C & LabWindows Fundamentals

Phys 4051
Kurt Wick
10/19/00

Table of Contents

Foreword

The first part of this handout was originally used to familiarize students with DOS, C and the QuickC compiler. Starting with winter quarter 1998, the DOS operating system was replaced with WindowsNT 4.0 and the QuickC compiler was replaced with National Instrument's LabWindows C-compiler. Therefore, you will find sections in the first parts of this manual outdated while others are still relevant. What follows is a brief list of the chapters in the first part and how they relate to the course.

Chapter 1 covers DOS and computer hardware. If you are unfamiliar with computers it may be useful to glance at the hardware section but you may skip the rest of chapter 1 about DOS entirely. I have included it as a reference because there are times when you may have to fall back on DOS, for example if you should work on an older machine, when you are setting up a computer or if you have a "sick" machine.

Chapter 2 covers various computer languages and explains some basic concepts about compilers. It will be helpful if you glance briefly at it.

Chapters 3 through 11 cover standard ANSI C concepts. You should read and understand chapters 3, 4, 5, 6, 7, 8 and 11 and you may want to glance at chapters 9 and 10 to get a general idea.

Acknowledgments

My thanks to Prof. Weyhmann and Jon Huber for their suggestions, proofreading and for providing various figures.

Kurt Wick 12/2/97

C and LabWindows 00b.doc 11/2/2000 5:52 PM

Previous Foreword

This handout provides an overview of the IBM disk operating system and of the C programming language. It is intended for people with little or no prior knowledge of IBM personal computers or the C programming language.

Students in the past have commented that the section on C programming is incomplete and that it should be expanded. We emphasize that this manual is NOT meant to be a complete reference covering all aspects of C; it is instead meant as an introduction to C programming, explaining the most often used expressions. The reason for not writing a more extensive C manual is that plenty of good (and not-so-good) books covering the C language already exist. (A list of the better ones is given in chapter 13.) Most of these are either very brief and are intended as a reference for people familiar with the subject, or they are written for the novice and are usually so large that important information is lost in details.

When learning the language, do not try memorize details about it; instead, learn how to find the corresponding information using the "Help" utility (see section 2.2 for details).

Acknowledgments

Prof. Keith Ruddick, Prof. Earl Peterson and Philip Johnson contributed many ideas and put in a lot of effort to make the manual more understandable and readable. Thanks also to James Flaten for his rigorous proofreading skills.

Copyrights

PART 1:

C AND DOS

1. DOS

1.1. Introduction: Operating Systems

As soon as the IBM PC is turned on, it begins its "boot" process; the computer hardware such as memory and disk drives are tested, software is loaded and various messages appear on the monitor. After 30 seconds, provided no hardware failures were found, a blinking cursor after an > prompt will appear on the monitor. (for example: c:>) The ">" indicates that you are now in the DOS (**D**isk **O**perating **S**ystem) environment.

Though all our lab computers use the DOS operating system, there are other operating systems; UNIX, OS\2, and VMS are some of the most common ones. Each operating system is a set of software instructions that specify and control how a computer interacts with the hardware that it is connected to, such as monitors, keyboards, mouse, floppy- hard- and tape drives, printers etc. For example, an operating system ensures that when you enter the proper command the contents of a data file will be displayed on a monitor. Though this may seem a trivial process, it involves many tasks such as positioning each letter on the screen and monitoring the need for inserting a line feed or scrolling the entire screen. Also, programs that are executed will repeatedly call the operating system and have it perform various tasks.

Some operating systems (UNIX and VMS) are primarily designed for main-frame computers; they allow multiple users on one computer to execute programs simultaneously (multi-user/multitask operating systems). Other systems are more suitable for microcomputers such as the IBM PC where usually only person uses a computer at any given time, though the user may run various programs simultaneously (single user/single task system such as DOS or single user/multitask system such as OS\2 and Windows). Theoretically, any operating system can be installed on any computer but operating systems that can handle many users and many tasks at a given time also require much larger and much faster memory and memory storage devices causing a computer system to become much more expensive. For these reasons, currently almost all IBM microcomputers use a version of the DOS operating system. However, the DOS operating system was designed a long time ago (16 years); by today's computer standards it is outdated and actually limits the performance of today's microcomputers. Therefore, it is very likely that either a completely new type of DOS will emerge or that microcomputers will switch in the next few years to a different operating system, with UNIX and OS\2 currently the best choices.

Operating systems also differ in their command structures (i.e. the instruction set that they use). This can be cumbersome for the user because some of the most commonly used commands are often similar in different operating systems but not identical. Switching from one operating system to another usually guarantees that programs developed in one system will not work in another one because the program will try to access the command structures of the previous operating system in the new operating system. (For example, a file stored on a disk using the Macintosh OS operating system can not be read by an IBM microcomputer using DOS.) Clearly, a universally agreed upon operating system would solve these problems (and UNIX is trying to emerge as that) but for now we have to live with different operating systems.

1.2. Computer Hardware Overview

1.2.1. Memory and Disk Drives

The computer can store information (data) in three different memory devices: ROM, RAM chips and disk drives.
-- ROM (Read Only Memory) chips store information that was "burned" into the chips in the factory; it can not be altered by the user (non-volatile memory). The information in these chips is used when the computer boots up.
-- RAM (Random Access Memory) chips store temporary information; all information stored in RAM is lost when the computer is turned off (volatile memory). After a boot up, any RAM memory that is not directly used by the operating system is empty. Because data in RAM can be very quickly accessed and updated, information such as programs and data files is loaded into these locations. Our lab computers contain 32 MB of RAM (1 MB is about 10^6 Bytes).
-- Disk Drives store information on a magnetic medium, the disk. Data is usually loaded from a disk drive to RAM, manipulated and then saved again on a disk.

There are two types of disk drives: hard drives (or fixed disk drives) and floppy drives. The hard disk is located inside the computer; only a little LED can be seen blinking on the lower right hand corner of the computer when the hard drive is in use. The main advantages of hard drives are that they are much faster and can store much more information than floppy drives. (The lab hard drives can store between 1 and 3 GB.) The drawback with hard drives is that they cannot be removed and stored or locked away. Hence, any information on a hard drive is available to anybody with access to the computer and it can be read, altered or erased by anyone. In addition, in case of a hard drive failure (which is not uncommon) all information stored on the drive is lost.

Unlike a hard drive, the disk in a floppy drive can be easily removed. Floppy drives are usually referred to by the physical size of the floppy disk that they use (either: 8", 5 1/4" or 3 1/2") and the maximum storage capability. Over the last 10 years floppy disks have shrunk in size and greatly increased in storage capability. The very first floppy disks, the 8" disks, had a capacity of 256 KB and are very seldom found today. 5 1/4" disks (usually sold in black covers, see picture below) can store either 360 KB or 1.2 MB, depending on whether they are sold as double-density or high-density disks. Our computers have a 5 1/4" drive that can read and write to 360 KB (double density) disks.

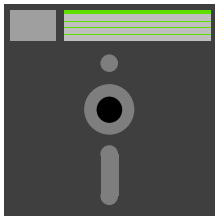


Figure 1.1. 5 1/4" Disk: unfortunately they are still used today (probably because they are very cheap). They also make mediocre Frisbees.

Finally, quickly becoming the most popular disk today, (probably because they were designed to fit in a shirt pocket) are the 3 1/2" disks (usually sold in blue). They are available with 720 KB capability (double-density) or 1.44 MB (high-density). (To determine whether you are using a 1.44 MB or 720 KB disk, count the number of 1/4" square holes in the disk; 1.44 MB disk have 2 holes, 720 KB disk have only one.)

Figure 1.2. 3 1/2" disks: the disk on the left is a 720 KB disk while the one on the right holds 1.44 MB.

Our computers have a 3 1/2" drive that can read and write to 1.44 MB and 720 KB disks. In the labs, we will be using mostly 3 1/2", 1.44 MB disks.

Other memory storage devices not mentioned previously (or used in our lab) are tape drives, CD ROM drives and other optical drives such as WORM drives (write once, read many times).

1.2.2. Accessing and Changing Disk Drives

To access a specific disk drive, a unique identification letter has been assigned to each disk drive. DOS specifies that the first two floppy drives must be A: and B:. In our case, all lab computers have drive A: assigned to the 3 1/2" drive and drive B: is the 5 1/4" drive. Computers elsewhere may have a different assignments for A: and B: DOS requires that the first hard drive always be drive C: This is true for any DOS computer with a hard drive.

Here is a list of the drive assignments:

A: 3 1/2" Floppy Drive (1.44 MB)
B: 5 1/4" Floppy Drive (360 KB)
C: Internal Hard Drive (20 MB)

1.2.3. Default Drive

When working on the computer you will find that for a certain task you will be using one disk drive much more often than any of the others. To speed things up, it is a good idea to declare this disk drive your "default drive". This is accomplished by typing the drive letter, a colon and a carriage return (i.e. hit the ENTER key). For example, if you need to use drive A, enter:

```
A:
```

The computer will respond either with:

```
A:\>
```

indicating that your default drive is now A, or with

```
Not ready reading Drive A
Abort, Retry, Fail?
```

In this case the computer attempted to make A the default drive but was not able to read or write to the default drive because there was probably no disk in drive A:. Put a floppy disk in drive A: and type R for retry.

1.2.4. Directories

Now that you know how to select a disk drive, you want to be able to find out what has been stored on the disk in your present default drive. Enter:

```
DIR
```

(note, you can use upper and/or lower case letters for any DOS commands, they are case-insensitive). You should see a listing of all the files in your root directory, including the date and time that each file was either created or last modified.

Given the size of today's hard disks, it is possible to store a few thousand files on a hard disk. Though it would be no problem for the computer to find a particular file, it would be very annoying for the user to read through the entire directory. Therefore, files are usually organized in directories and subdirectories in an upside-down tree structure (the root is on top and the branches on the bottom). For example, some of the directories on our hard drive C: are organized in the following way:

```
Root Directory              (contains some system files)
    |
    |---DOS           (contains dos utilities)
    |
    |---QC25          (only contains C subdirectories)
    |       |----  BIN        (contains C compiler files)
    |       |----  INCLUDE    (contains all C "include" files)
    |       |----  LIB        (contains C libraries)
    |        ----  SAMPLES    (contains C programs
    |
    |----MAP          (contains trackball utilities)
    |
    (any other directories)
```

Each disk must have a root directory. A root directory may contain any number of files and subdirectories and each subdirectory may contain files and further subdirectories and so on. To display all the directories on your harddrive enter:

```
TREE
```

The collection of subdirectories leading to a file is often referred to as a '**path**' For example, the path to a file in subdirectory 'sample' would be: \qc25\samples.

1.2.5. Moving to Various Directories

As an example, consider our hard drive. To move to the root directory type:

```
c:
cd \
```

The first command selects drive C: as the default drive; the second command, which is short for "**c**hange **d**irectory", puts us in the root directory. While each directory has a unique name, the root directory is denoted by the backslash symbol \. The computer will reply with:

```
c:\>
```

which means a) that our default drive is c: and b) that we are currently in the root directory of the default drive. Typing dir again we notice that some of the filenames end with <DIR>, indicating subdirectories, such as:

```
DOS <DIR>   9/26/90    3:05p
QC25    <DIR>   11/13/90   2:55p
```

To move to a subdirectory enter 'cd' again and the entire path name to the subdirectory. For example, typing:

```
cd \qc25\samples
```

will move us from the root directory through subdirectory 'QC2' into sub-subdirectory 'SAMPLES'. Note, while the very first backslash in the path statement always indicates that the path begins at the root directory, the subsequent backslashes are used merely as separators.

If you want to move from a subdirectory to the directory directly above, enter:

```
cd ..
```

For example, if your current default directory is qc2\samples, entering cd .. will change the current default directory to qc2. Clearly, you could have accomplished the same thing by entering: cd \qc25.

1.2.6. Creating Subdirectories

You can create your own subdirectory on the hard disk or any floppy disk. Though it is not very important to organize your files on a floppy disk, (it depends on how neat or organized you would like to be) you should put your files on the hard disk in your own subdirectory. Generally you should not store your personal files (programs) in the root directory on the hard disk because a) it is reserved for system files and b) it clutters up very fast with files from other people. Also don't go crazy with subdirectory levels; you rarely need to create more than two or three levels of subdirectories for a given topic.

To create a subdirectory select a unique name with 8 or fewer letters and a directory under which you will create your subdirectory. For example, if you want to create your own subdirectory named ANYJOE under subdirectory \qc25\samples\ enter:

```
md \qc25\samples\anyjoe
```

where md stands for "**m**ake **d**irectory." You can create any number of subdirectories as long as the directory under which the subdirectory will reside already exists; otherwise you must create it first. For example if you want to create directory ANYMARY under STUDENTS under the root directory, you probably have to create directory STUDENTS first and then directory ANYMARY.

1.2.7. Deleting Directories

When you no longer need a directory you may want to delete it. (Thou mayest never delete a directory that thou didst not create!!!) To delete a directory you must first delete all files in that directory plus any directories below that directory. (Read the section 1.3.4. on DEL or ERASE on how to delete files.) If there are no more files or subdirectories in the directory then you can enter 'rd', which is short for "**r**emove **d**irectory", and the directory name. For example, if you want to remove the previously created directory, STUDENTS, use:

```
rd \students\anymary
rd \students
```

Omitting the first line will result in the following error message:

```
Invalid path, not directory,
or directory not empty.
```

1.2.8. DOS Files and Filenames

Files store data, instructions or a combination of both and they can be identified by their filename and extension. DOS filenames can have 8 or fewer letters; a period is used to separate the filename from an (optional) 3 (or fewer) letter extension (for example, file 'test.dat' has filename 'test' and extension 'dat'). Filenames are usually unique because they identify a unique file; extensions usually indicate the type of file and follow some conventions.

Files that contain instructions are referred to as executable files, command files or batch files. To distinguish these files from all other files, these files have an extension of .EXE, .COM or .BAT. Only a file with such an extension can be executed directly from DOS by typing its filename and hitting the return key. For example, when you enter: QC the QC.BAT file will be executed and it will invoke the QuickC compiler program. Once a program begins to execute you are no longer in DOS and you may only do whatever the particular program that you are running allows you to do; it is usually up to that program to return you to the DOS environment.

Data files do not have a unique or reserved extension and hence any reasonable three letter extension can be used. As a matter of convenience, text files usually have extension .TXT or .DOC, numerical data files often use the extension .DAT and C source code files use extension .C

When naming your own files choose unique filenames, don't use names such HOMEWORK, PROBLEM1 etc. Be creative, but use standard extensions (such as .DAT for data files, .C for C programs.) If a file is copied to a directory where a file with the same name and extension already exists, then the file that has been residing in that directory will be automatically overwritten by the new file.

When working on a project, it is a good idea to keep the filenames to six letters and use the remaining two letters for version numbers, such as MUONS01.C, MUONS02.C, MUONS03.C, MUONS22.C, MUONS23.C etc. Save (copy) the file with a new version number whenever a reasonable change has been made to it. There are two reasons for that: first, in case of a hardware failure, you will have a backup copy if you saved your file to a floppy disk. Secondly, to quote W.S, "striving to better, oft we mar what's well". In other words, program upgrades can sometimes result in "downgrades"; hence, keep some of your previous versions.

1.2.9. Summary of Directory and Disk Drive Commands

To change default disk drive, enter drive letter and colon:	example: C:
To change a directory enter CD and path:	example: cd \qc25\bin
To create a directory enter MD and path:	example: md \qc25\test
To delete a directory enter RD and path:	example: rd \garbage

1.3. Some DOS Commands

1.3.1. FORMAT:

Newly purchased disks are usually not formatted; therefore, prior to its first use, every floppy disk must be "formatted" or "initialized". The formatting process erases and verifies every sector on the floppy disk; control information and various indexes that DOS uses to organize and store data are written to the floppy disk. The most basic format command is:

FORMAT [drive]

where drive stands for the identification letter of the floppy disk drive in which the disk to be formatted has been placed. For example to format your 3 1/2", 1.44 MB floppy disk, put the disk into drive A; and type: format a: and follow the on-screen instructions.

Though formatting is a very efficient way to get rid of data no longer needed, always remember that formatting erases the entire disk content! Always use the format command with caution. Typing the wrong drive letter can have disastrous consequences!

The format command will always format a disk to the maximum capability of the disk drive. Hence, if you try to format a 3 1/2", 720KB in our 3 1/2", 1.44MB floppy drive, you would get an "Invalid Media or Bad Track" error message; to format a 720KB disk use:

```
format a: /F:720
```

1.3.2. DIR:

The dir command lists the files in a directory. The most general form of this command is:

DIR [drive][path][filename] /W /P

where: [drive] specifies the disk drive on which the directory to be listed is located ([drive] is optional if it is the same as the default drive)

[path] is path to the directory to be listed (again, [path] is optional if directory is already the default directory)

if [filename] is omitted, all files in that directory will be listed, if [filename] is used, only the file matching that name will be displayed, if it exists.

/W is optional and lists files in the directory horizontally (useful if there are many files in a directory)

/P is optional; pauses when the listing reaches the bottom of the screen.

Some command such as DIR and COPY, that require a filename can also be used with the two "wildcard" characters "*" and "?". Replacing a character in a filename or extension with one or more "?" means that any character can occupy that space. A "*" in a filename means that any character can occupy that space and all the following character spaces in the filename or extension.

