use a byte, rather than a single bit, to represent each point in time on our waveform. We can convert this byte data to an analog signal using a "digital to analog converter" (how obvious) or DAC. This accepts some binary number as input and produces an analog voltage on its output. This allows us to represent an impressive 256 different voltage levels in the waveform. By using eight bits, we can produce a far wider range of sounds than are possible with a single bit. Of course, our data stream now consumes 40,000 bytes per second; quite a big step up from the 5,000 bytes/second in the previous example, but still relatively modest in terms of digital audio data rates.

You might think that 256 levels would be sufficient to produce some impressive audio. Unfortunately, our hearing is logarithmic in nature and it takes an order of magnitude difference in signal for a sound to appear just a little bit louder. Therefore, our 256 different analog levels aren't as impressive to our ears. Although you can produce some decent sounds with an eight-bit data stream, it's still not high fidelity and certainly not "CD-Quality" audio.

The next obvious step up the ladder is a 16-bit value for each point of our digital audio stream. With 65,536 different analog levels we finally reach the realm of "CD-Quality" audio. Of course, we're now consuming 80,000 bytes per second to achieve this! For technical reasons, the Compact Disc format actually requires 44,100 16-bit samples per second. For a stereo (rather than monaural) data stream, you need two 16-bit values each $\frac{1}{44,100}$ seconds. This produces a whopping data rate of over 160,000 bytes per second. Now you understand the claim a littler earlier that 5,000 bytes per second is a relatively modest data rate.

Some very high quality digital audio systems use 20 or 24 bits of information and record the data at a higher frequency than 44.1 KHz (48 KHz is popular, for example). Such data formats record a better signal at the expense of a higher data rate. Some sound systems don't require anywhere near the fidelity levels of even a CD-Quality recording. Telephone conversations, for example, require only about 5,000 eight-bit samples per second (this, by the way, is why phone modems are limited to approximately 56,000 bits per second, which is about 5,000 bytes per second plus some overhead). Some common "digitizing" rates for audio include the following:

- Eight-bit samples at 11 KHz
- Eight-bit samples at 22 KHz
- Eight-bit samples at 44.1 KHz
- 16-bit samples at 32 KHz
- 16-bit samples at 44.1 KHz
- 16-bit samples at 48 KHz
- 24-bit samples at 44.1 KHz (generally in professional recording systems)
- 24-bit samples at 48 KHz   (generally in professional recording systems)

The fidelity increases as you move down this list.

The exact format for various audio file formats is way beyond the scope of this text since many of the formats incorporate data compression. Some simple audio file formats like WAV and AIFF consist of little more than the digitized byte stream, but other formats are nearly indecipherable in their complexity. The exact nature of a sound data type is highly dependent upon the sound hardware in your system, so we won't delve any farther into this subject. There are several books available on computer audio and sound file formats if you're interested in pursuing this subject farther.

4.6.3 Representing Musical Information

Although it is possible to compress an audio data stream somewhat, high-quality audio will consume a large amount of data. CD-Quality audio consumes just over 160 Kilobytes per second, so a CD at 650 Megabytes holds enough data for just over an hour of audio (in stereo). Earlier, you saw that we could use a palette to allow higher quality color images on an eight-bit display. An interesting question is "can we create a sound palette to let us encode higher quality audio?" Unfortunately, the general answer is no because audio information is much less redundant than video information and you cannot produce good results with rough approximation (which using a sound palette would require). However, if you're trying to produce a specific sound, rather than trying to faithfully reproduce some recording, there are some possibilities open to you.

The advantage to the digitized audio format is that it records *everything*. In a music track, for example, the digital information records all the instruments, the vocalists, the background noise, and, well, *everything*. Sometimes you might not need to retain all this information. For example, if all you want to record is a keyboard player's synthesizer, the ability to record all the other audio information simultaneously is not necessary. In fact, with an appropriate interface to the computer, recording the audio signal from the keyboard is completely unnecessary. A far more cost-effective approach (from a memory usage point of view) is to simply record the notes the keyboardist plays (along with the duration of each note and the velocity at which the keyboardist plays the note) and then simply feed this keyboard information back to the synthesizer to play the music at a later time. Since it only takes a few bytes to record each note the keyboardist plays, and the keyboardist generally plays fewer than 100 notes per second, the amount of data needed to record a complex piece of music is tiny compared to a digitized audio recording of the same performance.

One very popular format for recording musical information in this fashion is the MIDI format (MIDI stands for Musical Instrument Digital Interface and it specifies how to connect musical instruments, computers, and other equipment together). The MIDI protocol uses multi-byte values to record information about a series of instruments (a simple MIDI file can actually control up to 16 or more instruments simultaneously).

Although the internal data format of the MIDI protocol is beyond the scope of this chapter, it is interesting to note that a MIDI command is effectively equivalent to a "palette look-up" for an audio signal. When a musical instrument receives a MIDI command telling it to play back some note, that instrument generally plays back some waveform stored in the synthesizer.

Note that you don't actually need an external keyboard/synthesizer to play back MIDI files. Most sound cards contain software that will interpret MIDI commands and play the accompany notes. These cards definitely use the MIDI command as an index into a "wave table" (short for waveform lookup table) to play the accompanying sound. Although the quality of the sound these cards reproduce is often inferior to that a professional synthesizer produces, they do let you play MIDI files without purchasing an expensive synthesizer module[12].

If you're interested in the actual data format that MIDI uses, there are dozens of texts available on the MIDI format. Any local music store should carry several of these. You should also be able to find lots of information on MIDI on the Internet (try Roland's web site as a good starting point).

---

12. For those who would like a better MIDI experience using a sound card, some synthesizer manufacturers produce sound cards with an integrated synthesizer on-board.