Examples:

For the following examples assume that our current default drive is C and our current default directory is DOS (i.e. our complete path would be C:\DOS)	
DIR	lists all files and subdirectories in directory \DOS on drive C:
dir A:	lists all files and directories in the root directory of disk in drive A:
Dir \qc25\samples	lists all files and subdirectories in directory c:\qc25\samples
DIR /w	lists all files and subdirectories in directory \DOS on drive C: horizontally
DIR *.com	lists all files and subdir. in directory \DOS on drive C: with extension .COM
DIR *.	lists all subdir. (since they generally have no extension) in the root directory

1.3.3. COPY:

The copy command copies one or more files from one directory to a different directory on the same or a different drive. The most general usage is:

copy [drive1][path1]filename1 [drive2][path2][filename2]

where:

[drive1] and [path1] are the drive and path of the directory where the source file, file1 is located. [drive1] and [path1] are optional if you are already in the directory where file1 is located.

filename1 is the complete name (filename + extension) of the file to be copied and is mandatory. filename1 can be used in combination with wildcard characters.

[drive2] [path2] are the drive and path of the directory to which the source file is to be copied. Again, if you are already located in the directory to which the file is to be copied, the path and drive specifications are optional.

[filename2] is the name of the copy of the file. If it is specified, it can be different than the original name; if it is omitted, the name of the original file will be used.

Examples:

copy test09.c finalkw.c	copies file test09.c to a new file named finalkw.c in the same directory
copy test09.c a:	copies file test09.c to floppy drive A:, maintaining the original name
copy muons2?.c a:	copies all files beginning with name muons2 and any one character and with extension c to drive a:, maintains the original filenames
copy m* c:\garbage	copies all files beginning with the letter 'm' to subdirectory garbage on drive C:

1.3.4. DEL or ERASE

The DEL or ERASE command deletes one or more files from a directory.

ERASE [drive][path]filename

where:

[drive] and [path] specify the drive and path where the file to be deleted is located and are optional if the file is located in the current default directory and drive.

The complete filename (filename and extension) is required. Wildcard characters such as "*" and "?" can be used.

Note: DEL and ERASE can be used interchangeably.

Examples:

del test.*	deletes all files in current directory with name test and any extension
del a:*.*	deletes all files in the root directory on drive a:
del test0?.c	deletes all files in current directory beginning with name test0 and any one character plus extension .c

1.3.5. TYPE

Type displays the content of an ASCII or text file.

TYPE [drive][path]filename

where:

[drive] and [path] specify the drive and path where the file to be deleted is located and are optional if the file is located in the current default directory and drive.

The complete filename (filename and extension) is required. Wildcard characters are not allowed.

If the file specified is not an ASCII file, DOS will attempt to display the file content until it finds an end--of-file marker. Usually, output from such a file results in the screen filling up with garbage and the loudspeaker beeping.

To print the content of a file on a printer, use:
TYPE [drive][path]filename > LPT1

Examples:

type a:readme.txt	displays content of file readme.txt on screen
type a:readme.txt > lpt1	prints content of file readme.txt on printer

1.4. Hotkeys

DOS has some specials keys reserved to interrupt a program or to reboot the computer. Usually they are a combination of keys which have to be pressed at the same time. For example CTRL-S means that you hold down the CTRL key at the same time as the S key. Here is a list of some of the keys:

- **CTRL-S**: will stop output to the screen temporarily; hitting any key will resume the program (useful in combination with the DIR command).
- **CTRL-Q**: resumes after a CTRL-S was encountered
- **CTRL-C**: will terminate a program or process (if CTRL-C doesn't work try CTRL-BREAK)
- **CTRL-ALT-DEL**: stops every process and reboots the entire machine; all data not saved previously is lost. Using CTRL-ALT-DEL is almost identical to turning the computer off and then on again.

2. EDITORS & COMPILERS

2.1. Introduction: Computer Languages

Last fall, you designed and built basic digital circuits such as counters, adders, comparators, etc. Suppose, that you were to build every important basic digital circuit on a single circuit board, and designed it so that the outputs from every circuit could be tied together, and so that each individual circuit could be activated by a combination of enable lines. (For example, with only 8 enable lines and some logic gates, you could turn on/off 256 individual circuits.) Then, if you should manage to put everything inside a 2x2" chip, you would have invented a CPU (Central Processing Unit), the heart (or brain) of every computer.

Each combination of 0 and 1 that activates or enables one specific circuit or function on the CPU is called an instruction. The CPU instructions perform very basic tasks such as the loading and storing of a number, comparison of two numbers, addition, subtraction, multiplication and division of integer numbers; furthermore, conditional branching instructions inform the CPU to skip a given number of instructions if certain conditions hold true.

The entire set of these instructions is referred to as machine language. It is the only language that the CPU understands. Sets of instructions, or programs, are executed most efficiently if they are directly written in machine language but this language is also the most cumbersome for humans to work with. For example, to add two numbers, the Intel 8088 CPU needs the following instructions: 0000 1000 0000 0001 0100 1000 0000 0001 1111 1111 1111 1111 0000 0000. These instructions can be made somewhat more readable by converting them from their binary base to a hexadecimal base; this would make the above instructions read: 81 C1 FF 00.

Clearly, programming in machine language is tedious. First, before an instruction can be used it must be looked up in some table. Second, reading even a short program would be very tiring because, all you would see is: B9 0A 00 B8 00 00 C1 C8 E2 FC 90.... Clearly, what is needed is some sort of translator converting a more intuitive symbol (such as ADD or simply +) into the corresponding machine language code (81C1). Many such translators are available and they are called compilers.

A compiler that follows most closely the machine language instructions is an assembly language compiler. These compilers assign every machine language instruction a three letter code, also known as op-code, making the writing and reading of a program much easier. For example, the above instruction 81 C1 FF 00 can now be entered as: ADD CX, 00FF and after some practice, you'll immediately recognize that this means: add value FF (or decimal 255) to some register CX.

Note that we said that the machine language instructions are very basic. For example, you will not find a machine language instruction, or its corresponding assembly language op-code, for calculating the square root, the sin or logarithm of a number. You will not even find that the most basic mathematical instructions (for example addition) are capable of dealing with real numbers. Hence, if you are working with real numbers in assembly or in machine language, you would need to write a short program that would convert your numbers to integers, then perform the mathematical operation and finally convert the integers back to real numbers. To write such a program or subroutine (i.e. part of a program) is not a trivial task.

Compilers exist that are capable of replacing one complex instruction by a whole set of machine language instructions. For example, entering 'sin(x)' instructs the compiler to replace this instruction with a machine language subroutine that calculates sin(x). Clearly, the number and types of instructions that such a compiler can handle is theoretically unlimited. Each set of instructions is called a computer language and is often optimized for a particular programming task (such as business, scientific or artificial intelligence oriented programming).

Name	Application	Year	Comments
COBOL	Business	1959	Used in business applications; ANSI Standard
RPG	Business	1962	Used for producing business reports
FORTRAN	Scientific	1954	Still (one) of the most popular scientific languages
ALGOL 58	Scientific	1958	Useful in mathematical problem solving
APL	Scientific	1962	Scientific applications
BASIC	General	1964	Used on micro computers
Pascal	General	1971	Structured language, used on micro computers
Ada	Scientific	1979	
MODULA-2	Scientific	1979	Multiprocessing language
C	System	1975	ANSI Standard
LISP	Special	1960	Used in artificial intelligence
SNOBOL 4	Special	1963	
PROLOG	Special	1970	Artificial intelligence
FORTH	General	1974	
Smalltalk 80	Special	1980	Object oriented language for graphical windows

Table of currently used high level languages.

Computer languages can be grouped according to 'high-level' or 'low-level' languages. Low-level languages follow closely the instruction set of the original machine language. They depend on the specific CPU used and they are not very transportable (i.e. they don't work with different CPUs). They are executed very efficiently by the computer but are less efficient to the programmer. The quality (speed, reliability) depends almost entirely on the skills of the programmer. Assembly language is an example of a low level language. High level languages have very little in common with the basic machine language instruction sets. They call on large libraries of subroutines to execute very complex instructions. With a suitable set of libraries, programs can be used on different computers. Very complex tasks can quickly be programmed in these languages but the execution of the actual program can be slow and depends to a large degree on the capabilities of the actual compiler. The most common high level languages are: Pascal, FORTRAN and BASIC.

2.1.1. Why Use C?

C is becoming ever more popular. Probably the most important reason for that is, aside from COBOL, C is the only computer language for which the National Bureau of Standards has established a standard (ANSI C); unlike other languages, only one official version of C exists. Hence, programs developed on your micro computer can be used on any other computer with a C compiler.

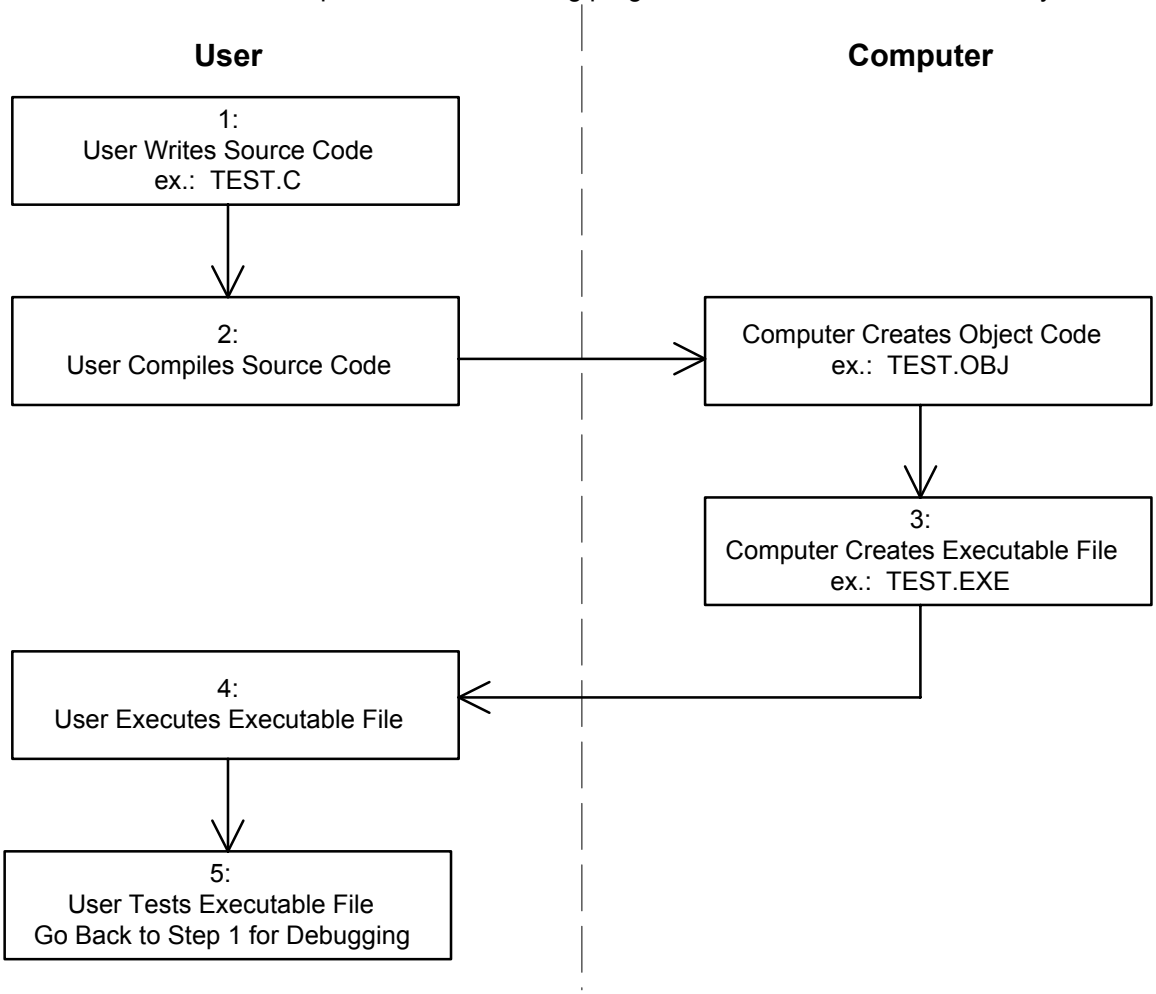
C is neither simply high-level nor low-level; it has features of both. It allows access to some very low-level language capabilities giving you much greater (and potentially much more dangerous) control over the actual computer; at the same time, you have access to standard high-level language features, such as for example, high-level mathematical functions. Most high level language compilers have very extensive error check routines which greatly limit your freedom in programming. C compilers allow for much more freedom in programming expressions and hence, occasionally allow you to make very grave mistakes.

C programs are much more readable than assembly language programs. If you work in a group on a common programming project, you'll quickly find that reading other people's programs can be very difficult. Hence, you will greatly appreciate any language feature that makes reading other people's (and your own) program easier.

2.2. Compiler Overview

Though we have talked about different computer languages it is the purpose of this chapter to describe how a computer program is compiled. We will discuss our specific C compiler, Microsoft QuickC, version 2.51, later; this first section applies to most compilers.

There are 5 distinct steps involved in creating programs. This is shown schematically below.



The first step in creating a program always requires the writing of the source code. The source code is a plain text file containing all your programming instructions. To create the source code you use a text editor or word processor. We will be using the text editor that is included with the QuickC compiler program though you could use any text editor that you prefer. Remember, the source code is just a plain text file. This means, at this stage the compiler does not care at all what you are writing; as far as the compiler is concerned, you might as well be writing a letter to your grandma.

In the second step, the compiler examines your source code very carefully and tries to make sense out of everything that it encounters in the source code file. If the compiler encounters any instructions that are not part of the C language or that it does not recognize, it will print an error message and abort the compilation and return you to your editor. Correct your mistakes and then compile again. If your program is free of syntax errors, the compiler automatically writes an object file (extension .obj) to your (hard) disk; the object file is a non executable binary file that is needed later to create an executable file. Note, though the compiler checks for mistakes in your program it can not check whether your program will work correctly. Try to think of the compiler as an English teacher; the teacher can make sure that the spelling and grammar will be correct but cannot necessarily say whether the statement being made is correct.

In the third step, the compiler combines the object file with various libraries and creates an executable file i.e. your final program. In this step, C statements are replaced with corresponding machine language statements. Remember from chapter 1 that executable files have extension .EXE

Fourth, you are able to "run" or execute your program. While working on a program you will run your program from within the QuickC environment; this way when the program terminates it will return you automatically to the QuickC editor allowing you to make changes easily. Once your program is perfect, you can run it directly from DOS from any IBM compatible computer.

As a (temporary) last step, if your program does not behave exactly the way you anticipated, you need to fix it or to "debug" it. You will return to the editor where there are various tools at your disposal to monitor your program while it is running. Don't try to rely too much on the old computer saying that "any undocumented feature is a bug and hence, any documented bug a planned feature."

3. INTRODUCTION TO C

Every C program must at least contain the following elements:

```
main( )
        {
        }
```

Generally, a full size C program may consist of the following elements:

a) Comment Statements
b) Include Files
c) Define Statements
d) Function Declarations
e) Global Variable Declarations
f) Main Program
g) Additional Function Definitions

A sample program containing all of these elements is listed below.

```
/* This program doesn't do anything   */          /* COMMENT STATEMENTS     */
/* though it will run                 */
/* Version 1.0 KW 1/1/90              */

#include <stdio.h>                                                          /*
INCLUDE FILES                         */
#include <physics.h>

#define SOMECONSTANT 12                           /* DEFINE STATEMENTS      */
#define SOMEOTHERCONST 3.14158

double My_first_function(short value, float other_value);  /* FUNCTION DECLARATIONS    */
float Mysecondfunction(short x);

unsigned short some_number;                       /* GLOBAL VARIABLES       */

short main()                                      /* MAIN PROGRAM           */
        {
        short a, b, any;
        short x = 5;
        short y = 7;
        float c = .33;
        double y_total;
        a = x * y;
        y_total = My_first_function( x, c);
        /* more statements */
        }

double My_first_function(short value, float other_value)   /* FUNCTION DEFINITIONS    */
        {
```

```
        double ax;
        ax = some_number * somevalue / other_value;
        return( ax );
        }

float Mysecondfunction( short x )
        {
        float a;
        a = SOMEOTHERCONSTANT*x;
        return ( a );
        }
```

Except for the comment statements, keep the elements in the order shown above. Note two important C conventions:

1: unlike DOS, C is case sensitive. For example, if you declare the two variables, x and X, C will assume that they are distinct variables!

2: C ignores blank spaces and carriage returns; the only reason we use them in our source code is to make the code more readable.

Finally, keep in mind that the program execution always starts at 'main'.

3.1. Comment Statements

'/*' and '*/' are C comment statements. Comments are included in a program to make it understandable to other people and to the programmer him/herself.

C comments can be written like this:

```
/* This is a comment */
/* It will be ignored by the compiler */

/* This is also a comment
   Note: you don't necessarily need an ending comment
   statement at the end of every line.  You only
   need it at the end of each comment!                  */
```

Remember that everything between the beginning comment statement '/*' and the ending one '*/' will be ignored by C. Hence, if you should ever forget to enter the ending comment statement, then C will consider everything else in your program as a comment, including the program itself!

A different form of comment statements has lately been used in C. Instead of using the standard convention, some people prefer to use double slashes, '//', a convention borrowed from C++.

```
/* standard ANSI C comment */
// C++ comment statement - note no double slashes at the end of the statements
```

There are two differences between these two styles. First, since the double slash method was borrowed from C++, it is not ANSI C compatible and it may or may not work with some C compilers. (It will work with the MS QuickC compiler used in the lab.) Second, the C++ version does not require an ending comment statement; it instead assumes that everything to the right of the statement is a comment and it assumes that the line following the comment statement is not a comment.