4.6.4 Representing Video Information

Recent increases in disk space, computer speed, and network access have allowed an explosion in the popularity of multimedia on personal computers. Although the term "multimedia" suggests that the data format deals with many different types of media, most people use this term to describe digital video recording and playback on a computer system. In fact, most multimedia formats support at least two mediums: video and audio. The more popular formats like Apple's Quicktime support other concurrent media streams as well (e.g., a separate subtitle track, time codes, and device control). To simplify matters, we limit the discussion in this section to digital video streams.

Fundamentally, a video image is nothing more than a succession of still pictures that the system displays at some rate like 30 images per second. Therefore, if we want to create a digitized video image format, all we really need to do is store 30 or so pictures for each second of video we wish to view. This may not seem like a big deal, but consider that a typical "full screen" video display has 640x480 pixels or a total of 307,200 pixels. If we use a 24-bit RGB color space, then each pixel will require three bytes, raising the total to 921,600 bytes per image. Displaying 30 of these images per second means our video format will consume 27,648,000 bytes per second. Digital audio, at 160 Kilobytes per second is virtually nothing compared to the data requirements for digital video.

Although computer systems and hard disk systems have advanced tremendously over the past decade, maintaining a 30 MByte/second data rate from disk to display is a little too much to expect from all but the most expensive workstations currently available (at least, in the year 2000 as this was written). Therefore, most multimedia systems use various techniques (or combinations of these techniques) to get the data rate down to something more reasonable. In stock computer systems, a common technique is to display a 320x240 quarter screen image rather than a full-screen 640x480 image. This reduces the data rate to about seven megabytes per second.

Another technique digital video formats use is to *compress* the video data. Video data tends to contain lots of redundant information that the system can eliminate through the use of compression. The popular DV format for digital video camcorders, for example, compresses the data stream by almost 90%, requiring only a 3.3 MByte/second data rate for full-screen video. This type of compression is not without cost. There is a detectable, though slight, loss in image quality when employing DV compression on a video image. Nevertheless, this compression makes it possible to deal with digital video data streams on a contemporary computer system. Compressed data formats are a little beyond the scope of this chapter; however, by the time you finish this text you should be well-prepared to deal with compressed data formats. Programmers writing video data compression algorithms often use assembly language because compression and decompression algorithms need to be very fast to process a video stream in real time. Therefore, keep reading this text if you're interested in working on these types of algorithms.

4.6.5 Where to Get More Information About Data Types

Since there are many ways to represent a particular real-world object inside the computer, and nearly an infinite variety of real-world objects, this text cannot even begin to cover all the possibilities. In fact, one of the most important steps in writing a piece of computer software is to carefully consider what objects the software needs to represent and then choose an appropriate internal representation for that object. For some objects or processes, an internal representation is fairly obvious; for other objects or processes, developing an appropriate data type representation is a difficult task. Although we will continue to look at different data representations throughout this text, if you're really interested in learning more about data representation of real world objects, activities, and processes, you should consult a good "Data Structures and Algorithms" textbook. This text does not have the space to treat these subjects properly (since it still has to teach assembly language). Most texts on data structures present their material in a high level language. Adopting this material to assembly language is not difficult, especially once you've digested a large percentage of this text. For something a little closer to home, you might consider reading Knuth's "The Art of Computer Programming" that describes data structures and algorithms using a synthetic assembly language called *MIX*. Although MIX isn't the same as HLA or even x86 assembly language, you will probably find it easier to

convert algorithms in this text to x86 than it would be to convert algorithms written in Pascal, Java, or C++ to assembly language.

4.7   Putting It All Together

Perhaps the most important fact this chapter and the last chapter present is that computer programs all use strings of binary bits to represent data internally. It is up to an application program to distinguish between the possible representations. For example, the bit string %0100_0001 could represent the numeric value 65, an ASCII character ('A'), or the mantissa portion of a floating point value ($41). The CPU cannot and does not distinguish between these different representations, it simply processes this eight-bit value as a bit string and leaves the interpretation of the data to the application.

Beginning assembly language programmers often have trouble comprehending that they are responsible for interpreting the type of data found in memory; after all, one of the most important abstractions that high level languages provide is to associate a data type with a bit string in memory. This allows the compiler to do the interpretation of data representation rather than the programmer. Therefore, an important point this chapter makes is that assembly language programmers must handle this interpretation themselves. The HLA language provides built-in data types that seem to provide these abstractions, but keep in mind that once you've loaded a value into a register, HLA can no longer interpret that data for you, it is your responsibility to use the appropriate machine instructions that operate on the specified data.

One small amount of checking that HLA and the CPU does enforce is size checking - HLA will not allow you to mix sizes of operands within most instructions[13]. That is, you cannot specify a byte operand and a word operand in the same instruction that expects its two operands to be the same size. However, as the following program indicates, you can easily write a program that treats the same value as completely different types.

```
program dataInterpretation;
#include( "stdlib.hhf" );
static
    r:  real32 := -1.0;

begin dataInterpretation;


    stdout.put( "'r' interpreted as a real32 value: ", r:5:2, nl );

    stdout.put( "'r' interpreted as an uns32 value: " );
    mov( r, eax );
    stdout.putu32( eax );
    stdout.newln();

    stdout.put( "'r' interpreted as an int32 value: " );
    mov( r, eax );
    stdout.puti32( eax );
    stdout.newln();

    stdout.put( "'r' interpreted as a dword value: $" );
    mov( r, eax );
    stdout.putd( eax );
    stdout.newln();

end dataInterpretation;
```

---

13. The sign and zero extension instructions are an obvious exception, though HLA still checks the operand sizes to ensure they are appropriate.

Program 4.4   Interpreting a Single Value as Several Different Data Types

As this sample program demonstrates, you can get completely different results by interpreting your data differently during your program's execution. So always remember, it is your responsibility to interpret the data in your program. HLA helps a little by allowing you to declare data types that are slightly more abstract than bytes, words, or double words; HLA also provides certain support routines, like stdout.put, that will automatically interpret these abstract data types for you; however, it is generally your responsibility to use the appropriate machine instructions to consistently manipulate memory objects according to their data type.

Chapter Five

Questions, Projects, and Lab Exercises

5.1 Questions

1) List the legal forms of a *boolean* expression in an HLA IF statement.

2) What data type do you use to declare a
   a) 32-bit signed integer?
   b) 16-bit signed integer?
   c) 8-bit signed integer?

3) List all of the 80x86:
   a) 8-bit general purpose registers.
   b) 16-bit general purpose registers.
   c) 32-bit general purpose registers.

4) Which registers overlap with
   a) ax?
   b) bx?
   c) cx?
   d) dx?
   e) si?
   f) di?
   g) bp?
   h) sp?

5) In what register does the condition codes appear?

6) What is the generic syntax of the HLA MOV instruction?

7) What are the legal operand formats for the MOV instruction?

8) What do the following symbols denote in an HLA boolean expression?
   a) @c
   b) @nc
   c) @z
   d) @nz
   e) @o
   f) @no
   g) @s
   h) @ns

9) Collectively, what do we call the carry, overflow, zero, and sign flags?

10) What high level language control structures does HLA provide?

11) What does the *nl* symbol represent?

12) What routine would you call, that doesn't require any parameters, to print a new line on the screen?

13) If you wanted to print a nicely-formatted column of 32-bit integer values, what standard library routines could you call to achieve this?

14) The stdin.getc() routine does not allow a parameter. Where does it return the character it reads from the user?

15) When reading an integer value from the user via the stdin.geti$X$ routines, the program will stop with an exception if the user enters a value that is out of range or enters a value that contains illegal characters. How can you trap this error

16) What is the difference between the stdin.ReadLn() and stdin.FlushInput() procedures?

17) Convert the following decimal values to binary:

a) 128	b) 4096	c) 256	d) 65536	e) 254
f) 9	g) 1024	h) 15	i) 344	j) 998
k) 255	l) 512	m) 1023	n) 2048	o) 4095
p) 8192	q) 16,384	r) 32,768	s) 6,334	t) 12,334
u) 23,465	v) 5,643	w) 464	x) 67	y) 888

18) Convert the following binary values to decimal:

a) 1001 1001	b) 1001 1101	c) 1100 0011	d) 0000 1001	e)1111 1111
f) 0000 1111	g) 0111 1111	h) 1010 0101	i) 0100 0101	j) 0101 1010
k) 1111 0000	l) 1011 1101	m) 1100 0010	n) 0111 1110	o) 1110 1111
p) 0001 1000	q) 1001 111 1	r) 0100 0010	s) 1101 1100	t) 1111 0001
u) 0110 1001	v) 0101 1011	w) 1011 1001	x) 1110 0110	y) 1001 0111

19) Convert the binary values in problem 2 to hexadecimal.

20) Convert the following hexadecimal values to binary:

a) 0ABCD	b) 1024	c) 0DEAD	d) 0ADD	e) 0BEEF
f) 8	g) 05AAF	h) 0FFFF	i) 0ACDB	j) 0CDBA
k) 0FEBA	l) 35	m) 0BA	n) 0ABA	o) 0BAD
p) 0DAB	q) 4321	r) 334	s) 45	t) 0E65
u) 0BEAD	v) 0ABE	w) 0DEAF	x) 0DAD	y) 9876

Perform the following hex computations (leave the result in hex):

21) 1234 +9876

22) 0FFF - 0F34

23) 100 - 1

24) 0FFE - 1

25) What is the importance of a nibble?

26) How many hexadecimal digits in:

a) a byte   b) a word   c) a double word

27) How many bits in a:

a) nibble   b) byte   c) word   d) double word

28) Which bit (number) is the H.O. bit in a:

a) nibble b) byte   c) word   d) double word

29) What character do we use as a suffix for hexadecimal numbers? Binary numbers? Decimal numbers?

30) Assuming a 16-bit two's complement format, determine which of the values in question 4 are positive and which are negative.

31) Sign extend all of the values in question two to sixteen bits. Provide your answer in hex.

32) Perform the bitwise AND operation on the following pairs of hexadecimal values. Present your answer in hex. (Hint: convert hex values to binary, do the operation, then convert back to hex).

a) 0FF00, 0FF0 b) 0F00F, 1234 c) 4321, 1234 d) 2341, 3241   e) 0FFFF, 0EDCB

f) 1111, 5789 g) 0FABA, 4322 h) 5523, 0F572 i) 2355, 7466   j) 4765, 6543

k) 0ABCD, 0EFDC l) 0DDDD, 1234 m) 0CCCC, 0ABCD n) 0BBBB, 1234 o) 0AAAA, 1234

p) 0EEEE, 1248 q) 8888, 1248 r) 8086, 124F s) 8086, 0CFA7 t) 8765, 3456

u) 7089, 0FEDC v) 2435, 0BCDE w) 6355, 0EFDC x) 0CBA, 6884 y) 0AC7, 365

33) Perform the logical OR operation on the above pairs of numbers.

34) Perform the logical XOR operation on the above pairs of numbers.

35) Perform the logical NOT operation on all the values in question four. Assume all values are 16 bits.

36) Perform the two's complement operation on all the values in question four. Assume 16 bit values.