3.2. Include Statements & Include Files

Directly following the introductory comment statements are the include statements. Include statements call include files (sometimes also referred to as "header files") which are text files containing information about function declarations, constants and structures. When your program is compiled, the information in the specified include file is 'pasted' directly into your program.

```
/* Examples:      */
#include <math.h>                                                    /* incl. file
with math. function info                    */
#include <stdio.h>                                                   /* include file
with I/O info      */
#include <time.h>                                                    /* include file
with time funct. info                       */
#include <graph.h>                          /* include file with graphics info      */
#include <myconst.h>                        /* see example directly below           */
```

Here is a very simple example which illustrates the purpose of include files. Assume that we were to write programs which depend on predefined physical constants, such as h, c, G, k etc. Instead of looking the values up and entering them directly into each of our programs, we could write a short text (i.e. ASCII) file using the QuickC editor. Something like this:

```
/* SI Constants                                                                 */
/* File Name:  myconst.h                                                        */
#define c 3.0e8             /* this statement defines the constant 'c'          */
                            /* see section I. c) for more information on 'define' statements. */
#define G 6.67e-11          /* this defines the constant 'G'                    */
#define k 1.38e23           /* this defines the constant 'k'                    */
```

Next, we save this file in the include subdirectory as "myconst.h" (Note, the extension ".h" is used to distinguish include files from C programs which have extension .c). If we enter in any of our future programs:

```
#include <myconst.h>
```

then all the above defined constants can be used directly in such a program and they do not have to be declared or entered again. In other words, when your program compiles, the entire content of 'myconst.h' is directly pasted or 'included' in the program.

The most commonly used include files contain function declarations for predefined C functions. Each C function must be declared somewhere, i.e. the function's name, its type and the arguments must be declared before it can be used. In our programs, we will be using numerous predefined C functions such as sin(), printf(), scanf() etc. Since these functions must be declared we should type their often lengthy and cumbersome declarations in our programs. Lucky for us, someone has already entered all predefined function declarations in various include files. Hence, using an include statement in our program which refers to a particular include file is identical to typing all the statements in the include file directly into our program.

The declarations from related functions are usually grouped together. For example, the declarations for all predefined math functions, such as sin(), exp(), rand(), log() etc., are stored in the file 'math.h'. To find out which include file is needed for a particular function, read the summary help screen (i.e. move the cursor to the function statement in your program and press f1). Note, if you forget to enter the corresponding include file for a function, the program may still compile without error messages but it may not work; hence, always check that each predefined C function has its corresponding include file declared!

Final Note: No semicolon follows the include statements.

3.3. Define Statements

Define statements declare constants. Constants are 'variables' that will not change during the program execution. Define statements can be stored in a separate include file which is useful if there is a large number of them. You don't need to use capital letters for the constant names, but it is a good (and very common) practice and helps to distinguish them from variables.

Three very common mistakes with define statements are:
1) to include an equal sign between the name and its declaration,
2) to use a semicolon at the end of the statement,
3) to use type definitions, such as short, float etc. with define statements.

```
#define PI = 3.1416                    /* W R O N G ! ! !                */
#define PI 3.1416;                                                        /* W R O N G
! ! !        */
#define PI 3.1416                                                         /* R I G H T
           */
#define float PI 3.1416                /* W R O N G                      */
```

3.4. Function Declarations

There are two different categories of functions that you will use in your C programs: predefined functions and functions that you will write.

Predefined functions were written by someone else and are stored in some function library on your hard disk, ready to be used by anyone who so desires. During the course we will provide you with predefined functions to simplify the programming needed in the lab.

These functions consist of a core of standard ANSI C functions that cover most basic programming tasks such as input and output, file maintenance and the basic mathematical functions. In addition to these functions, QuickC also provides some additional non-standard C functions, most of which are related to graphics. An abbreviated listing of most of the standard and non-standard functions is given in Table 1 or can be found using HELP/CONTENTS/LIBRARY FUNCTIONS. (A detailed description can be found by typing the function name and F1.) Do not panic when you look over the list of functions; of the hundred or so functions, you really need to know only about 10 functions but you should be aware that some of the other ones exist. Here is a table of functions that you should know:

Function Name	Include File	Description	Usage
printf()	<stdio.h>	Prints formatted data to screen	Display numbers and text
scanf()	<stdio.h>	Reads in formatted data.	To read in numbers
puts()	<stdio.h>	Prints character string to screen	Display text
gets()	<stdio.h>	Reads in a string (termin. by carriage return)	To read in text
kbhit()	<conio.h>	Detects if any key on keyboard has been hit	To stop a repeated process
inp()	<conio.h>	Reads a byte of data from a port	Computer interfacing
outp()	<conio.h>	Sends a byte of data to a port	Computer interfacing
malloc()	<stdlib.h>	Creates a data buffer	Arrays and data input
rand()	<stdlib.h>	Creates a random # between 0 and 32768	Simulations, Tests
pow()	<math.h>	Calculates x^y	
sqrt()	<math.h>	Calculates $\sqrt{x}$	

Sometimes you want to use a function that is not included in the library of predefined functions. If this is the case then you will have to **create your own function**.

Here is an example: you need to convert Kelvin to Fahrenheit. As a first alternative, you could simply enter the code for the conversion into your main program. The code for the conversion will simply become part of the main code. This approach is fine if you need to use the Kelvin Fahrenheit conversion in your program only once or twice. On the other hand, if you need the conversion often and at different places in your program, you are better off to write a function which, when it is called, will return the converted value.

In general, it is a good idea to package your code in functions. The decision when to write your own function and when to keep the code in 'main' depends on how often the function will be called and from how many different places in the program. Writing your own functions makes your program much more readable and 'debuggable'. In addition, if any of your functions turn out to be useful then they can be 'recycled' in other programs and over time you will build-up your own library of functions.

Without going into the details of the function syntax, (see section 3.7) it is important to understand that every C function that you intend to use in your program must be declared before it can be called. If you use predefined functions, then the function declarations have already been stored in include files; hence, at the beginning of the program, you must specify these include files (see section 3.2). If you write your own functions, place the function declarations before 'main'. For the exact syntax of the function declarations, see section 3.7 where we cover functions in more detail.

3.5. Global Variables

Any variables declared outside of 'main' will be global, i.e. they can be "seen" by 'main' and any other functions.

If you don't understand what local and global variables are, don't worry; they are explained in more detail in chapter 4.

3.6. main( )

Every C program needs a 'main' function. This is the place where the main body of your code is placed. A successful C program executes all the statements placed in main. Functions are only executed if they are called from within 'main'. Main is the only C function that doesn't need to be declared.

3.7. Functions

A function consists of three elements:

a) Function Declaration
b) Function Definition
c) Function Calls.

First let's see how you would use a **predefined function**. Below is a complete program using the 'printf' function.

```
/* Simple Stoneage Program */
#include <stdio.h>                          /* function declaration for 'printf" function */

main( )
    {
```

```
    printf( "Yabadabadoooo!!!!" );              /* function call */
    }
```

As you probably guessed, when you execute this program it will simply write "Yabadabadooo!!!!" (without the quotation marks) across your screen. Not very impressive but, nevertheless, a beginning.

We compare this program now with the three elements that a function consists of. The first element, the **function declaration** for 'printf', has already been declared (by someone else) in the include file 'stdio.h'. Instead of specifically declaring the 'printf' function we simply paste the entire 'stdio.h' file into the program by using the include statement. Not only does this free us from having to type in the exact and very cumbersome function declaration for 'printf', the 'stdio.h' file also contains many other useful function declarations of other frequently used functions.

The second element, the **function definition**, is missing entirely. Because we are using a predefined function, someone else has already defined it for us. There is no need for us to enter the 'printf' function definition again.

The third element, the **function call**, is executed by placing an argument, in this case "Yabadabadoooo!!!!", into the 'printf' function.

That's really all there is to using predefined functions. First you declare them by pasting the appropriate include file into your program and then you call the function with suitable arguments.

If you need to **create your own function** then you will need to use all three function elements.

a) To **declare** a function, use the following syntax:

type Name(type arg1, type arg2,....);

```
/* Examples:  Function Declarations                    */

short Factorial( short x);
float Binomial_Dist( short N, short n, float prob_true);
float Fahrenheit( float Kelvin );
```

Function declarations 'tell' the compiler the number and type of arguments to expect and the function type. Similar to variables, functions are always of a specific type because they return a value of that type. Always declare the functions before main. It is a good idea to capitalize the first letter of a function name to distinguish it from a variable name.

b) To **define** (i.e. write) a function, use:

```
type Name(type arg1, type arg2,....)
    {
    statements;
    return (value);
    }
```

```
/* Examples:  Function Definitions                                          */

float Fahrenheit( float Kelvin )
      {
      float temp;                          /* local variable               */
      temp = 1.8 *( Kelvin - 273) + 32.0;  /* conversion                   */
      return( temp );
      }

void Clear_Screen( void )                  /* this function just calls another function;  */
      {                                    /* no value is returned         */
      _clrscreen();
      return;
      }
```

Note, while the function declaration statement is followed by a semicolon, the function definition statement is not followed by a semicolon. (Confusing? Try to remember that main is a function (definition) and it is not followed by semicolon.)

c) Finally, to **call** a function use:

Name(arg1, arg2,....);

```
/* Example:  Calling Function Fahrenheit                                    */
/* Note:  This is a complete and working C program                          */

#include <stdio.h>                         /* needed for printf() statement */

float Fahrenheit( float Kelvin );          /* Function Declaration          */

main( )
      {
      float a = 20;                        /* local variables              */
      float temp_F;

      temp_F = Fahrenheit(a);              /* call function Fahrenheit     */

      printf("Temp is: %f\n",temp_F);      /* print it                     */
      }


float Fahrenheit( float Kelvin )           /* Function Definition          */
      {
      float temp;
      temp = 1.8 *( Kelvin - 273) + 32.0;  /* conversion                   */
      return( temp );
      }
```

As you can see, when you call a function you must not specify the type of the function or the type of the function arguments; you have already taken care of that in your declaration and definition statements.

Here are a few fundamental facts about functions.

Though a function may have any number of arguments, it can return **at most** only **one** item; the returned item can be a character, an integer, a floating point number or a pointer. Hence, you state in the function declaration what you expect the function to return. If your function returns nothing, which is perfectly fine, then declare it type 'void' and simply use 'return;' and omit the parentheses in the return statement.

```
void Print_it( char *it)          /*The function definition; note, the function is not */
    {                             /* expected to return anything.                      */
    printf( it );
    return;                       /* The function returns nothing.                     */
    }                             /*  Note the omission of the parentheses.            */
```

Function arguments do not have to be of the same type.

Finally, do not declare the function arguments again in your function definition.

```
float Joules( float ev)           /* Example CORRECT:                                  */
    {
    return( ev / 1.6e-19 );
    }

float Joules( float ev)           /* Example WRONG:                                    */
    {
    float ev;                     /*ERROR:  ev has already been declared!              */
    return( ev / 1.6e-19 );
    }
```

For some additional examples about functions see chapter 7, Loop Instructions.

4. VARIABLES

4.1. Type and Range of Variables

The following C variables will be used: (A similar table can be found in the QuickC help index under: Help/Contents/Data Types).

/* Type	Range	Bytes	
char	-128 to 127	1	
unsigned char	0 to 255	1	
short	-32768 to 32767	2	
unsigned short	0 to 65535	2	
long	-2147483648 to 2147483647	4	
unsigned long	0 to 4294967295	4	
float	3.4e+/-38	4	
double	1.7e+/-308	8	*/

In older versions of C, variables of type 'short' are also sometimes referred to as 'int'.

4.2. Declaration of Variables

Every variable has to be declared. The declaration consists of a type specification, the variable's name and an optional initialization value. Variable names can be as long as you like, though only the first 31 letters will be used to identify them. Note, the declaration alone does not initialize the variable to 0.

```
/* EXAMPLES:  Variable Declarations */
     short x, y, z;
     float a,                                /* remember C ignores carriage      */
           b,                                /* returns                          */
           c;
     short this_is_a_very_long_variable_name;

     float delta = 1e-6;                     /* declaration and initialization combined.  */

     x = y = z = 1;                          /* multiple assignments are OK      */
```

Inside a function, you may declare a variable only once. If you want to change the type of a variable you may assign a temporary new type to the variable through '**type-casting**.' To type-cast a variable precede the variable with the new type enclosed in parentheses. For example, in the above program, variable x was declared type 'short' (i.e. an integer); using an expression such as:

```
a = (float) x/3
```

will change the variable x for this equation temporarily to type 'float' (i.e. a real number). Type casting is used often in mathematical expressions and in function calls.

4.3. Scope of Variables

Depending on how (i.e. where)a variable is declared determines which parts of your program may use the variable.

For example, variables declared inside of a particular function definition may only be used within that particular function definition; if you refer to that variable from within another function (or main), you will get an error message: **Variable Not Declared**. Variables declared within a function have a meaning only within that function; they are created when the function is called and they are destroyed again when the function is exited. Variables which exist only in a function are called '**local**' i.e. they are local with respect to a particular function; sometimes, local variables are also referred to as 'being visible' only to a particular function.

On the other hand, if you declare a variable before 'main', that variable will be visible to all functions, including 'main'. Such variables are referred to as '**global**'. Hence, once a global variable has been declared in your program it can be referred to (without any additional declarations) in every function and 'main'; it is said that global variables are 'transparent' because they can be seen by every function.

If a function needs to use a local variable defined in another function, you have to pass the variable as function argument.

```
/* Example (This is Only Part of a Program) */

void First_function(void)
      {
      short j;                        /* j is local                                          */
      short i = 3;                    /* i is local                                          */
      j = Second_function( i );       /* i is passed as argument to function Second_function */
      return( void );                 /* i is still 3 but                                    */
      }                               /* j is now 15                                         */

short Second_function( short value)
      {
      short k = 5;                    /* 'k' is local to Second_function                     */
      return( value * k );
      }
```

Now consider writing a very short and simple program; all your code would be placed in 'main' and you would not write your own functions. The question is should you declare your variables as global (Example 1) or as local variables (Example 2)?

```
/* Example 1:  x is global */
short x;
main()
      {
      x = 21234/2;
      }
```

```
main()  /* Example 2:  x is local */
      {
      short x;
      x = 21234/2;
      }
```

As a matter of fact, both Example 1 and Example 2 would work and for simple programs it doesn't really matter. Nevertheless, Example 2 is better because you should always keep global variables to a minimum. This is particularly important if you write large programs which contain include files and your own functions.

Here are more complicated examples using global and local variables.

```
/* Example 3:  */

void My_Function( void );          /* Function Declaration                */

short temp;                        /* temp is global                      */

main()
      {
      temp = 412;
      printf("%d", temp);          /* prints temp                         */
      My_Function();               /* calls Function with arg. void       */
      printf("%d", temp);          /* prints temp                         */
      }

void My_Function( void )
      {
      temp = 3;
      return( void );
      }
```

```
/*  Example 4: */

void My_Function( void );          /* Function Declaration                */

main()
      {
      short temp;                  /* temp is now local                   */
      temp = 412;
      printf("%d", temp);          /* prints temp                         */
      My_Function();               /* Call Function                       */
      printf("%d", temp);          /* prints temp                         */
      }

void My_Function( void )
      {
      short temp;
      temp = 3;
      return( void );
      }
```

Note, the only difference between the two examples is, 'temp' is a global variable in Example 3 and a local variable in Example 4. After running both examples, the printout would be: ('printf' prints the value of the variable following the comma)

```
Example 3: 412 3
Example 4: 412 412
```

Explanation: In the first program, 'temp' is global and both main and My_Function can "see" 'temp'. Changing the value of 'temp' anywhere in the program affects the value of 'temp' everywhere else.

On the other hand, in the second program, 'temp' is local. Though the variable name 'temp' is being used in main and in My_Function, 'temp' is now two distinct variables, one local to main and the other local to My_Function. Since local variables can not "see" local variables belonging to other functions, changing the value of 'temp' in My_Function will not affect the value of 'temp' in another function.

When you write very long programs you often will run out of short and easy to type variable names that are distinct. If all your variables were global then you have to worry how assigning a value to a variable might affect your entire program somewhere else. (A worry every BASIC programmer is familiar with.) With local variables, a change in a variable will affect only that variable within that function. Furthermore, if you want to reuse any of your own functions in future programs it is nice to know that they can be called directly from your program without having to declare all kinds of global variables.

4.4. Reading and Printing Variables

One of the most important aspects of any program is being able to exchange information between the user and the computer. This is done through reading information into a program or by printing it out. The two most frequently used predefined C functions for these tasks are: 'printf( )' and 'scanf( ).' We will now examine these function in great detail. Keep in mind that it is not necessary to memorize all the little details and idosynchrasities; instead concentrate on what these functions are capable of and where to find information about them.

4.4.1. The 'printf( )' Function

We first explore the extremely versatile 'printf( )' function. To access this function you need to include the "<stdio.h>" file at the beginning in your program.

In its most simplest form this function will simply print a string, that has been enclosed in quotation marks, to the screen. The "Stone-age" program listed earlier is an example of this. A slightly more enhanced version is show below.

```
#include <stdio.h>                          /* function declaration for 'printf" function  */
main( )
     {
     printf( "Yabadabadoooo!!!!" );         /* function calls                              */
     printf( "man who knows does not speak" );
     printf( "man who speaks does not know.  Lao Tzu."):
     }
```

The output from this program will be:

```
Yabadabadoooo!!!!man who knows does not speakman who speaks does not know.  Lao
Tzu.
```

This is not very readable because the output from each printf( ) function call starts right where the previous one ended. The readability can be improved greatly by formatting the output. This is done using, what is referred to in C as "**escape sequences**", i.e. a backslash (\ ) followed by a letter.