37) Sign extend the following hexadecimal values from eight to sixteen bits. Present your answer in hex.

a) FF   b) 82   c) 12   d) 56   e) 98

f) BF   g) 0F   h) 78   i) 7F   j) F7

k) 0E l) AE   m) 45   n) 93   o) C0

p) 8F q) DA   r) 1D   s) 0D   t) DE

u) 54 v) 45   w) F0   x) AD   y) DD

38) Sign contract the following values from sixteen bits to eight bits. If you cannot perform the operation, explain why.

a) FF00 b) FF12   c) FFF0   d) 12   e) 80

f) FFFF g) FF88   h) FF7F   i) 7F   j) 2

k) 8080 l) 80FF   m) FF80   n) FF   o) 8

p) F   q) 1   r) 834   s) 34   t) 23

u) 67   v) 89   w) 98   x) FF98   y) F98

39) Sign extend the 16-bit values in question 22 to 32 bits.

40) Assuming the values in question 22 are 16-bit values, perform the left shift operation on them.

41) Assuming the values in question 22 are 16-bit values, perform the logical right shift operation on them.

42) Assuming the values in question 22 are 16-bit values, perform the arithmetic right shift operation on them.

43) Assuming the values in question 22 are 16-bit values, perform the rotate left operation on them.

44) Assuming the values in question 22 are 16-bit values, perform the rotate right operation on them.

45) Convert the following dates to the short packed format described in this chapter (see "Bit Fields and Packed Data" on page 81). Present your values as a 16-bit hex number.

a) 1/1/92 b) 2/4/56   c) 6/19/60   d) 6/16/86   e) 1/1/99

46) Convert the above dates to the long packed data format described in this chapter.

47) Describe how to use the shift and logical operations to *extract* the day field from the packed date record in question 29. That is, wind up with a 16-bit integer value in the range 0..31.

48) Assume you've loaded a long packed date (See "Bit Fields and Packed Data" on page 81.) into the EAX register. Explain how you can easily access the day and month fields directly, without any shifting or rotating of the EAX register.

49) Suppose you have a value in the range 0..9. Explain how you could convert it to an ASCII character using the basic logical operations.

50) The following C++ function locates the first set bit in the *BitMap* parameter starting at bit position *start* and working up to the H.O. bit . If no such bit exists, it returns -1. Explain, in detail, how this function works.

```
int FindFirstSet(unsigned BitMap, unsigned start)
{
        unsigned Mask = (1 << start);

        while (Mask)
        {
            if (BitMap & Mask) return start;
            ++start;
            Mask <<= 1;
        }
        return -1;
}
```

51) The C++ programming language does not specify how many bits there are in an unsigned integer. Explain why the code above will work regardless of the number of bits in an unsigned integer.

52) The following C++ function is the complement to the function in the questions above. It locates the first zero bit in the *BitMap* parameter. Explain, in detail, how it accomplishes this.

```
int FindFirstClr(unsigned BitMap, unsigned start)
{
        return FindFirstSet(~BitMap, start);
}
```

53) The following two functions set or clear (respectively) a particular bit and return the new result. Explain, in detail, how these functions operate.

```
unsigned SetBit(unsigned BitMap, unsigned position)
{
        return BitMap | (1 << position);
}

unsigned ClrBit(unsigned BitMap, unsigned position)
{
        return BitMap & ~(1 << position);
}
```

54) In code appearing in the questions above, explain what happens if the start and position parameters contain a value greater than or equal to the number of bits in an unsigned integer.

55) Provide an example of HLA variable declarations for the following data types:

a) Eight-bit byte

b) 16-bit word

c) 32-bit dword

d) Boolean

e) 32-bit floating point

f) 64-bit floating point

g) 80-bit floating point

h)   Character

56) The long packed date format offers two advantages over the short date format. What are these advantages?

57) Convert the following real values to 32-bit single precision floating point format. Provide your answers in hexadecimal, explain your answers.

a) 1.0   b) 2.0   c) 1.5   d) 10.0

e) 0.5   f) 0.25   g) 0.1   h) -1.0

i) 1.75   j) 128   k) 1e-2   l) 1.024e+3

58) Which of the values in question 41 do not have exact representations?

59) Show how to declare a character variable that is initialized with the character "*".

60) Show how to declare a character variable that is initialized with the control-A character (See "The ASCII Character Set" on page 104 for the ASCII code for control-A).

61) How many characters are present in the standard ASCII character set?

62) What is the basic structure of an HLA program?

63) Which HLA looping control structure(s) test(s) for loop termination at the beginning of the loop?

64) Which HLA looping control structure(s) test(s) for loop termination at the end of the loop?

65) Which HLA looping construct lets you create an infinite loop?

66) What set of flags are known as the "condition codes?"

67) What HLA statement would you use to trap exceptions?

68) Explain how the IN operator works in a boolean expression.

69) What is the *stdio.bs* constant?

70) How do you redirect the standard output of your programs so that the data is written to a text file?

5.2   Programming Projects for Chapter Two

1) Write a program to produce an "addition table." This table should input two small *int8* values from the user. It should verify that the input is correct (i.e., handle the *ex.ConversionError* and *ex.ValueOutOfRange* exceptions) and is positive. The second value must be greater than the first input value. The program will display a row of values between the lower and upper input values. It will also print a column of values between the two values specified. Finally, it will print a matrix of sums. The following is the expected output for the user inputs 15 & 18

```
add     15  16  17  18
15      30  31  32  33
16      31  32  33  34
17      32  33  34  35
18      33  34  35  36
```

2) Modify program (1), above, to draw lines between the columns and rows. Use the hyphen ('-'), vertical bar ('|'), and plus sign ('+') characters to draw the lines. E.g.,

```
add  | 15 | 16 | 17 | 18 |
-----+----+----+----+----+
15   | 30 | 31 | 32 | 33 |
-----+----+----+----+----+
16   | 31 | 32 | 33 | 34 |
-----+----+----+----+----+
17   | 32 | 33 | 34 | 35 |
-----+----+----+----+----+
18   | 33 | 34 | 35 | 36 |
-----+----+----+----+----+
```

For extra credit, use the line drawing character graphics symbols listed in Appendix B to draw the lines. Note: to print a character constant as an ASCII code, use "#nnn" where "nnn" represents the ASCII code of the character you wish to print. For example, "stdout.put(#179);" prints the line drawing vertical bar character. (This option is available only on machines that support the IBM-PC extended character set.)