While the complete list of escape sequences is given in Table 3 at the end of this booklet, here we will concentrate on the two most commonly used ones: the newline '\n' and the horizontal tab '\t'.

Here is a program that prints the first few lines of a work by Chaucer.

```
#include <stdio.h>                              /* function declaration for 'printf" function  */
main( )
    {
    printf("\tWhan that Aprill ");              /* function calls                              */
    printf("with his shoures soote\nThe droghte of March hath perced to the roote,\n");
    printf("And bathed every veyne in swich licour\nOf which vertu eng");
    printf("endered is the flour;\n");
    }
```

Note how the escape sequence characters can be inserted anywhere within a string. Here is the output from the program:

```
    Whan that Aprill with his shoures soote
The droghte of March hath perced to the roote,
And bathed every veyne in swich licour
Of which vertu engendered is the flour;
```

In addition to printing strings, 'printf( )' can also print numerical values assigned to a variable. To do so, a percentage sign followed by a **type specifier** is inserted into a string and the name of the variable is listed at the end of the string. When the function is executed, the percentage sign will be replaced by the numerical value assigned to the variable. The syntax for a typical printf( ) function call, for printing both strings and a variable is:

printf("string%type specifier string", variable name);

where one or both 'string's' may be omitted. The type specifier depends on the type of variable used; for a sensible printout, the variable type and the type specifier must match. Not matching them correctly is a very common mistake!

A more complete table of type specifiers is given in table 3 at the end. Here are the most common ones:

/* Variable Type	Type Specifier	Comments
char	c	
short (signed)	d	
unsigned short	u	
long (signed)	ld	
unsigned long	lu	
float	f	Decimal Notation (ex. 123.456)
float	e	Scientific Notation (ex. 1.2345e02)
double	lf (or le)	
character array (string)	s	
	x	Prints a variable in hex. notation. */

Though ultimately you will remember most of these type specifiers (or know where to look them up) there is a short a cut. Remember only one or two of the most common ones (probably 'e' for scientific notation for a variable type 'float') and then type cast (see section 4.2) all your variables to that type.

Study the program below and its output carefully:

```
#include <stdio.h>
main( )
      {
      short i = -1;
      long  k = 10000000;
      printf("Type 'short' variable: i = %d\n", i);                    /* First printf call        */
      printf("Wrong type specifier (i is signed short): i = %u\n", i); /* Second prinft call       */
      printf("Multiple Statement: i = %d\tk = %ld\n", i, k);            /* Third printf call        */
      printf("Type Casting (k to float) k = %e\n", (float)k);           /* Fourth printf call       */
      printf("%x\n", k );                                              /* Fifth printf call        */.
      }
```

The output from this program is:

```
Type 'short' variable: i = -1
Wrong type specifier (i is signed short): i = 65535
Multiple Statement: i = -1k = 10000000
Type Casting (k to float) k = 1.000000e+007
9680
```

Let's look at each function call in more detail.

Compare the first and the second 'printf( )' function call. Though they differ in their string, they both attempt to print the value assigned to the variable 'i' which is of type '(signed) short'. The second function call prints an incorrect value because the type specifier used refers to a variable of type unsigned short. Though the variable and its type specifier do not match, no error or warning will be generated by the C compiler! This can be very frustrating especially when one tries to print out results of lengthy calculations because one will automatically assume that an incorrect value was assigned to the variable; this in turn will lead to searching for errors in the program where there are none.

So far we have only printed one variable per function call. The third function call shows how to print multiple variables within a single string. In such a case, the value of the first variable replaces the first percentage sign and the second variable is assigned to the second percentage sign and so on. Also note the tab escape sequence that was used to improve the readability.

The fourth function call shows how to use type casting. Though 'k' is type long, it is converted temporary and printed as a type 'float" using scientific notation.

In the last call, the numerical value of k is printed in hexadecimal notation.

4.4.2. The 'scanf( )' Function

Now that you know how to print numerical values to the screen, you need to learn how to read such values from the keyboard into a program. The C function which does that is 'scanf( )'. Lucky for us, its syntax is very similar to that of 'printf( )':

To use the 'scanf( )' function you need to declare it first by including the "<stdio.h>" file at beginning of your program. This is the same file that also contains the function declaration for 'printf( )'; so, if you are using both 'scanf( )' and 'printf( )' you need to declare the "<stdio.h>" file only once.

The syntax for the 'scanf( )' function is:

scanf("%type specifier", & variable name);

The type specifiers for the 'scanf( )' function are identical to the ones used for the 'printf( )' function; again, they must match the variable they refer to.

Here is a simple program that uses the 'scanf( )' function.

```
#include <stdio.h>
main()
      {
      short i;                                    /* declare variable 'i'                    */
      printf("Enter a number of type 'short':  ");
      scanf("%d", & i);
      printf("Value entered: %d\n", i );
      }
```

Its output will depend on the value entered but it should look something like this:

```
Enter a number of type 'short':  55
Value entered: 55
```

Here are some common pitfalls of the 'scanf( )' function. Notice that an ampersand sign (&) is required before the variable name; this tells the compiler not to look at the actual variable but rather its address, i.e. it converts the variable to a pointer. (Don't worry if you don't understand addresses and pointers; they will be covered later in chapter 9.) Nevertheless, the 'scanf( )' function will not work if you should forget the ampersand! To ensure that your 'scanf( )' function works properly, it is a good idea to "echo" each time the value entered by using a 'printf( )' statement immediately after the 'scanf( )' function.

Prompts are instructions informing the user what to do next. In order to be useful, they should be as specific as possible. Avoid vague statements such as "Enter a number" or "Hit any key." (The second statement has resulted in hundreds of hours of lost productivity because people spend that time searching in vain for the "ANY" key on their keyboard.) Instead use specific statements such as: "Enter the temperature in degrees Celsius: (use an integer)."

Such prompts are extremely important when using a 'scanf( )' statement. If no prompt is given the user will have no clue when the computer reaches a 'scanf( )' function call. Unfortunately, the 'scanf( )' function does not allow for prompts to be included in the function arguments. Unlike 'printf( )', in 'scanf( )' you may not combine prompt strings with the format specifier! For example, the following statement will not work:

```
scanf("Enter a number:  %d", &i);          /* WRONG!!!*/
```

Therefore, if you want to inform the user what data she is expected to enter, then precede your scanf statement with a 'printf( )' statement!

4.4.3. The 'getch( )' Function

While 'scanf( )' is useful for reading numerical values, you should use 'getch( )' for reading a single character from the keyboard. The syntax for 'getch( )' is very simple: when the function is called (with arguments of type 'void') it will return the ASCII value of the first character entered. Here is a simple program:

```
#include <conio.h>                  /* needed for getch                                  */
#include <stdio.h>
main()
      {
      char ch;                      /* variable to store character returned by getch()  */
      printf("Enter a character: ");
      ch = getch();                 /* call getch function                               */
```

```
    printf("\nThe character entered was:  %c", ch);
    }
```

To read an entire string of characters, the 'gets( )' function is used. Since we have not yet learned how to store strings in variables we will postpone the discussion of the 'gets( )' function till chapter 9.

5. C OPERATORS

5.1. Arithmetic Operators

+	self explanatory	
-	"	"
*	"	"
/	"	"
%	returns the remainder (ex. 7%4 returns 3)	

C evaluates mathematical expressions according to precedence of the operator (i.e. multiplications and divisions are done before additions and subtractions) and the expression is executed from left to right. A table of operator precedence is given in Table 2, chapter 14, but whenever there is any doubt use parentheses.

If you are not careful C arithmetic can be tricky. Note that whenever an operation is performed with variables of different types, the variable with the lower precision is automatically converted to the type of the higher precision.

```
main{}
    {
    short x = 1;
    short y = 3;
    float b;
    float a = 3.0;

    b = x/a;
    }
```

Hence, in this case 'b' will be 0.33333 Now consider changing the last line in the program above to:

b = x/y;

Though we still divide 1 by 3, 'b' takes on the value of 0.000000. Explanation: In the first case, a variable of type short (x) was divided by a variable of type float (a); hence, the result was a value of precision 'float' which was then assigned to a variable of type float (b). In the second case, both variables were of type 'short' (x and y) and, hence, the precision of the resulting division is only of type 'short' i.e. 0; assigning the result to a variable type 'float' can not recover the lost precision. Hence, be very careful when performing multiplications and divisions with integer-type variables; use type casting when in doubt! For example, changing the last line to:

b = (float) x/y;

results again in b = 0.33333 because 'x' is temporarily changed to type 'float'.

The same arguments also hold true when an operation is performed on a variable and a constant. For example,

b = x/3;

results in b = 0.00000, while

b = x/3.0;

results in b = 0.333333

Note no power operator exists in C (i.e. if you want to calculate 2^8, 2^8 or 2**8 is meaningless in C, you need to use the pow(2,8) function).

C includes some shorthand notations for arithmetic operations. For example, to increment the variable 'x' by one, you could write:

```
x = x + 1;
```

The same statement can also be written in C as:

```
++x;
```

or:

```
x++;
```

(++x means that x is to be incremented directly before the statement is to be executed and x++ increments x directly after; except for a few rare cases the distinction between the two cases is not important). The statements x = x+1 and x++ are identical; nevertheless, the second one is easier to read.

Increment	Decrement
x = x + 1;	x = x - 1;
++x;	--x;

If you want to increment (decrement, multiply, divide) a variable by a constant value, you can use the following notation:

x = x + 5;	x = x - 5;	x = 5 * x;	x = x / 5;
x += 5;	x -= 5;	x * = 5;	x /= 5;

5.2. Relational Operators

The following relational operators are used to test conditions in branching or loop instructions.

!	Negation (i.e. NOT)
<	Less Than
<=	Less Than or Equal to
>	Greater Than
>=	Greater Than or Equal to
==	Equal (**NOTE: C uses two equal signs to distinguish it from the assignment operator =**)
!=	Not Equal to

5.3. Logical Operators

The C AND and OR operators are:

&&	AND	(example: (a < x) && (x < b))
||	OR	(the '|'-key' is 'SHIFT \-key')

Logical operators are often used in conjunction with relational operators. A common example is to check whether a value falls in a certain range, such as:

```
a < x < b
```

Since relational operators in C can operate on only two values at one time, the above expression has to

be split up into:

(a < x) && (x < b)

If your knowledge of logical operators is sufficient and if you possess a sense for warped jokes, you should have no problem understanding the significance of the following question: (2B) || ! (2B).

5.4. Bitwise Logical Operators

The logical operators in the previous section return only one bit: 1 if it is true, 0 if it is false. For example, the statement:

```
x = a || b;                    /* x, b, c integers */
```

would result in 'x' being 0 if both 'a' and 'b' are 0. If 'a' or 'b' are non-zero, 'x' will be 1. In effect, these operators reduce each variable to a single bit on which they then perform their logical operation. For the vast majority of logical operations, this is all that is needed.

In addition to these logical operators, C provides us with a second type of logical operators: the bitwise logical operators. Instead of reducing variables to single bits on which the operations are performed, bitwise operators perform their operation on each bit individually and then return all these bits. Here is a list of bitwise logical operators:

&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR

Let's look at an example:

```
a = 5;
b = 10;
x = a && b;      /* Logical AND */
y = a & b;                     /* BITWISE Logical AND */
```

The logical AND in the third line results in 'x' having a value of 1 assigned; on the other hand, the bitwise logical AND operation in the fourth line results in 'y' being 0.

Here are the reasons why these two operations yield different answers. In the third line, the logical AND is executed on the variables as a whole: since 'a' is not 0, it is true; the same holds for 'b'; finally, since both 'a' and 'b' are true, 'x' is also true. Therefore, x = 1.

The AND operation in the forth line is bitwise. In binary notation 'a' is 0101, while 'b' is 1010. Each bit will be compared and stored individually in 'y'. Beginning with the least significant bits, 1 AND 0 results in 0. This is stored in the least significant bit of 'y'. The operator then goes on to compare the second least significant bit (see below).

```
a:        0 1 0 1
b:        1 0 1 0
y = a&b   0 0 0 0
```

This process is repeated for each pair of bits and as you can see, it results in 'y' being 0.

Bitwise operations are often used to "mask" certain bits, i.e. for testing a specific bit within a word. For example, if you need to check if the third bit in the variable 'z' is true, you would use the following (incomplete) statement:

```
if( z & 0x4 )
```

The truth table below shows that except for the third bit, the other bits are all masked off. ('X' stands for "don't care.")

z:	X Y X X
4:	0 1 0 0
y = z&4	0 Y 0 0

5.5. Shift Operators

<<	Left shift
>>	Right shift

These operators shift a word by n bits. For example, in:

```
y = x << 2;
```

each bit in the variable 'x' is shifted two bit positions to the left; the two right most bits are set to zero; the two leftmost bits are discarded. For example, 3 << 1 equals 6.

6. BRANCHING INSTRUCTIONS

```
if( xxxx )                  /* xxxx stands for some test using relational */
    {                       /* operators; for example,  a >= b, etc.      */
    statements;             /* the statements will be executed if above   */
    }                       /* test was true.                             */


if( xxxxx )
    {
    statements;             /* executed only if xxxxx is true             */
    }
else
    {
    statements;             /* executed only if xxxxx is false            */
    }



if( xxxx1)
    {
    statements;             /* executed only if xxxx1 is true             */
    }
else if( xxxx2)
    {
    statements;             /* executed only if xxxx2 is true             */
    }
else if( xxxx3 )
    {
    statements;             /* executed only if xxxx3 is true             */
    }                       /* additional 'if else' blocks can be used if desired*/
else
    {
    statements;             /* executed if none of the above is true      */
    }
```

```
variable = ( xxxx ) ? variable1 : variable2;
                                    /* if xxxx is true than variable = variable1
                                    if xxxx is false then variable = variable2 */
```

```
small = (x<y) ? x:y;                /* Example:  Find smaller value*/
```

A word about the placement of opening braces { and closing braces }: Braces in if-statements and in loop-statements are optional if they enclose one, and only one, statement; otherwise they are mandatory. Nevertheless, even when these braces are optional, it is a good idea to use them (especially, if you are new to C programming) and we will follow this convention throughout the rest of this paper. Also note that every statement between braces must always end with a semicolon!

```
/* EXAMPLES using a single statement between braces */

if(x == 2)                                  /* correct way                    */
      {
      printf("x = 2");
      }
else
      {
      printf("x != 2");
      }

if( x == 2)                                 /* also correct                   */
      printf("x = 2");
else
      printf("x != 2");
```

An other branching instruction, which is often used in LabWindows programs, is the switch-construction. At each *case* statement, the switch variable (in the example shown below *x*) is compared with a constant. If the variable and the constants are identical, the statements immediately following the case statement will be executed until a *break* instruction is encountered. If none of constants are identical to the switch variable, then the statements following the *default* instruction will be executed.

```
/* EXAMPLE using a "switch" construction */

switch (x)                                      // Variable x
      {
      case CONSTANT1:                           // if x == CONSTANT1 then statement 1 is executed
            statement1;
            break;
      case CONSTANT2:
      case CONSTANT3:                           // if x == CONSTANT2 or
            statement3;                         // x == CONSTANT3 then statement 3 is executed
            break;
      default:
            default_statement;                  // if none of the statements applied, this
            break;                              // statement will be executed.
      }
```

You should always keep in mind that the switch construction only works by comparing a switch variable with a constant. Unlike an if-statement, you can never use the switch construction to compare two variables!

So far, all our branching instructions were based on conditions. C also has an unconditional branching instruction 'goto label', where label is any name. Keep the use of 'goto' statements to a minimum because they can obscure your code. Here is a program segment using a goto statement:

```
/* Example:  Simply Menu Selection Using a Goto Statement to Repeat Process          */

main()
      {
      unsigned short x;

AGAIN:                              /* Label for goto statement                      */
      printf("Enter 1 to read in data, 2 to display, anything else to exit program");
      scanf("%u", &x)               /* read in menu selection                        */
      if(x == 1)                    /* if 1 was selected call function GetData()     */
            GetData();
      else if(x == 2)               /* if 2 was selected call function DisplayData() */
            DisplayData();
      else                          /* exit program if neither 1 or 2 was selected   */
            exit();
      goto AGAIN;                   /* repeat menu selection; go back to Label AGAIN */
      }
```

7. LOOP INSTRUCTIONS

7.1. FOR Loops

A FOR loop executes the statement(s) between the braces a preset number of times. FOR loops are very useful when it is known ahead of time how often the statements will be executed.

```
for( start; endcond; increment)
    {
    statements;
    }

/*  - where start is the initialization of the count variable, such as x = 0
    - where endcond is a relational condition to see whether the count
    variable has reached the end of the loop, for example x < 1000
    - where increment is the incrementation of the count
    variable, for example ++x or x+=5 etc.          */
```

One very common mistake with FOR loops is forgetting to declare the counting variable.

Here is a working program that will calculate the Poisson distribution. By now, you should be able to understand this program entirely.

```
#include <math.h>
#include <stdio.h>

double Factorial( double x );
double Poisson( double x, double mean);

main()
    {
    double x, mean;

    printf("Enter Mean For Poisson Distribution:  ");
    scanf( "%lf", &mean );
    printf("Enter x:  ");
    scanf( "%lf", &x );

    printf("P(x):  %lf \n", Poisson( x, mean ) );
    }

double Poisson( double x, double mean)                                  /* see page 42
in Reif Stat. Physics                   */
    {                                   /* P(x) = m^x e^-m / x!                */
    return( pow( mean, x ) * exp(-mean) / Factorial(x) );
    }

double Factorial( double n )            /* calculate n factorial (n! = 1*2*3*...(n-1)*n)   */
```

```
{
double fact = 1;
double i;
for( i = 1; i <= n; i++)
      {
      fact *= i;                    /* short for: fact = fact * i                    */
      }
return( fact ):
}
```

7.2. WHILE and DO Loops

WHILE- and DO loops are used when it can not be determined in advance how many times a loop has to be executed to fulfill a certain condition.

```
/* WHILE Loop:                                                          */
while( xxxxx )    /* xxxxx stands for some conditional                      */
    {             /* test, for example:  a > b                          */
    statements;
    }


/* DO Loop:                                                             */
do
    {
    statements;
    }
while ( xxxxx );  /* xxxxx stands again for some conditional test       */
```

A very common usage of the WHILE loop is in conjunction with the kbhit() function. The kbhit() function returns 0 (False) when no key has been pressed and 1 (True) when any key on the keyboard has been pressed. This is very useful if you have a program that executes a repetitious task (i.e. if you monitor something) and you want to terminate the task. The following program segment will execute the statements in the WHILE loop until any key was pressed.

```
while( ! kbhit() )                      /* Note Negation !, i.e. while NOT kbhit()   */
      {
      printf("%d\n", (short) inp(0x310); /* prints values read in from Port HEX 310   */
      }
```

The difference between a WHILE and a DO loop is: in a WHILE loop, the conditional statements are checked before a loop cycle is executed; in a DO loop, the conditional statements are tested after the execution of each loop cycle. Hence, a DO loop executes its statement(s) at least once and as many times as the conditional test remains true. On the other hand, if the conditional test in a WHILE loop is false the very first time, the statement(s) in the WHILE loop will never be executed.

7.3. Nested Loops

Loops can be nested, i.e. one loop can be placed inside of another loop. Consider the following example:

```
/* Example:  Nested Loops */

main()
      {
      short row, column;
      for( row = 1; row < 5; ++row)
            {
            for( column = 1; column < 4; ++column)
                  {
                  printf("row: %hd  column: %hd\n",row, column);
                  }
            }
      printf("That's it folks!");
      }
```

The printout would look something like:

```
row:  1 column:  1
row:  1 column:  2
row:  1 column:  3
row:  2 column:  1
row:  2 column:  2
```

... and so on, until

```
row:  4 column:  2
row:  4 column:  3
That's it folks!
```

As you can see from the printout, the innermost loop is always executed first. When all the innermost loop cycles have been completed, the loop enclosing the innermost loop executes one cycle. Afterwards, the innermost loop cycles are executed all over again. Control is then transferred to the outer loop cycle and it is executed just once. The entire process continues until the condition test in the outermost loop cycle is no longer true.

If this seems confusing to you, try to think of nested loops in terms of n-dimensional spaces composed of discrete blocks. For example, the above nesting consists of two loops; hence, consider a two dimensional space made of discrete elements, such as a tiled floor. When the nested loop is executed then first all the elements in the first row will be selected (i.e. row = 1). Next all the elements in the second row will be chosen and finally the elements in the third and fourth row are selected.

If the above example had three nested loops, instead of the two shown, then you could imagine a cube made of building blocks; while the two innermost loops would select a whole layer of elements, the outermost would select a particular layer.

8. ARRAYS

So far, our programs have been very limited because we could assign only one value to one specific variable. What we would like to do is to assign values to a whole group of variables and manipulate the entire group of data.

For example, if you take 6000 readings in an experiment and you want to calculate their average, it would be ridiculous if you had to type in your C program a statement such as: 'average = (x1 + x2 + x3 + x4 + x5 + + etc)/6000'. Therefore, let's look at the problem from a mathematical viewpoint and express it using a mathematical equation:

$$\bar{x} = \frac{1}{n}\sum_{k=1}^{n} x_k$$

This expression is much simpler to understand than our first expression with the 6000 terms because it uses a summation sign and an indexed variable, x_k. We have already learned how to implement a summation in C, namely by using a loop statement, but we have not yet learned how to express something like an indexed variable in C. Hence, we need to learn about 'arrays' which are the computer-equivalent of indexed variables.

An array is a collection of variables which are all of the same type. For example the declaration:

```
float mydata[ 5000 ];
```

sets aside sufficient memory for 5000 members of an array called 'mydata'. Each member of the array is type 'float'. Each individual member of the array can be accessed by using its corresponding index, also known as offset. For example if you like to print out the value that is stored in the 457th member of array 'mydata', you would type:

```
printf("%f", mydata[ 457 ]);
```

Though we used a numerical value for the offset, we could just as well have used a variable or even an element of another array. If you use a variable for the offset it must always be of type unsigned short! (Think, it would make no sense to refer to a fractional or floating point member, such as the 2.4344th member.)

An array, just like a variable or function must be declared first. It can be made 'global' by declaring it outside of main. Be careful if you decide to declare your array 'local' inside of a function or main. Remember that all local variables are created on the stack when the function is called, and destroyed again when the function is exited. While this is no problem with ordinary variables, arrays can quickly gobble up large amounts of memory (above array 'float mydata[5000]' would use 20000 bytes) and the stack has only about 2000 - 4000 bytes of memory. Hence, if you must use a large array either make it 'global' or precede the local declaration with the **static** statement (i.e. use: static float mydata[5000];). The declaration must always specify the type of the array, the array name and the number of its members.

You can declare the array size with a numerical value such as: short little[33], or with a constant that has been previously defined; you can never use a variable in the array declaration statement.
The following examples are legal:

```
short little[ 3 ];
float mydata[ 6000 ];

#define MAXELEMENT 92                    /* MAXELEMENT is a constant */
long mass[ MAXELEMENT ];
```

C arrays always begin with the zeroth member and the array contains as many members as specified in the declaration. Consider the above declaration for array little; it consists of three members, namely:
1st member: little[0]
2nd member: little[1]
3rd member: little[2]
Note: Though 'short little[3]' has been declared, 'little[3]' is not a member of the array! (Warning, in your program, you could write: little[3] = 5; and it will compile error-free. The C program will store the contents of little[3] in the memory locations directly following little[2] and, without any warnings, it will overwrite any program code or what ever else it finds there. When you run your program, it will either completely 'bomb out' or return garbage, or maybe, run just fine. But don't count on it!)

Now we are ready to attack our original problem, namely the calculation of the average from n data points.

```
#include <conio.h>                  /* for inp() function                                   */
#define MAXNUMBER 6000              /* max. number of data points                           */
float mydata[MAXNUMBER];            /* array is declared global                             */

main()
      {
      short i;
      float average;
      float sum = 0;

/* here are statements that read data in from an experiment and assign them to an array.  */
/* It may look something like this:                                                       */
      for( i = 0; i < MAXNUMBER; ++i)
            {
            mydata[ i ]= inp(0x345);      /* reads data in from port 345h                 */
            }
      for( i = 0; i < MAXNUMBER; ++i)/* now calculate the average                         */
            {
            sum += mydata[ i ];           /* calculate the sum of all values              */
            }
      average = sum / MAXNUMBER;  /* divide the sum by the number of datapts            */

      printf("The average is: %f \n", average);
      }
```

When you declare an array, its elements are not initialized. Until you assign a value to an array element, it contains whatever information that has previously 'lived' at its memory address.

There are some useful short-cuts in assigning values to individual elements. Instead of declaring the array and then assigning a value to each individual member, the values can be enclosed in braces. This short-cut method becomes very powerful when dealing with the most common C arrays, character strings.

```
float small[3];                     /* Clumsy:  Declaration & Assignment separate      */
small[0] = 3.0;
small[1] = 334.9;
small[2] = 13433.8888;

float small[3] = { 3.0, 334,9, 13433.8888};                                        /* More
efficient:                                                                         Declaration &
Assignment in one statement         */
```

```
/* Character String Arrays                                              */
char name[40];                  /* Clumsy:  Declaration & Assignment separate   */
name[0] = 'G';                  /* note:  though 40 elements were declared we   */
name[1] = 'o';                  /* only use the first 6.  This is ok. It is a good idea */
name[2] = 'p';                  /* to 'waste' a few extra bytes in a string to allow */
name[3] = 'h';                  /* for terminating characters.                  */
name[4] = 'e';
name[5] = 'r';

char name[40] = {'G','o','p','h','e','r'};   /* Better:  Declaration and Assignment together */

char name[40] = "Gopher";       /* Best:  Works ONLY with character arrays      */
```

As a final note, I briefly mention multi-dimensional arrays. Whereas in mathematics you would use a variable x_{ij}, in C you would declare this variable as:

```
/* Multi Dimensional Arrays                                             */
/* Program prints every value in an multi-dimensional array             */

#include <math.h>
#define DATAPOINTS 100

float temp[ DATAPOINTS ][ DATAPOINTS ];      /* 2 dimensional array          */

main()
    {
    short x, y;

    for( y = 0; y < DATAPOINTS; ++y)         /* fill array with random numbers */
        {
        for( x = 0; x < DATAPOINTS; ++x)
            {
            temp[x][y] = (rand()%1000)/1000.0;  /* function rand() creates a rand. # */
            }
        }

    for( y = 0; y < DATAPOINTS; ++y)         /* print the random numbers     */
        {
        for( x = 0; x < DATAPOINTS; ++x)
            {
            printf("Row:  %d\t Column:  %d\t Value: %f\n", y, x, temp[x][y]);
            }
        }
    }
```

The output from this program would look like: (Values may be different)

```
Row: 0  Column: 0   Value: 0.345
Row: 0  Column: 1   Value: 0.434
Row: 0  Column: 2   Value: 0.923
```

etc.... until:

```
Row 99  Column: 98  Value: 0.743
Row 99  Column: 99  Value: 0.334
```

9. POINTERS

9.1. Pointers and Variables

So far we have used variables and arrays to store or to retrieve data. For example, when we use the statement:

short a = 432;

the C compiler will set two bytes of memory aside and store the value 432 there. We don't know where the content of variable 'a' is stored. Frankly, most of the time we don't care to know because the C compiler very efficiently takes care of setting aside memory, and retrieves and stores variables for us. Nevertheless, C gives you the ability to retrieve the address of a variable and you can assign a variable to particular address. Furthermore, you can store a memory address in a variable. If this is the case, then the variable does not store the content of a memory address (as conventional variables do); the variable contains a memory address. Because this variable 'points' to a memory address it is referred to as a 'pointer'.

To use a pointer it must be declared first. To declare a pointer you must specify its type, or rather the type of variable that it will point to, its name, and you always must precede the pointer's name with an asterisk: (in this case * has nothing to do with multiplication)

```
/* valid pointer declarations */
short *somepointer;
char *someotherpointer;
```

A pointer is only useful if it points somewhere. Above declarations merely create pointers but the pointers don't know yet where to point to. Hence, the pointers are assigned to the address of a variable to which they will point. This is done by preceding a variable name with an ampersand (&), also referred to in C as an address operator:

```
/* assigning a pointer the address of a variable                                    */
short a = 300;
char X = 'y';

somepointer = &a;                              /* Note: this pointer was declared in the above example
            */
someotherpointer = &X;      /* This pointer was also declared in above example      */
```

Now the pointers 'somepointer' and 'someotherpointer' contain the address at which variables 'a' and 'X' are stored.

Next want to use the value that the pointer is pointing to. In other words, we want to print out the value of variables 'a' and 'X' without using them. To find the value that is stored at the location that a pointer points to, we use the indirection operator, the asterisk: * . Hence, from the above example, the expression:

*somepointer

is identical to: a

```
/* POINTER EXAMPLES */
short x;
short *pointer_to_x;
```

```
x = 3;
pointer_to_x = &x;          /* store address of variable 'x' in  pointer 'pointer_to_x'     */

*pointer_to_x = 5;                              /* store value '5' at location where pointer is pointing at
          */
printf( "x: %d\n", x);      /* print value variable 'x' contains                          */
```

The output from the above program segment would be:

```
x:  5
```

9.2. Pointers and Arrays

By now you should be wondering. In the above example, two additional lines of code and two additional bytes of memory were required to assign simply a value to a variable. So what is so great about pointers?

Pointers allow you to write more efficient and faster running code when you work with functions and arrays. To understand why this is so, we need to understand how a compiler works.

For example, some array: 'short mass[4000]' is declared and you refer to element 876, i.e. 'mass[876] = 1.' The compiler first finds the beginning address of the array, which is identical to the address of mass[0]. To find the address of the 876th element, 2*876 is added to the address of beginning address of the array. (2*876 is added because element 876 is 876 elements past the zeroth one, and since each element takes 2 bytes, the address of element 876 is 2*876 bytes past the address of element 0.) Since arrays are most often used in loops, each time an array element is referred to, the compiler has to go through the above calculations. Hence, if we initialize our array to 0 in a FOR loop, above calculations have to be carried out by the compiler 4000 times. (See the program below.)

```
/* Example:  Initialize an array without using pointers                              */
main()
     {
     short i;
     short mass[ 4000 ];

     for( i = 0; i < 4000; ++i)  /* initialize the array                            */
          {
          mass[ i ] = 1;         /* set every array element to 1                 */
          }
     }
```

Here is the same example but this time we use pointers. If we want to refer to an element in the array we first find, similar to the compiler, the starting address. Hence we find the address of mass[0]. We could use:

```
short mass[ 4000 ];                         /* declare array                          */
short *startarray;                          /* declare a pointer                      */
startarray = &mass[ 0 ];    /* assign the pointer to the beginning of the array    */
```

This would be perfectly fine but unnecessary. Because the beginning address of an array is referred to very often, a special pointer was invented. Without any further explanation, this special pointer for the above array would simply be: mass. Hence, the statements '&mass[0]' and 'mass' are identical. Above example can be rewritten:

```
short mass[ 4000 ];                    /* declare array                                    */
short *startarray;                     /* declare pointer                                  */
startarray = mass;                     /* assign the pointer to the beginning of the array */
```

Note that the pointer 'mass' has already been declared implicitly in the array declaration statement.

So far we have only found the beginning of the array. A pointer to the 876th element can be found by adding 876 to the beginning of the array pointer. This example, will print out the value of the 876th element.

```
short mass[ 4000 ];                    /* declare array mass                               */
short *member_876th;          /* declare pointer                                           */
member_876 = mass + 876;      /* calculate 'address' of 876th element                      */
printf("%u", *member_876);    /* print value of 876th element of array mass[]              */
```

Remember, in the first example it was said that to reach the 876 element, 2*876 bytes were added to the beginning address, taking into consideration that each element of the array uses two bytes. In the above example, only 876 was added to the starting address because C automatically converted it (i.e. multiplied it by 2) to account for the element size. In other words, if you increment a 'short' pointer by one, it actually increments by two bytes.

If you are only accessing the 876th member of array 'mass', then the pointer approach certainly would have been a mess and a waste of time. On the other hand, if you initialize or access a large array the advantage and the power of pointers should become obvious.

```
/* Example:  Initialize an array  using pointers  */
main()
      {
      short i;
      short mass[ 4000 ];
      short *temp_ptr;          /* temporary pointer                                      */

      temp_ptr = mass;          /* temp_ptr points now to beginning of array              */

      for( i = 0; i < 4000; ++i)  /* initialize the array i.e. set every element to 1     */
            {
            *temp_ptr = 1;      /* store 1 wherever the pointer points to                 */
            ++temp_ptr;         /* increment pointer to point to the next element; FAST   */
            }
      }
```

9.3. Pointers, Arrays and Functions

When arguments are passed to a function, the C compiler will make a copy of the arguments and push them onto the stack. The function will pop the arguments from the stack and use them. While this works fine for simple variables it would be prohibitively time and memory consuming for entire arrays. Therefore, whenever a local array needs to be passed to a function, only a pointer to the array is passed.

As the last example, we write a function that will set all elements of an array to some value. Since we want to keep the array local, we will only pass an array pointer to the function. Furthermore, we will pass the value that we want the array to be initialized to, as an argument. The function itself will operate directly on the array; hence, no value will be returned by the function and we can declare it type 'void'. For example:

```
void Initialize(short *array_beg, short init_value);
```

The only problem with the above function declaration is that the function would not know how many elements our array contains. Hence, we need to provide the function with this information and we pass the information as an argument. Here is our final function which you could use in any program.

```
void Init(short *array_beg, short elements, short init_value);

main()
    {
    static short lots_of_datapoints[ 30000 ];    /* declare the local array static          */

    /* set all array elements to 273 */
    Init( lots_of_datapoints, 30000, 273 );      /* call function Init()                    */
    }

void Init(short *array_beg, short elements, short init_value)
    {                                            /* Function works only for arrays with     */
    short i;                                     /* less than 32767 elements.               */
    for( i = 0; i < elements; ++i)               /* initialize entire array                 */
        {
        *array_beg = init_value;                 /* assign init_value to each element       */
        ++array_beg;                             /* increment pointer                       */
        }
    }
```

9.3.1. String Functions: 'gets( )' and 'strlen( )'

Character strings are always stored in arrays. Therefore, functions that read, print and manipulate strings request a pointer to such a character string. In chapter 4 you learned how to manipulate numerical values. In this section you will become familiar with character strings.