3) Write a program that generates a "Powers of Four" table. Note that you can create the powers of four by loading a register with one and then successively add that register to itself twice for each power of two.

4) Write a program that reads a list of positive numbers from a user until that user enters a negative or zero value. Display the sum of those positive integers.

5) Write a program that computes (n)(n+1)/2. It should read the value "n" from the user. Hint: you can compute this formula by adding up all the numbers between one and n.

5.3   Programming Projects for Chapter Three

Write each of the following programs in HLA. Be sure to fully comment your source code. See Appendix C for style guidelines and rules when writing HLA programs (and follow these rules to the letter!). Include sample output and a short descriptive write up with your program submission(s).

1) Write a program that reads a line of characters from the user and displays that entire line after converting any upper case characters to lower case. All non-alphabetic and existing lower case characters should pass through unchanged; you should convert all upper case characters to lower case before printing them.

2) Write a program that reads a line of characters from the user and displays that entire line after swapping upper case characters with lower case; that is, convert the incoming lower case characters to upper case and convert the incoming upper case characters to lower case. All non-alphabetic characters should pass through unchanged.

3) Write a program that reads three values from the user: a month, a day, and a year. Pack the date into the long date format appearing in this chapter and display the result in hexadecimal. If the date is between 2000 and 2099, also pack the date into the short packed date format and display that 16-bit value in hexadecimal form. If the date is not in the range 2000..2099, then display a short message suggesting that the translation is not possible.

4) Write a date validation program that reads a month, day, and year from the user, verifies that the date is correct (ignore leap years for the time being), and then packs the date into the long date format appearing in this chapter.

5) Write a "CntBits" program that counts the number of one bits in a 16-bit integer value input from the user. *Do not use any built-in functions in HLA's library to count these bits for you.* Use the shift or rotate instructions to extract each bit in the value.

6) Write a "TestBit" program. This program requires two integer inputs. The first value is a 32-bit integer to test; the second value is an unsigned integer in the range 0..31 describing which bit to test. The program should display true if the corresponding bit (in the test value) contains a one, the program should display false if that bit position contains a zero. The program should always display false if the second value holds a value outside the range 0..31.

7) Write a program that reads an eight-bit signed integer and a 32-bit signed integer from the user that computes and displays the sum and difference of these two numbers.

8) Write a program that reads an eight-bit unsigned integer and a 16-bit unsigned integer from the user that computes and displays the sum and the absolute value of the difference of these two numbers.

9) Write a program that reads a 32-bit unsigned integer from the user and displays this value in binary. Use the SHL instruction to perform the integer to binary conversion.

10) Write a program that uses *stdin.getc* to read a sequence of binary digits from the user (that is, a sequence of '1' and '0' characters). Convert this string to an integer using the AND, SHL, and OR instructions. Display the integer result in hexadecimal and decimal. You may assume the user will not enter more than 32 digits.

11) Using the LAFH instruction, write a program that will display the current values of the carry, sign, and zero flags as boolean values. Read two integer values from the user, add them together, and them immediately capture the flags' values using the LAHF instruction and display the result of these three flags as boolean values. Hint: use the SHL or SHR instructions to extract the specific flag bits.

5.4   Programming Projects for Chapter Four

1) Write an HLA program that reads a single precision floating point number from the user and prints the internal representation of that value using hexadecimal notation.

2) Write a program that reads a single precision floating point value from the user, takes the absolute value of that number, and then displays the result. Hint: this program does not use any arithmetic instructions or comparisons. Take a look at the binary representation for floating point numbers in order to solve this problem.

3) Write a program that generates an ASCII character set chart using the following output format:

```
   | 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
--+------------------------------------------------
20|      !  "  #  ...
30|   0  1  2  3  ...
40|   @  A  B  C  ...
50|   P  Q  R  S  ...
60|   `  a  b  c  ...
70|   p  q  r  s  ...
```

Note that the columns in the table represent the L.O. four bits of the ASCII code, the rows in the table represent the H.O. four bits of the ASCII code. Note: for extra consideration, use the line-drawing graphic char-

acters (see Appendix B) to draw the lines in the table (this option is available only on systems that support the IBM-PC extended chracter set).

4) Using only five FOR loops, four calls to *stdout.putcSize*, and two calls to *stdout.newln*, write a program that draws a checkerboard pattern. Your checkerboard should look like the following:

```
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
        ********        ********        ********        ********
```

5.5   Laboratory Exercises for Chapter Two

Before you can write, compile, and run a single HLA program, you will need to install the HLA language system on your computer. If you are using HLA at school or some other institution, the system administrator has probably set up HLA for you. If you are working at home or on some computer on which HLA is not installed, you will need to obtain the HLA distribution package and set it up on your computer. The first section of this set of laboratory exercises deals with obtaining and installing HLA.

Once HLA is installed, the next step is to take stock of what is present in the HLA system. The second part of this laboratory deals with the files that are present in the HLA distribution.

Finally, and probably most important, this set of laboratory exercises discusses how to write, compile, and run some simple HLA programs.

5.5.1 A Short Note on Laboratory Exercises and Lab Reports

Whenever you work on laboratory exercises in this textbook you should always prepare a lab report associated with each exercise. Your instructor may have some specific guidelines concerning the content of

the lab report (if your instructor requires that you submit the report). Be sure to check with your instructor concerning the lab report requirements.

At a bare minimum, a lab report should contain the following:

- A title page with the lab title (chapter #), your name and other identification, the current date, and the due date. If you have a course-related computer account, you should also include your login name.
- If you modify or create a program in a lab exercise, the source code for that program should appear in the laboratory report (do not simply reprint source code appearing in this text in order to pad your lab report).
- Output from all programs should also appear in the lab report.
- For each exercise, you should provide a write-up describing the purpose of the exercise, what you learned from the exercise, and any comments about improvements or other work you've done with the exercise.
- If you make any mistakes that require correction, you should include the source code for the incorrect program with your lab report. Hand write on the listing where the error occurs and describe (in handwriting, on the listing) what you did to correct the problem. **Note**: no one is perfect. If you turn in a lab report that has no listings with errors in it, this is a clear indication that you didn't bother to perform this part of the exercise.
- Appropriate diagrams.

The lab report should be prepared with a word processing program. Hand-written reports are unacceptable (although hand-drawn diagrams are acceptable if a suitable drawing package isn't available). The report should be proofread and of finished quality before submission. Only the listings with errors (and hand written annotations) should be in a less than finished quality. See the "HLA Programming Style Guidelines" appendix for important information concerning programming style. Adhere to these guidelines in the HLA programs you submit.

5.5.2 Compiling Your First Program

Once HLA is operational, you can compile and run actual working programs. The HLA distribution contains lots of example HLA programs, including the HLA programs appearing in this text. Since these examples are already written, tested, and ready to compile and run, it makes sense to work with one of these example files when compiling your first program.

A good first program is the "Hello World" program appearing earlier in this volume (repeated below):

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

begin helloWorld;

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

end helloWorld;
```

The source code for this program appears in the "HelloWorld.hla" file in the "aoa\volume1\ch01" directory. Create a new subdirectory (e.g., in your "home" directory) and name this new directory "lab1". From the command window prompt, you can create the new subdirectory using the following command:

```
                    mkdir lab1
```

The first command above switches you to the root directory (assuming you're not there already). The second command (mkdir = "make directory") creates the *lab1* directory[1].

Copy the "Hello World" program (HelloWorld.hla) to this *lab1* directory using the following command window statement[2]:

```
copy c:\AoA\Volume1\CH02\HelloWorld.hla   c:\lab1
```

From the command prompt window, switch to this new directory using the command:

cd lab1

To compile this program, type the following at the command prompt:

hla HelloWorld

After a few moments, the command prompt should reappear. At this point, your program has been suc-cessfully compiled. To run the executable (HelloWorld.exe) that HLA has produced, you would use the fol-lowing command:

HelloWorld

The program should run, display "Hello World", and then terminate. At that time the command window should be waiting for another command.

If you have not successfully completed the previous steps, return to the previous section and repeat the steps to verify that HLA is operational on your system. Remember, each time you start a new command window under Microsoft Windows, you must execute the "ihla.bat" file (or otherwise set up the environ-ment) in order to make HLA accessible in that command window.

In your lab report, describe the output of the HLA compiler. For additional compilation information, use the following command to compile this program:

hla -v HelloWorld

The "-v" option stands for *verbose* compile. This presents more information during the compilation pro-cess. Describe the output of this verbose compilation in your lab report. If possible, capture the output and include the captured output with your lab report. To capture the output to a file, use a command like the fol-lowing:

hla -test -v HelloWorld >capture.txt

This command sends most of the output normally destined to the screen to the "capture.txt" output file. You can then load this text file into an editor for further processing. Of course, you may choose a different filename than "capture.txt" if you so desire.

5.5.3 Compiling Other Programs Appearing in this Chapter

The *volume1/ch02* directory contains all the sample programs appearing in Chapter Two. They include

- HelloWorld.hla
- CharInput.hla
- CheckerBoard.hla
- DemoMOVaddSUB.hla
- DemoVars.hla
- fib.hla
- intInput.hla

---

1. Some school's labs may not allow you to place information on the C: drive. If you want or need to place your personal working directory on a different drive, just substitute the appropriate drive letter for "C:" in these examples.

2. You may need to modify this statement if the AoA directory does not appear in the root directory of the C: drive or in "/usr"

- NumsInColums.hla
- NumsInColums2.hla
- PowersOfTwo.hla

Copy each of these files to your lab1 subdirectory. Compile and run each of these programs. Describe the output of each in your lab report.

5.5.4 Creating and Modifying HLA Programs

In order to create or modify HLA programs you will need to use a *text* editor to manipulate HLA source code. Windows provides two low-end text editors: notepad.exe and edit.exe. Notepad.exe is a windows-based application while edit.exe is a console (command prompt) application. Neither editor is particularly good for editing program source code; if you have an option to use a different text editor (e.g., the Microsoft Visual Studio system that comes with VC++ and other Microsoft languages), you should certainly do so. Since the choice of text editor is very personal, this text will not make any assumptions about your choice; this means that this lab exercise cannot explain the steps you need to follow in order to edit an HLA program. However, this text does assume that you've written, compiled, and run high level language programs; you can generally use the same tools for editing HLA programs that you've been using for high level language programs.

Warning: do not use Microsoft Word, Wordpad, or any other word processing programs to create or modify HLA programs. Word processing programs insert extra characters into the document that are incompatible with HLA. If you accidentally save a source file from one of these word processors, you will not be able to compile the program[3].

For the time being, edit the "HelloWorld.hla" program and modify the statement:

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

Change the text 'World of Assembly Language' to your name, e.g.,

stdout.put("Hello Randall Hyde", nl);

After you've done this, save the file to disk and recompile and run the program. Assuming you haven't introduced any typographical errors into the program, it should compile and run without incident. After making the modifications to the program, capture the output and include the captured output in your lab report. You can capture the output from this program by using the I/O redirection operator as follows:

Windows: HelloWorld >out.txt

This sends the output ("Hello Randall Hyde") to the "out.txt" text file rather than to the display. Include the sample output and the modified program in your lab report. Note: don't forget to include any erroneous source code in your lab report to demonstrate the changes you've made during development of the code.

5.5.5 Writing a New Program

To create a brand-new program is relatively east. Just create a new file with your text editor and enter the HLA program into that file. As an exercise, enter the following program into your editor (note: this program is not available in the AoA directory, you must enter this file yourself):

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

---

3. Note that many word processing programs provide a "save as text" option. If you accidentally destroy a source file by saving it from a word processor, simply reenter the word processor and save the file as text.