To read a character string from the keyboard the 'gets( )' function is used. This function stores them in an array till it encounters a carriage return entry. The address of the array is passed to the function as argument in the form of a pointer. This sounds far more complicated than it actually is. Here is a simple example:

```
#include <conio.h>                               /* required for gets( ) */
#include <stdio.h>
char st[80];                                     /* character array declaration */
main()
    {
    gets( st );                                  /* read string:  Note 'st' is a pointer! */
    printf("%s\n", st );                         /* print string */
    }
```

Since the character strings is an array, one may manipulate it like any other array. Consider the following program:

```
#include <conio.h>                          /* needed for gets */
#include <ctype.h>                          /* needed for toupper */
#include <stdio.h>
#include <string.h>                         /* needed for strlen */
char st[80];
main()
      {
      short i;
      gets( st );

      for( i = 0; i < strlen( st ); i++)    /* convert string to upper case */
            {
            printf("%c", toupper(st[i]) );
            }
      printf("\n");

      for( i = strlen( st ); i--; i > 0)    /* print string backwards */
            {
            printf("%c", st[i] );
            }
      }
```

The output of the program depends on the input, but it may look something like:

```
This is a TEST
THIS IS A TEST
TSET a si sihT
```

There are two new string functions in this program: 'strlen( )' and 'toupper( )': 'strlen( )' returns the number of characters a string contains and 'toupper( )' converts a lower case character to upper case. See if you can understand how the program manages to print the string backwards.

9.3.2. Time Functions: 'clock( )' and 'time( )'

There are two different time functions: 'clock( )' and 'time( )'. The difference between them is the resolution. The 'clock( )' function returns how many milliseconds have passed since the 'clock( )' function has been called the first time within the program. The 'time( )' function returns the number of seconds elapsed since 1/1/70 GMT. Both function return their result in a variable of type 'long'.

Here is a program that uses the 'clock( )' function:

```
#include <stdio.h>                  /* required for 'clock( )'                  */
#include <time.h>
main()
      {
      long t0, t1, t2;              /*variables to store the clock() return values  */
      t0 = clock();                 /* first 'clock( )' function call           */
      printf("t0:  %f sec.\n", t0/1000.0);
      getch();                      /* delay                                    */
```

```
    t1 = clock();                       /* second 'clock( ) function call          */
    printf("t1:  %f sec.\n", t1/1000.0);
    getch();                            /* delay                                   */
    t2 = clock();                       /* third 'clock( ) function call           */
    printf("t2:  %f sec.\n", t2/1000.0);
    }
```

The output from this program is:

```
t0:  0.000000 sec.
t1:  2.960000 sec.
t2:  4.880000 sec.
```

As expected, the first function call returns 0. Consecutive calls indicate the time elapsed between the first call and any call thereafter.

There are two potential problems with the 'clock( )' function. First, though the function returns the time elapsed in milliseconds, its accuracy is nowhere near that. The internal clock is only updated every 55 milliseconds. Therefore, reading the clock more than once within 55 milliseconds is meaningless. Second, though the PC returns its time from the 'clock( )' function in milliseconds, this convention is not universal. Therefore, if you want to make your program 'portable' you should always divide the value returned by 'CLOCKS_PER_SEC'; this is a constant defined in <time.h> and it ensures that the result corresponds to the number of seconds elapsed and it is independent of the type of computer used. (A common error in dividing the returned value by CLOCKS_PER_SEC is ignoring the fact that you are performing an integer division and may loose some accuracy; if this is the case, type cast either CLOCKS_PER_SEC or the returned value to a type 'float'. Note, this is why the program above divides by 1000.0 (instead of by 1000)).

Though the 'time( )' function is not as "accurate" as the clock function, it returns the number of seconds elapsed since January 1, 1970 GMT. This information then can be used to extract the date. Luckily, you do not have to calculate it; various (string) functions for such conversions into various date formats exist. Here is such a program:

```
#include <time.h>
#include <stdio.h>
#include <string.h>
void main()
    {
    long ltime;
    time( &ltime );                         /* Get UNIX-style time and display as number and string.
*/
    printf( "Time in seconds since GMT 1/1/70:\t%ld\n", ltime );
    printf( "UNIX time and date:\t\t\t%s", ctime( &ltime ) ); /* convert string          */
    }
```

The output from the program is:

```
Time in seconds since GMT 1/1/70: 753561598
UNIX time and date:           Wed Nov 17 10:39:58 1993
```

Note how the 'ctime( )' function was used to convert the result from seconds elapsed into a more useful form.

The biggest problem with the 'time( )' function is that it assumes that the internal clock in your computer has been set correctly. Unfortunately, PCs are notorious for amnesia with regard to time and date. Also, the internal clock often runs at it own peculiar pace. Therefore, if you are going to rely on the

time( ) function, always check first that the internal clock works and that the DOS command 'date' and 'time' displays the correct value.

10. STRUCTURES

Structures are similar to arrays in that they contain a collection of data. The differences between structures and arrays are:

Arrays	Structures
Consists of elements	Consists of members
Elements are all of the same type	Members can be of different type
Elements are referred to by index	Members are referred to by name

Before you can declare a structure you must first create a structure template. The syntax for the template is:

```
struct structurename
    {
    members;
    }
```

As an example, consider a structure that contains various information about a chemical element.

```
/* Structure Template:  anyelement                                          */
struct anyelement
    {                       /* Member List:  Note members can be of different type  */
    char *name;             /* Name of element                                      */
    short number;           /* Atomic number of element                             */
    float mass;             /* Mass of element                                      */
    float density:          /* Density of element                                   */
    float melttemp:         /* Melting Temperature of element                       */
    };
```

Now we can declare the actual structures:

```
struct anyelement Al;       /* declare structure with name 'Al' to follow template 'anyelement'*/
struct anyelement Pb, H;  /* declare additional structures 'Pb' and 'H'                 */
```

Note that there is a big difference between creating a structure template and declaring a structure. Creating a template does not set aside any memory for a structure; it merely informs the compiler that a structure will be declared later, the number of members that it will contain and their type. Declaring a structure will set aside the necessary amount of memory for each structure as outlined in the structure template. In the above example, 'anyelement' is the template, while 'Al', 'Pb' and 'H' are the actual structures.

Once a structure has been declared you can assign values to its members. A member is referenced by the structure's and the member's name separated by a period. Here is an example:

```
/* Assign values to individual structure members      */
Al.name = "Aluminum";
Al.number = 13;
All.mass = 26.982;
Al.density = 2.7;
```

```
Al.melttemp = 933;

/* short cut: declaration and assignment combined    */
struct anyelement C = {"Carbon", 6 ,12.01, 2.26, 4300};
```

Each member of a structure can be treated just like an ordinary variable and the content of an entire structure can be directly assigned to another structure.

```
printf("Al melting point is: %f", Al.melttemp); /* Al.melttemp is similar to an ordinary variable */

Pb = Al                                         /* all members of struct. Pb are assigned the values
                                                   prev. assigned to the members of struct. Al    */
```

Instead of declaring a single structure, such as Al, Pb etc., arrays of structures can be declared. Consider the following example:

```
struct anyelement PTable[5];
PTable[0].name = "Hydrogen";
PTable[1].name = "Helium";
PTable[2].name = "Beryllium";                   /* etc... */
PTable[0].mass = 1.0079;
PTable[1].mass = 4.0026;                        /* and so on..*/
```

11. DATA FILES

A large number of programs in experimental sciences deal with data manipulation. Very often, a designated computer (or an entire system) is set up to monitor a process. To increase the speed of the data acquisition, the raw data collected by the computer is written to a data file on a disk. After the experiment has been completed, the raw data will be read back into the computer and analyzed; the results may be stored in a new file and written to disk again. Depending on the size of the project, the last step may be repeated many times.

11.1. ASCII and Binary Data Representation

When working with numerical data it is important to understand the difference between the two different data formats: ASCII and Binary. Why they exist can be seen by the following example. Assume that the integer value 12345 needs to be stored either in the computer memory or in a data file.

The first method, the ASCII representation, breaks the number into individual digits and then assigns one byte of information, i.e. the ASCII value, to each number. It then stores each of these ASCII values in one byte size records. (A table of ASCII values can be found in the QC help screen.) In other words, the ASCII value for '1' is 49, for '2' is 50 etc. Therefore, the number '12345' will be stored in an ASCII file with the following five bytes: 49 50 51 52 53.

The second method converts the integer value 12345 to a binary representation i.e. HEX 0x3039. If it is saved as a variable of type 'short', it then will be stored in a two byte block.

Each of two methods have their advantages and disadvantages. If speed and compactness is a consideration, the binary encoding clearly is superior over the ASCII method. As our example shows, the binary method uses two thirds less bytes to store the same information. Also, binary records are always of the same length i.e. one, two or four bytes depending on data type; this allows for faster search and data access. Since the computer stores numbers in binary format, no conversion is needed when reading or writing numbers to or from a file.

The price paid for the compactness of the binary storage method is that the user must have some knowledge about how the binary data has been stored. For example, if you have a binary data file and you do not know if the data has been stored as shorts, floats or characters, then there is almost no way of determining the data type except through trial and error and a lot of guesses. On the other hand, the ASCII method makes no such assumptions; everything is converted the same way, though somewhat inefficiently. This makes ASCII files certainly more "portable," i.e., they can be read by different applications and computer systems without any great difficulties. Clearly, there is no great advantage in storing a text file as a binary file and, therefore, almost all text files are stored in ASCII format. Also, remember that an ASCII file can be read through the DOS command 'TYPE' and it can be created and altered with any text editor program such as QC or EDIT. Below is a table summarizing some of the characteristics:

DATA REPRESENTATION	ASCII	BINARY
"Typical" Application:	Text	Numbers
Storage Efficiency:	Worse	Better
Access Time to Individual Records:	Slower	Faster
Portability:	Better	Worse
Reading or Editing File Content:	Easy: use "TYPE" or any Text Editor	Difficult, use specific program.

When you write data to a file then the choice is yours which format to use. As a rule of thumb, whenever speed and size are of no great importance, it's preferred to use the ASCII format. Also, if you want to read your data into a spreadsheet, such as Excel, then you should use the ASCII format.

When you read a data file then you do not have a choice on which data format to use. You must always read it using exactly the same data format it was written. If it's a binary file then you must (in addition to reading it as a binary file) also use the same variable types that were used to create the file!

11.2. Files: General

Working with data or text files involves three steps:

1) A file is opened for reading and/or writing
2) Data is read from the file and/or written to it
3) The file is closed (i.e. buffers needed for the file input/output are cleared)

Step one and three are almost identical for ASCII and binary files. Step two is different for each type, that's why it's covered last.

11.3. Opening and Closing a Data File

The C function 'fopen()' opens a file. Prior to using the function, a pointer of type 'FILE' must be declared in the program. This pointer is often referred to as 'file pointer'. Since it is possible to have multiple files open at any given time, a distinct file pointer must be declared for each file.

The syntax of the 'fopen' function is:

```
FILE *some_filepointer;
some_filepointer = fopen( filename, activity and filetype);
```

The filename can be any legal DOS filename with extension and path, enclosed in quotation marks. 'Activity' can be any of these choices (the quotation marks are required!)

"wt"	Open an ASCII file for writing. Creates the file if it does not exist.
"rt"	Open an ASCII file for reading. The file must already exist.
"at"	Open an ASCII file for appending (write only at the end of file.) Creates the file if it does not exist.
"wb"	Open a binary file for writing. Creates the file if it does not exist.
"rb"	Open a binary file for reading. The file must already exist.
"ab"	Open a binary file for appending (write only at the end of file.) Creates the file if it does not exist.

If the file type (i.e. "t" or "b") is omitted, C defaults to ASCII files. A file created as an ASCII file must always be accessed as an ASCII file; similarly, a binary file can only be read reliably as a binary file.

Function 'fclose()' closes a data file; it clears any buffers that were created during 'fopen()'. The syntax for the 'fclose()' function is:

```
fclose( some_filepointer );
```

Note that both fopen() and fclose() are C functions and that they return arguments. Though you can ignore the arguments, it is usually a good idea, especially when dealing with files, to check these arguments. For example, if 'fopen()' succeeds, it returns a file pointer; if it fails, it returns a NULL pointer. Hence, it is usually a good idea to use an if-statement to check for a NULL pointer argument.

11.4. Writing to an ASCII Data File

Of the many C functions available for writing and reading ASCII data files, we will only cover 'fprintf()' and 'fscanf()' because a) we should be familiar with their non-file analog and b) they are the most versatile functions for handling numbers, particularly floating point type.

The syntax for the 'fprintf()' is:

fprintf(some_filepointer, control, arguments);

'Control' and 'arguments' are identical to the 'printf()' function: 'control' consists of escape sequences enclosed in quotation marks, and 'arguments' consists of the variables.

'fprintf()' is a very versatile function: replacing the pointer 'some_filepointer' with **stdout** (no quotation marks) turns 'fprintf()' into our old 'printf()' function. (This feature can be very helpful when debugging a program). Also, changing 'some_filepointer' to **stdprn** redirects all the output to a printer.

Here is a complete program that creates a data file called "test.dat" and then writes 50 real numbers to the file:

```
/* minimum File Write Program                                          */
#define MAX 50                  /* number of data points to be written to file  */
#include <stdio.h>              /* needed for function fopen() fprintf()        */

main()
    {
    FILE *fptr;                 /* declare our file pointer of type FILE        */
    short i;

    fptr = fopen( "test.dat", "wt");  /* open  and create text file "test.dat" for writing */
    for( i = 0; i < MAX; i++)
        {
        fprintf(fptr, "%d",i/MAX ); /* write data (i.e. the value of a/MAX) to file */
        }
    fclose(fptr);               /* close file                                   */
    }
```

11.5. Reading an ASCII Data File

The 'fscanf()' function reads data from file. Its syntax is:

fscanf(some_filepointer, control, addresses);

Again, control and addresses are identical to the 'scanf()' function.

When data is read from a file, often it can not be determined ahead of time how many data points a file contains. Hence, data is read from the file until the end of the file (eof) has been found. Function 'feof()' is used for this purpose; it returns argument True when the end of file has been found, False otherwise.

```
/* minimum File READ Program                    */
#include <stdio.h>                              /* required for functions fscanf() and feof()  */

main()
    {
    FILE *fptr;                     /* declare file pointer                                    */
    float in;                       /* variable                                                */

    fptr = fopen( "test.dat", "rt");    /* open file "test.dat" for reading only               */

    while( ! feof( fptr ) )         /* while end of file has NOT been reached, repeat          */
        {
        fscanf(fptr, "%f", &in);        /* read in a data point                                */
        fprintf(stdout, "%f\n", in);    /* print out data point (to screen)                    */
        }

    fclose(fptr);                   /* close file                                              */
    }
```

11.5. Writing Excel Files

Data acquired in a C program can easily exported to spreadsheets such as Excel if you follow the following rules when creating your data file.

- Store your data in an ASCII file.
- Separate the output to each column by a tab: “\t”
- At the end of each row, add a new-line character: “\n”

Study the sample program below.

```
#define MAX 20
#include <stdio.h>
#include <math.h>

main()
        {
        FILE *fptr;
        short i;
        float cval, sval;
        fptr = fopen("test.dat", "wt");
        for( i = 0; i < MAX; i++)
                {
                cval = cos( i /10.0 );
                sval = sin( i/ 10.0 );
                fprintf( fptr, "%d\t%f\t%f\n", i, cval, sval );
                }
        fclose( fptr );
        }
```

The output file “test.dat” can now be read into Excel. It will look like this:

0	1	0
1	0.995004	0.099833
2	0.980067	0.198669
3	0.955337	0.29552
4	0.921061	0.389418
5	0.877583	0.479426
6	0.825336	0.564642
7	0.764842	0.644218
8	0.696707	0.717356
9	0.62161	0.783327
10	0.540302	0.841471
11	0.453596	0.891207
12	0.362358	0.932039
13	0.267499	0.963558
14	0.169967	0.98545
15	0.070737	0.997495
16	-0.0292	0.999574
17	-0.128844	0.991665
18	-0.227202	0.973848
19	-0.32329	0.9463

11.6. Writing and Reading a Binary Data File

To write data in a binary format to a data file you need to use the 'fwrite()' function; to read the data back use the 'fread()' function. The syntax for these functions is:

fread(void * buffer, short size, short count, FILE *stream);
fwrite(void * buffer, short size, short count, FILE *stream);

where 'buffer' is a pointer to an array to store or read the data from; 'size' corresponds to the number of bytes each data element takes up; 'count' is the maximum number of items to be read or written; 'stream' is a file pointer. Both fread( ) or fwrite( ) can be called numerous times before the fclose( ) statement. In the case of fwrite( ), the new data will be appended to the data file while in the case of fread( ) data reading will continue, starting at the point where it left off in the previous fread( ) call; care must be taken not to read past the end-of-file.

Here a sample program that creates such a data file and then sends it to the DAC.

```
#include <conio.h>
#include <stdio.h>
#include <phys5122.h>

#define MAX 4096
short data[MAX];
char filename[50] = "c:\\test1.dat";          /* Note the double '\\' to indicate the directory.      */

main( )
      {
      short i;
      FILE *fptr;                             /* declare a file pointer                              */
      float f = 10000;

      for( i = 0; i < MAX; i++)               /* fill array with data                                */
            data[i] = i;
```

```
fptr = fopen( filename, "wb" );                 /* open file for writing as a binary file                 */
for( i = 0; i < 100; i++)                       /* write data to file repeatedly!                         */
        fwrite( data, sizeof( short), MAX, fptr); /* Note:  the same array is written repeatedly to */
                                                /* the data file; it is appended each time to the file.   */
fclose( fptr );                                 /* close the file                                         */

while( !kbhit( ) )                              /* send the data to the DAC until a key has               */
        D2A_DMA_CH0_File( filename, 409600, f );  /* been hit.                                            */
getch();
}
```

12. GRAPHICS

A word of caution: Computer graphics are very device dependent. This means that the functions listed below will work only on (some) IBM computers. Clearly, the graphics functions are not part of standard ANSI C.