```
static
    One: int32;

begin onePlusOne;

    mov( 1, One );
    mov( One, eax );
    add( One, eax );
    mov( eax, One );
    stdout.put( "One + One = ", One, nl );

end onePlusOne;
```

Program 5.1   OnePlusOne Program

Remember, HLA is very particular about the way you spell names. So be sure that the alphabetic case is correct on all identifiers in this program. Before attempting to compile your program, proof read it to check for any typographical errors.

After entering and saving the program above, exit the editor and compile this program from the command prompt. If there are any errors in the program, reenter the editor, correct the errors, and then compile the program again. Repeat until the program compiles correctly.

Note: If you encounter any errors during compilation, make a printout of the program (with errors) and hand write on the printout where the errors occur and what was necessary to correct the error(s). Include this printout with your lab report.

After the program compiles successfully, run it and verify that it runs correctly. Include a printout of the program and the captured output in your lab report.

5.5.6 Correcting Errors in an HLA Program

The following program (HasAnError.hla in the appropriate AoA directory) contains a minor syntax error (a missing semicolon). Compile this program:

```
// This program has a syntactical error to
// demonstrate compilation errors in an HLA
// program.

program hasAnError;
#include( "stdlib.hhf" );
begin hasAnError;

    stdout.puts( "This program doesn't compile!" )  // missing ";"