To display any picture on your computer screen, little dots on your computer screen, known as 'pixels' are turned on/off. The appearance of an image depends on two factors: resolution and grayscale capability. The resolution is dependent on how many pixels a screen or a computer can display. Clearly, the more pixels that are available, the more crisp the image. The grayscale capability is defined by the number of gray shades a given pixel can take; most newspaper pictures consist of only two gray values, black and white.

The resolution and the grayscale capabilities depend on the type of video adapter installed in the computer and the type of monitor attached to the computer. At the current time, there exist about 6 major types of video adapters available for the IBM. Each of these video adapters is capable of producing video modes with varying resolutions and grayscale capabilities. (Table 4 lists the available video modes and their resolution.)

13. ADDITIONAL INFORMATIONABOUT C PROGRAMMING

Additional information can be found in the following books:

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Second Edition. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

This book is a classic because it is written by the people who actually invented C and it covers all of C. It is useful as a reference and can be used for more advanced programing techniques. (It is not very helpful as an introduction.) Be sure that you buy the second edition because the first edition is not ANSI C.

Waite, Prata, Costales, Henderson. *Microsoft QuickC Programming*. Second Edition. Redmond W.A.: Microsoft Press, 1990.

This book covers the Microsoft QuickC editor and compiler and the C language. It can be used as an introduction to programming.

PART 2:

LABWINDOWS

1. INTRODUCTION

The C-compiler that you will be using for this course is part of National Instrument's LabWindows/CVI (v4.01) (LW) package. In addition to an ANSI C compiler, it contains libraries written for data analysis, instrumentation interfacing and *graphical user interfaces* (GUI). The reason for relying on these additional libraries is that ANSI C is very limited, particularly when programming in a Windows based environment because ANSI C does not include any graphics commands.

Since most of today's software uses GUI's, we think you should become familiar programming in such an environment. On the one hand, such programs are more attractive and user-friendly. On the other hand, the simplicity and ease for the user is bought at the expense of the programmer, and what appears very trivial to a user can often be very complicated to implement from the programmer's viewpoint. In other words, the complexity shifts from the user to the programmer. Therefore, you as the programmer will have to learn, in addition to ANSI C programming, a rudimentary understanding about the Windows environment.

Understanding fully how the Windows environment works is far too complex and beyond the scope of this course. Luckily for you, the LW software package will remove some of the difficult tasks and even write some of the program code for you. Nevertheless, you still need to be able to read C-code and know how to modify it.

2. DOS / SINGLE TASK OPERATING SYSTEM

In a *character mode based operating system*, such a DOS, the operating system either waits for input or executes a single task. In a typical sequence you input the name of a program, DOS loads the program and executes it; when the program needs some user input, it outputs information and then waits until the user responds and then continues.

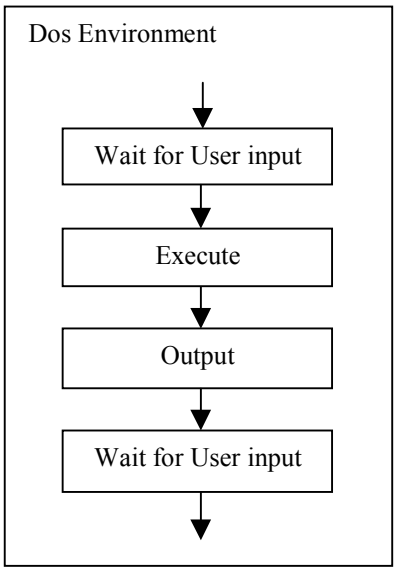


Figure 2-1: DOS Environment Flowchart

From a programming view, there are some advantages to this type of environment: it's very easy to visualize the program flow because it flows from top to bottom. Also, input to the program is received only at specific times during the program execution, namely when the program waits for it because it has requested it in its code. Furthermore, the input comes from a predetermined device, most often the keyboard. Finally, since at any given instance only one program can execute, no provisions need to be made to check if some other program has changed the environment. As you can imagine, this type of environment is fairly efficient in terms of video and memory resources required while executing a program. The standard ANSI C is ideal for such situations and contains all the functions to work well in such an environment.

From a user's viewpoint, there are serious disadvantages to this environment. Because it is a single task environment, in order to run an application all other applications must have been terminated. For example, you cannot work on a document while you print a draft copy of it. Whatever task is running owns the machine. Since the user input is character based, you have to type a lot, which implies that the user has to remember a lot of commands. Overall, the hardware is not used very efficiently since most of the time the computer is waiting for user input.

As we can see today, clearly the user's based viewpoint has won and not many users want to go back to a character-based operating system. (Programmers may be a different story.)

3. WINDOWS OPERATING SYSTEM

Windows is a multitasking operating system that uses a graphical user interface (GUI). Multiple tasks can execute at one time and all user input is received through windows. While such an environment is user-friendlier, it is more complex than the simple character mode based operating system and a discussion of the complete operating system is clearly beyond the scope of this manual. What follows is a brief overview with excerpts taken from the Microsoft Visual C++ 4 compiler help manual. Do not worry if you do not understand everything, instead try to get familiar with the key concepts presented.

3.1. Process and Threads

In the Windows operating system, when you run an application, you begin a *process*, which consists of creating at least one *thread*, loading the program code and loading the required dynamic link libraries (DLL's). A thread represents one of possibly many tasks needed to accomplish the job.

A process can have a single thread or it can be "*multithreaded*." A multithreaded process is useful when a task requires considerable time to process. This task can run within one thread, while another task runs within a separate thread. The threads can be scheduled for execution independently on the processor, which allows both operations to appear to occur at the same time. The benefit to the user is that work can continue while the first thread completes its task. Another benefit is that on a multiprocessor system running Windows NT two or more threads can run concurrently, one on each processor.

3.2. Multitasking and Scheduling

Windows provides an environment in which, from the user's point of view, multiple programs appear to execute simultaneously. This is called "*multitasking*." A processor is capable of processing one thread at a time. Multitasking under Windows involves dividing the CPU time among the threads that belong to the various processes. This allows processing to switch back and forth between the threads.

One type of multitasking, preemptive multitasking, is a feature of a Win32 operating system that prevents one application from monopolizing the processor. Instead of waiting for a thread to voluntarily relinquish control of the processor, the operating system interrupts processing of the thread after a particular amount of time or after receiving a request from a thread that has a higher priority. This process gives all the threads necessary access to the processor.

The operating system achieves multitasking by:

- Running a thread until the thread's execution is interrupted or until the thread must wait for a resource to become available.
- Saving the thread's context.
- Loading another thread's context.
- Repeating this sequence as long as there are threads waiting to execute.

A key issue in multitasking concerns *scheduling*, that is, determining which thread should run next. At the time of creation, each thread either assumes or is explicitly given a priority of execution. Threads have various states, such as being inactive for a period of time, waiting for some event to occur, or being ready to run. In general, the highest priority thread that is ready to run is given the processor. It is placed in the running state until it is preempted, or it enters a waiting state until a particular event occurs.

3.3. Messages and Windows

Most Windows users are familiar with the term “window” and the visual elements that characterize applications. From your standpoint as a Windows programmer, windows take on a new meaning. In a Windows-based application, windows are the primary method of communicating information from the application to the user. Similarly, the user uses the window to communicate with the application, thus achieving the desired behavior to accomplish a task.

The Windows environment is a message-based system. Whenever an *event* of interest occurs within the system, such as a mouse movement or a key press, a message is generated by Windows or by an application. How an application responds to these messages determines its behavior. Messages contain information about the event, such as its type, the time it occurred, and the specific window to which it was directed.

When a Windows-based application is started, the program begins at an entry point in the code. This entry point is a function that is called WinMain in Windows, while in LabWindows it is called RunUserInterface. The application then creates one or more windows. Each window contains a window procedure that is responsible for determining what the window displays and how the window responds to user input. A piece of code called a message loop retrieves messages from the message queue and gives them back to Windows to send to the appropriate window procedure or *callback function*. This gives the application a chance to preprocess messages before they are sent to a window.

3.4. Event Driven Programming

The Windows architecture radically differs from the program-driven architectures that most character-mode applications use. By comparison, Windows-based applications act "passively” - that is, they perform significant work only in response to messages generated by the end user, the operating system, or other applications.

Event-driven programming involves writing code that is able to interpret these messages and respond in the appropriate manner. Typically, tasks that are assigned to message handling code such as callback functions involve reacting to the user. Making the conceptual shift to the event-driven paradigm is one of the major barriers for programmers as they make the transition to Windows-based programming.

4. LABWINDOWS CONCEPTS

The following is a detailed description of the LW environment, which operates in the Windows environment. You should try understanding the concepts presented because you will use them for the rest of this course. When you get to the examples presented in this manual, it is very important that you follow them using one of the lab machines running LW.

4.1. Graphical User Interface (GUI)

The GUI is a vital part of your application. It is relatively easy to create sophisticated GUI's through the *user interface editor (uir-editor)* which is part of the LW package.

Panels: Parent Panels and Child Panels

The most fundamental element of the GUI is the window, or as LW calls it the *panel*. (In this context, the names "window" and "panel" are synonyms.) Every application that you create will run inside a panel.

The top most or outermost panel is called the *parent panel*. It can contain *child panels*, which in turn can contain child panels. For example, the desktop is the parent panel of all child panels.

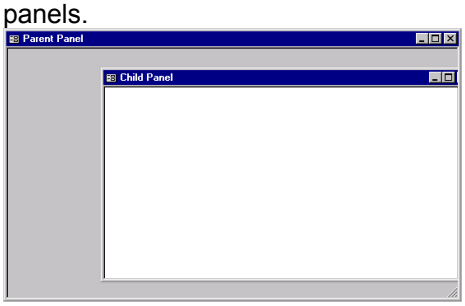


Figure 4-1: Parent and child panel

Child panels are tied to the parent panels in the following ways:

- A child panel is always located on top of its parent panel.
- A child panel is confined (or clipped) to the client area of the parent panel. If moved outside the parent's drawing area, any portion of the child panel outside that area is not drawn.
- The main actions performed on the parent (hide, move, destroy, maximize, and minimize) automatically affect a child panel.
- The Windows operating system and, therefore, LW also manages child panels indirectly by managing their parents.

Do not think of panels only as rectangular objects as shown in Figure 4-1. Panels, especially child panels, come in all types of shapes and colors as shown in Figure 4-2.

Panel Names

Each panel must have a constant name associated with it. Depending on the circumstances, LW also refers to these names as "panel *labels*" or "panel *handles*" or “control IDs”.

Since the labels are essential for the operation of the GUI, LW will automatically suggest a default name, which for most occasions is suitable and does not need changing. For example, the suggested label for the parent panel is PANEL. A binary switch (child) panel placed onto the parent panel will be labeled as “PANEL_BINARYSWITCH”. (Note: as a convention, constants in C are always

capitalized.) A GUI with all its child panels and the associated labels is shown in Figure 4-2.

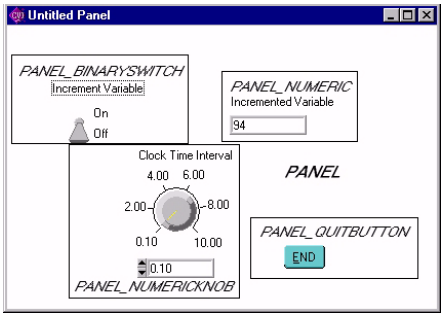


Figure 4-2: GUI with panels and panel labels

Controls and Displays

All (child) panels fall into two categories: they are either *controls* (inputs) or *displays* (outputs.) We will say more about them later.

4.2. User Event Handler and Callback Functions

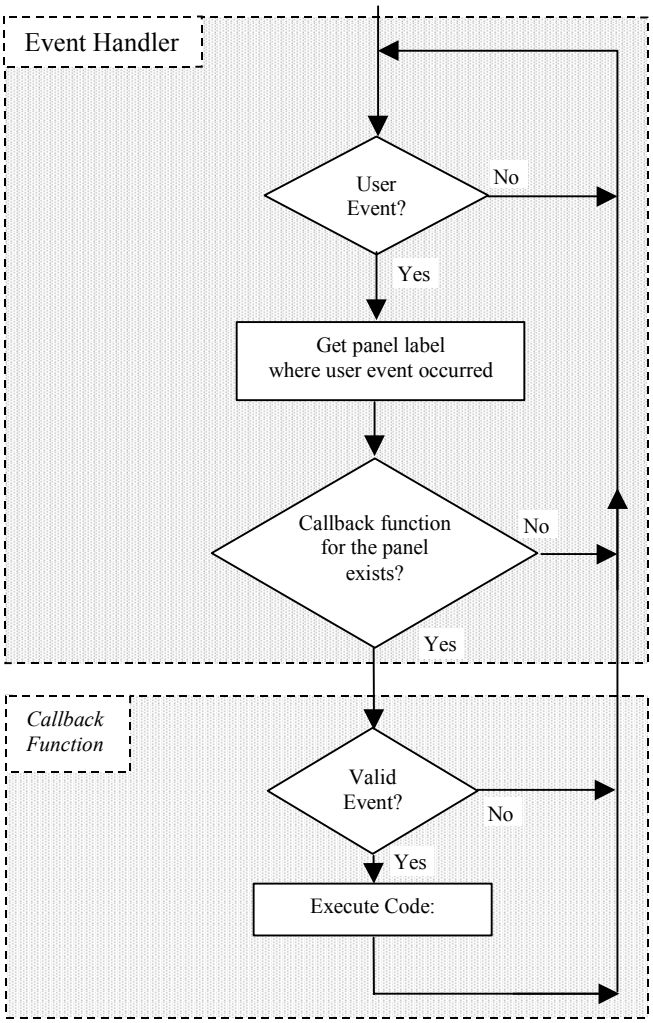
Since LW is event driven, at the heart of it lays the (user) *event handler* routine. You may find it helpful to think of the event handler as a very busy traffic cop that determines how to deal with the many user events that are created whenever you move the mouse or hit a key on the keyboard.

The event handler processes the user events by applying some very simple rules. First, it determines "where" the event was created, i.e. it obtains the panel name where the event occurred. It then checks if the panel had a procedure, in LW also known as a *callback function*, associated with it.

In case the event handler finds no such callback function associated with the panel, it simply ignores the user event and goes back to directing traffic or waits for the next user event to occur.

In case the event handler finds a callback function associated with the panel, it forwards various information about the event to that callback function and lets the callback function process the event. As far as the event handler is concerned, it is done with processing that particular event and it will go on waiting for the next event to occur. As you can see, while the event handler is very busy, it is really in the callback functions where all the action takes place, so let's look more closely at the callback functions.

Though the name callback function sounds rather intimidating, the structure of a callback function follows that of the standard C-functions, i.e., it has a function declaration, header and body. The function body usually is one large C-case statement that determines what should be done for a particular event. It's a lot like saying if this event happens do this, if that event happens do that and if an event occurs that nobody has thought about before, then ignore it. In other words, the callback function only processes *valid* events.



What is a valid event? As you have already read, any event that will be dealt with in the callback function is a valid event. Since there are a large number of possible user events and an even larger number of combinations thereof, for the sake of simplicity we restrict the valid user events to what is called "COMMIT" events: the user clicks with the left mouse button or hits the ENTER key. In other words, while LW has the capability to handle every type of user event, we will limit ourselves throughout this course to COMMIT events because they are adequate for most GUI situations.

Since what you have learned is very crucial for understanding LW programming, what follows is a reiteration and a flowchart. Be sure to study them carefully.

When a user event is received, the LW event handler executes the following steps:

- First, LW determines the name of the panel where the event occurred.
- Next, LW checks if a callback function has been associated with the particular panel to handle the event. If no callback function has been associated with the panel where the user event occurred, it is ignored and LW returns to the event handler and waits for the next user event to occur.
- If a callback function has been associated with the panel, LW enters it and begins to process the user event.
- First the callback function determines if the user event was *valid*. A valid event is one that falls into the EVENT_COMMIT category, i.e. left click and ENTER key. Events that do not fall into the EVENT_COMMIT category are ignored and LW returns to the event handler and waits for the next user event to occur.
- Finally, any additional code in the callback function is executed and after doing so, LW returns to the event handler and waits for another user event.

4.3. Control vs. Display Panels or Input vs. Output Panels

You may have noticed from the previous description, some panels do have callback functions associated with them while others do not. The difference can be understood whether a panel is being used as a *control* or a *display*, or to phrase it in terms of input and output, whether the panel is used to *input* or to *output* information. Consider the following examples.

An LED, a message or a graph panel displays the state of a system. They indicate the status of one or more variables and *output* that information. For example an LED panel can be used to display that the system is busy acquiring data. Generally speaking, you do not expect that the user will input information into such a display and, therefore, no valid user events should ever be created on such a *display* panel. From this it follows that panels which are chosen to output information do not need a callback function associated with them because you do not expect a user event to occur in these panels. As you have learned in the previous section, even if a user still creates an user event on a display panel, it will be ignored because no callback function has been associated with the panel.

On the other hand, you expect the user to generate a user event on a *control* panel such as a switch, a command button or any other *input* panel. To process such an event, you need a callback function associated with that panel. Without a callback function, the user has no way to interact with the program code because all the "important" stuff is executed in the callback functions. Almost all of the C programming that you will do in this course will consist of writing code for callback functions.

You may have noticed a certain vagueness in the description to what constitutes a control or a display panel. The reason for that is that most of the LW panels do not have an inherent property assigned to them which designates them a control or a display. Most of them can be used as either a display or a control or as both and how they are being used depends entirely on whether or not they have a callback function associated with them. Nevertheless, from a psychological viewpoint it is very important that you use the panels the way most users would expect them to operate. For example, LW allows you to use LED panels as controls and switches as displays. For the sake of clarity, refrain from doing so.