end hasAnError;
```

Program 5.2   Sample Program With a Syntax Error

When you compile this program, you will notice that it doesn't report the error on line nine (the line actually containing the error). Instead, it reports the error on line 11 (the "end" statement) since this is the first point at which the compiler can determine that an error has occurred.

Capture the error output of this program into a text file using the following command:

Windows: hla -test HasAnError >err1.txt

Include this output in your laboratory report.

Correct the syntax error in this program and compile and run the program. Include the source code of the corrected program as well as its output in your lab report.

5.5.7 Write Your Own Sample Program

Conclude this laboratory exercise by writing a simple little program of your own. Include the source code and sample output in your lab report. If you have any syntax errors in your code, be sure to include a printout of the incorrect code with hand-written annotations describing how you fixed the problem(s) in your program.

5.6   Laboratory Exercises for Chapter Three and Chapter Four

Accompanying this text is a significant amount of software. The software can be found in the AoA/volume1 directory. Inside this directory is a set of directories with names like *ch03* and *ch04*, with the names obviously corresponding to chapters in this textbook. All the source code to the example programs in this chapter can be found in the *ch03* and *ch04* subdirectories. The *ch04* subdirectory also contains some executable programs for this chapter's laboratory exercises as well as the (Borland Delphi) source code for the lab exercises. Please see this directory for more details.

5.6.1 Data Conversion Exercises

In this exercise you will be using the "convert.exe" program found in the *ch04* subdirectory. This program displays and converts 16-bit integers using signed decimal, unsigned decimal, hexadecimal, and binary notation.

When you run this program it opens a window with four *edit boxes*. (one for each data type). Changing a value in one of the edit boxes immediately updates the values in the other boxes so they all display their corresponding representations for the new value. If you make a mistake on data entry, the program beeps and turns the edit box red until you correct the mistake. Note that you can use the mouse, cursor control keys, and the editing keys (e.g., DEL and Backspace) to change individual values in the edit boxes.

For this exercise and your laboratory report, you should explore the relationship between various binary, hexadecimal, unsigned decimal, and signed decimal values. For example, you should enter the unsigned decimal values 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, and 32768 and comment on the values that appear in the other text boxes.

The primary purpose of this exercise is to familiarize yourself with the decimal equivalents of some common binary and hexadecimal values. In your lab report, for example, you should explain what is special about the binary (and hexadecimal) equivalents of the decimal numbers above.

Another set of experiments to try is to choose various binary numbers that have exactly two bits set, e.g., 11, 110, 1100, 1 1000, 11 0000, etc. Be sure to comment on the decimal and hexadecimal results these inputs produce.

Try entering several binary numbers where the L.O. eight bits are all zero. Comment on the results in your lab report. Try the same experiment with hexadecimal numbers using zeros for the L.O. digit or the two L.O. digits.

You should also experiment with negative numbers in the signed decimal text entry box; try using values like -1, -2, -3, -256, -1024, etc. Explain the results you obtain using your knowledge of the two's complement numbering system.

Try entering even and odd numbers in unsigned decimal. Discover and describe the difference between even and odd numbers in their binary representation. Try entering multiples of other values (e.g., for three: 3, 6, 9, 12, 15, 18, 21, ...) and see if you can detect a pattern in the binary results.

Verify the hexadecimal <-> binary conversion this chapter describes. In particular, enter the same hexadecimal digit in each of the four positions of a 16-bit value and comment on the position of the corresponding bits in the binary representation. Try entering several binary values like 1111, 11110, 111100, 1111000, and 11110000. Explain the results you get and describe why you should always extend binary values so their length is an even multiple of four before converting them.

In your lab report, list the experiments above plus several you devise yourself. Explain the results you expect and include the actual results that the convert.exe program produces. Explain any insights you have while using the convert.exe program.

                           Beta Draft - Do not distribute

5.6.2 Logical Operations Exercises

The "logical.exe" program is a simple calculator that computes various logical functions. It allows you to enter binary or hexadecimal values and then it computes the result of some logical operation on the inputs. The calculator supports the dyadic logical AND, OR, and XOR. It also supports the monadic NOT, NEG (two's complement), SHL (shift left), SHR (shift right), ROL (rotate left), and ROR (rotate right).

When you run the logical.exe program it displays a set of buttons on the left hand side of the window. These buttons let you select the calculation. For example, pressing the AND button instructs the calculator to compute the logical AND operation between the two input values. If you select a monadic (unary) operation like NOT, SHL, etc., then you may only enter a single value; for the dyadic operations, both sets of text entry boxes will be active.

The logical.exe program lets you enter values in binary or hexadecimal. Note that this program automatically converts any changes in the binary text entry window to hexadecimal and updates the value in the hex entry edit box. Likewise, any changes in the hexadecimal text entry box are immediately reflected in the binary text box. If you enter an illegal value in a text entry box, the logical.exe program will turn the box red until you correct the problem.

For this laboratory exercise, you should explore each of the bitwise logical operations. Create several experiments by carefully choosing some values, manually compute the result you expect, and then run the experiment using the logical.exe program to verify your results. You should especially experiment with the masking capabilities of the logical AND, OR, and XOR operations. Try logically ANDing, ORing, and XORing different values with values like 000F, 00FF, 00F0, 0FFF, FF00, etc. Report the results and comment on them in your laboratory report.

Some experiments you might want to try, in addition to those you devise yourself, include the following:

- Devise a mask to convert ASCII values '0'..'9' to their binary integer counterparts using the logical AND operation. Try entering the ASCII codes of each of these digits when using this mask. Describe your results. What happens if you enter non-digit ASCII codes?
- Devise a mask to convert integer values in the range 0..9 to their corresponding ASCII codes using the logical OR operation. Enter each of the binary values in the range 0..9 and describe your results. What happens if you enter values outside the range 0..9? In particular, what happens if you enter values outside the range 0h..0fh?
- Devise a mask to determine whether a 16-bit integer value is positive or negative using the logical AND operation. The result should be zero if the number is positive (or zero) and it should be non-zero if the number is negative. Enter several positive and negative values to test your mask. Explain how you could use the AND operation to test *any* single bit to determine if it is zero or one.
- Devise a mask to use with the logical XOR operation that will produce the same result on the second operand as applying the logical NOT operator to that second operand.
- Verify that the SHL and SHR operators correspond to an integer multiplication by two and an integer division by two, respectively. What happens if you shift data out of the H.O. or L.O. bits? What does this correspond to in terms of integer multiplication and division?
- Apply the ROL operation to a set of positive and negative numbers. Based on your observations in Section 5.6.2, what can you say about the result when you rotate left a negative number or a positive number?
- Apply the NEG and NOT operators to a value. Discuss the similarity and the difference in their results. Describe this difference based on your knowledge of the two's complement numbering system.

5.6.3 Sign and Zero Extension Exercises

The "signext.exe" program accepts eight-bit binary or hexadecimal values then sign and zero extends them to 16 bits. Like the logical.exe program, this program lets you enter a value in either binary or hexadecimal and the program immediately zero and sign extends that value.

For your laboratory report, provide several eight-bit input values and describe the results you expect. Run these values through the signext.exe program and verify the results. For each experiment you run, be sure to list all the results in your lab report. Be sure to try values like $0, $7f, $80, and $ff.

While running these experiments, discover which hexadecimal digits appearing in the H.O. nibble produce negative 16-bit numbers and which produce positive 16-bit values. Document this set in your lab report.

Enter sets of values like (1,10), (2,20), (3,30), ..., (7,70), (8,80), (9,90), (A,A0), ..., (F,F0). Explain the results you get in your lab report. Why does "F" sign extend with zeros while "F0" sign extends with ones?

Explain in your lab report how one would sign or zero extend 16 bit values to 32 bit values. Explain why zero extension or sign extension is useful.

5.6.4 Packed Data Exercises

The packdata.exe program uses the 16-bit Date data type appearing in Chapter Three (see "Bit Fields and Packed Data" on page 81). It lets you input a date value in binary or decimal and it packs that date into a single 16-bit value.

When you run this program, it will give you a window with six data entry boxes: three to enter the date in decimal form (month, day, year) and three text entry boxes that let you enter the date in binary form. The month value should be in the range 1..12, the day value should be in the range 1..31, and the year value should be in the range 0..99. If you enter a value outside this range (or some other illegal value), then the packdata.exe program will turn the data entry box red until you correct the problem.

Choose several dates for your experiments and convert these dates to the 16-bit packed binary form by hand (if you have trouble with the decimal to binary conversion, use the conversion program from the first set of exercises in this laboratory). Then run these dates through the packdata.exe program to verify your answer. Be sure to include all program output in your lab report.

At a bare minimum, you should include the following dates in your experiments:

2/4/68, 1/1/80, 8/16/64, 7/20/60, 11/2/72, 12/25/99, Today's Date, a birthday (not necessarily yours), the due date on your lab report.

5.6.5 Running this Chapter's Sample Programs

The *ch03* and *ch04* subdirectories also contain the source code to each of the sample programs appearing in Chapters Three and Four. Compile and run each of these programs. Capture the output and include a printout of the source code and the output of each program in your laboratory report. Comment on the results produced by each program in your laboratory report.

5.6.6 Write Your Own Sample Program

To conclude your laboratory exercise, design and write a program on your own that demonstrates the use of each of the data types presented in this chapter. Your sample program should also show how you can interpret data values differently, depending on the instructions or HLA Standard Library routines you use to operate on that data. Your sample program should also demonstrate conversions, logical operations, sign and zero extension, and packing or unpacking a packed data type (in other words, your program should demonstrate your understanding of the other components of this laboratory exercise). Include the source code, sample output, and a description of your sample program in your lab report.