5. LW SOFTWARE: EXAMPLE 1

5.1. LW Components Overview

The LW software package consists of these components: *project window*, *uir-editor* and *C-compiler*. The *project window* keeps track of the various files, libraries and drivers needed to create a LW application. See
Figure 5-2 for an example of a project window.

You will design your GUI with the aid of the *user interface editor* (*uir-editor*) which creates a *uir-file*, which is short for *user interface resource file*. (See Figure 5-4.) The uir-editor contains a library of sample panels that you will rely on and you will also use it to place the panels, move, color, resize and rearrange or delete them.

The C-compiler will be used to display, alter, compile and execute the skeleton code created by the uir-editor. For a picture of the C-compiler window see Figure 5-8.

5.2. Example 1: Getting Started

You will now develop your first working LW application. This example is pretty simple and will consist only of a parent panel and a termination button. Nevertheless, it is the basis of every LW project that you will develop. Later on you will expand on this example and ultimately develop it into a multi-paneled GUI functioning as an adjustable clock. The following exercises will cover most of the concepts that you will need for all future assignments. Be sure to sit in front of a computer and actively follow the exercise along so that you become familiar with the environment.

5.3. Start LabWindows

- To start LW, click on the LW icon (see Figure 5-1) located on the Windows desktop, or you can also start it by selecting from the Window's taskbar:
  Start/Programs/LabWindows

Figure 5-1: LabWindows Icon

5.4. Project Window

Once you start LW, the first window you will see is the project window displaying all files that are part of your current project. If this is the first time you start LW, then the display will be empty (as shown in
Figure 5-2) because you have not added or created any files yet for your project.

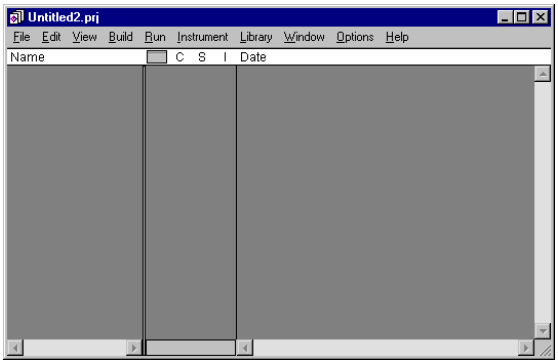


Figure 5-2: Empty Project Window

If at a later time, you choose to unload a project and want to start with a new one, select:

File / New / Project

Answer "yes" to the confirmation message and when the "Transfer Project Options" windows appears, make sure all options are checked.

5.5. UIR-Editor

The best way to start a project is to design first the GUI using the uir-editor. This can be a lot of fun because you don't have to worry (yet) too much about programming and instead have a chance to use your creative and esthetic skills.

- To start the uir-editor, select from the project window menu:
  File / New / User Interface

A window like the one shown in Figure 5-3 will open.

Figure 5-3: Empty User Interface Editor (uir-editor) Window

First, you need to create the parent panel, which will contain all your child panels.

- To add the parent panel select from the uir-editor menu:
  Create / Panel

Every GUI created in LW needs a method for termination. If you omit it, your only method to terminate your application might be the ON/OFF switch on the computer, which, needless to say, is not a very user and hardware friendly approach to programming!

Therefore, you must add at least one control that allows the user to terminate your application gracefully. (Note: LW will automatically add the standard Windows controls in the upper right hand corner of the parent panel such as the minimize, maximize and exit buttons. While the first two will work, the exit button does not work by default and you have to set some additional switches in LW. You will not use this exit button to terminate the program and instead always add your own as described in the following sections.)

Add a control for terminating the program by inserting a command button into the parent panel.

- In the uir-editor select:
  Create / Command Button

(If the "Command Button" choice is grayed or disabled, click on the top bar of the parent panel.)

- Select one of the command buttons and then drag it to your preferred location on the parent panel. (See Figure 5-4.)

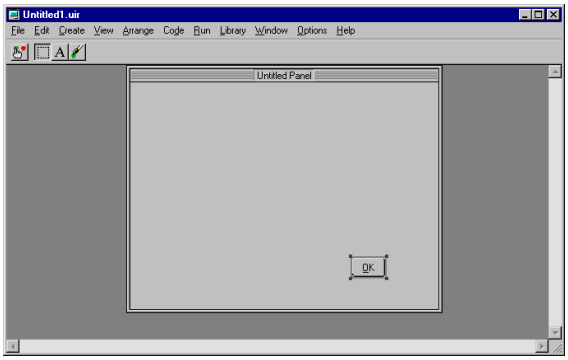


Figure 5-4: UIR-Editor

LW often displays a puzzling selection of panels for a particular subject. While their appearance is different, their functionality is almost always identical. For example, among the five choices of command buttons displayed, the functionality

is identical. The only difference is that the rightmost button allows you to paste a picture of your choice onto the button.

If you want to check or alter the properties of a panel, double click on it. You will now alter some of the properties of the command button that you just inserted.

- Double click on the "OK" button.

A new panel, the Edit Command Button panel (see Figure 5-5) opens displaying various default properties that LW has already assigned to the OK button.

You will now alter some of these properties: change the constant name of the panel, associate the panel with a callback function and then change the label on the button.

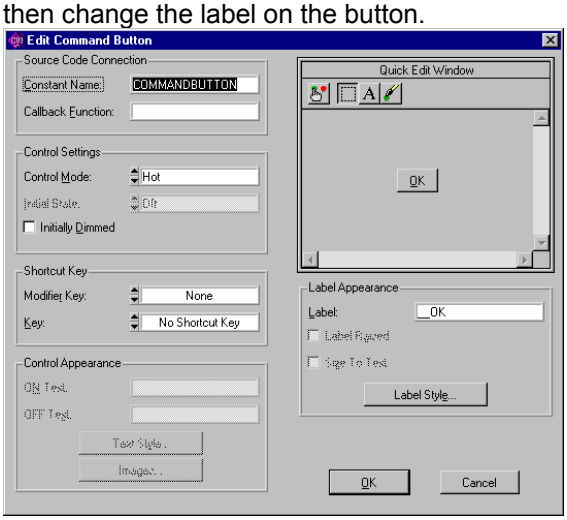


Figure 5-5: UIR-Editor Properties Panel

You have already learned that every panel must have a constant name. When a panel is created in the uir-editor, LW assigns it a default name. From the Edit Command Button panel, Figure 5-5, you see that the command button's default name is COMMANDBUTTON.

The default names are usually adequate and there is no need to change them. This is particularly true if you have only a small number of panels with similar functionality. On the other hand, things can become confusing when you have multiple panels with identical functions. Under these circumstances, LW will label additional command buttons as COMMANDBUTTON1, COMMANDBUTTON2, etc. Trying to remember which command button does what can become confusing and you probably should assign them more descriptive names.

- For this exercise, change COMMANDBUTTON to QUITBUTTON.

You also have learned that the difference between a control and a display panel is that a control panel always has a callback function associated with it. Since you want the command button to terminate the program when you click on it, it is clear that it is an input or control panel and, therefore, requires a callback function associated with it.

- In the Callback Function window of the Edit Command Button panel enter the name of the callback function which will be used to terminate the program. Call the function: *Bye*. (Note again the C-function convention, the first letter of the function is always capitalized.)

The last change that you will make to the QUITBUTTON involves its appearance in the GUI. When you look in the upper right hand panel of the Edit Command Button panel you see how the QUITBUTTON will be displayed in the GUI. Since it does not make much sense to call the termination control button "*OK*," change its appearance.

- In the Label Appearance section change the label from OK to "*END*" or something sensible.

- After you have made these three changes, close the Edit Command Button window by clicking on the OK button at the bottom of this window.

You should see a display similar to the one shown in Figure 5-6.

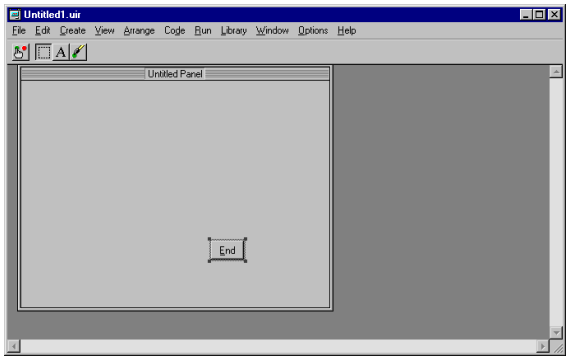


Figure 5-6: UIR-Editor

5.6. Generating Skeleton C-Code

So far you have only designed the GUI. No code has been generated yet that will execute the GUI and implement its functionality. You will now generate this code using the uir-editor.

Incidentally, the code generated by the uir-editor is called the *skeleton code*. On the one hand, the skeleton code is complete because it will run the GUI and even has structures for the callback functions. On the other hand, it is a skeleton code because only the structures for the callback functions exist. You still will have to add your own code to the callback functions to determine what will be done when a callback function is called with a valid user event.

Before LW can create the skeleton C-code you must save the newly created GUI.

- In the uir-editor select:
  File / Save As / enter some name

Make sure the files are saved in your network directory, i.e. at U:\LabWin.

- In the uir-editor, start the code generation by selecting:
  Code / Generate All Code

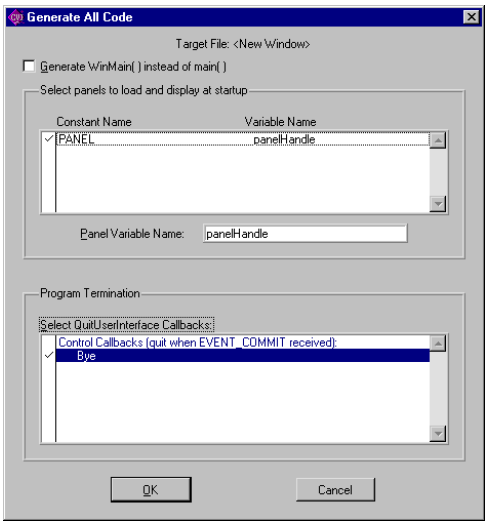


Figure 5-7: Generating the C-Code

The window shown in Figure 5-7 appears. It is a reminder to you that every parent panel must have a control that allows LW to terminate the program and clean up any windows associated with the parent panel. This is accomplished by calling the LW QuitUserInterface function.

The bottom panel of the window shown in Figure 5-7 lists all callback functions associated with your GUI and asks you if you want to use any of them to call the QuitUserInterface function. Your application consists of only one callback function, namely "*Bye*," which has been associated with the QUITBUTTON panel.

- Since we want to associate this callback function with the QuitUserInterface function, click to the left of Bye in the Program Termination / Select QuitUserInterface Callbacks window until a checkmark appears as shown in Figure 5-7.

Finally, when you click on the OK button at the bottom of the panel shown in Figure 5-7, the generation of the skeleton C- code begins

---

5.7. C-Compiler

After you have executed all the previous steps a new window opens, the *C-compiler* window, displaying the newly created skeleton C-code. It should look like the one shown in Figure 5-8.

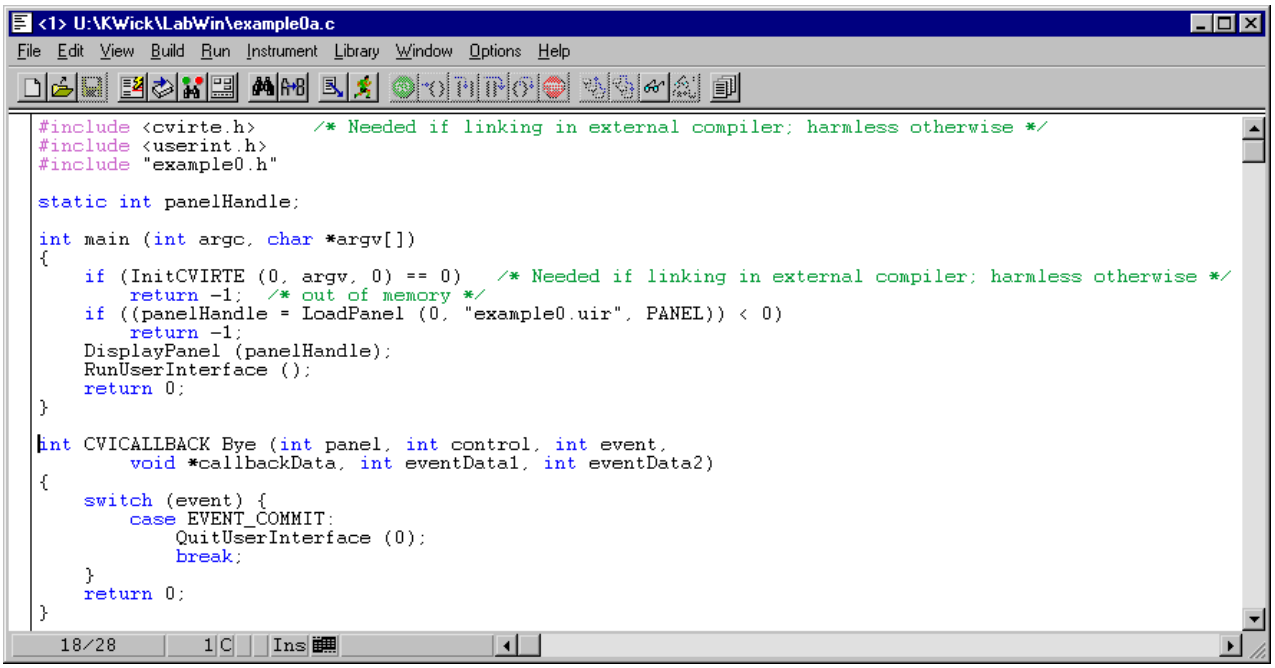


Figure 5-8: C-Compiler with Code

Figure 5-8 shows the C-code generated which will run your application. We will discuss the general structure of it later. Instead, let's try to execute the code. Before you can do so you must save various files that you have created and also add them to the project window.

First save the C-code:

- In the C-compiler select:
  File / Save As / some filename

Next add the C-code and uir-file to the project and then save it all:

- On your desktop, find the project window (see Figure 5-2).
- In the project window select:
  Edit / Add Files to Project / All

- Select and add the uir-file created previously in the uir-editor and the C-code that was just generated.

The project window should now look similar to Figure 5-9 and it should display the C-code file and the uir-file.

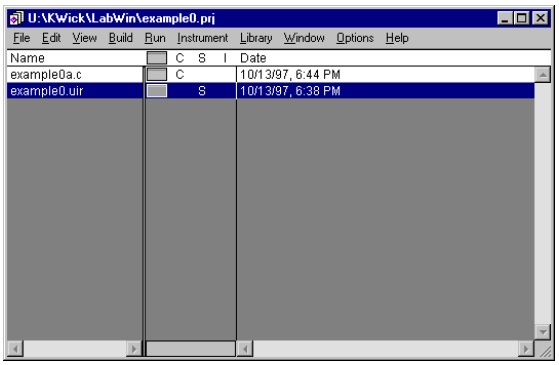


Figure 5-9: Project Window with Project Files

Finally, save the project itself:

- In the project window select:
  File / Save As / some filename

Now you are ready to execute your first LW program.

- In the project window select:

Run / Run Project

You should now see the GUI that you earlier designed, something similar to what is shown in Figure 5-10.

- Click on the END button to terminate it.

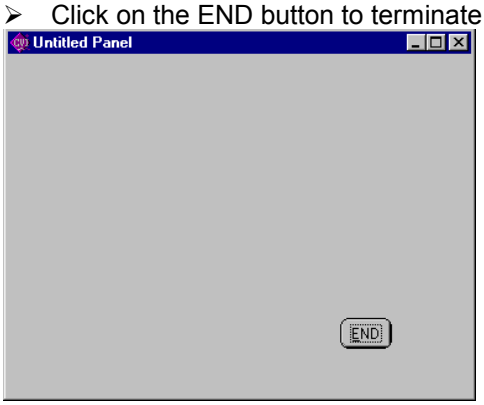


Figure 5-10: Your GUI

5.8. Changing the Appearance of the GUI

Running this application is probably not the most exciting program that you ever have come across, but this is only the start of your programming career. Therefore, you may want to change the appearance of the GUI to make it more attractive by changing some of its default colors.

- Go back to the uir-editor and click on its top bar to activate it.

In the upper left-hand corner of the uir-editor, you will see the icons like the ones shown in Figure 5-11.

- Click on the rightmost icon, the paintbrush

Figure 5-11: UIR Editor Tools

Move the mouse cursor over an object in the parent panel that you want to change the color of. You may choose the parent panel itself. When you right-click the mouse, a color palette will appear. Slowly move the mouse cursor over the color palette until you find the only esthetically acceptable color. Left click on your color choice and repeat this procedure for any other panel.

Important: When you are done choosing colors, be sure to select again the second icon from the left in the uir-editor which is the doted square, see Figure 5-11. If you forget to do so, you will not be able to add more panels to the parent panel or move panels.

5.9. Explanation of the C-Code

Before moving on to the next example, study the C-code that was generated. You do not need to understand every single command but you should have a general idea what each part does because soon you will have to modify the code.

It is easier to analyze the code if you break it into three segments. The first segment, shown in Program 5-1, contains the "include files" needed by LW and the global variable declarations. Of the three include files listed, the last one is particularly important because it contains the information about your GUI such as panel labels, colors and positions.

The last statement in this segment declares the global variable "panelHandle". This variable stores a numerical value that is assigned by Windows to the parent panel of LW. Note that the “static” key word is redundant since global variables are by default already static.

```
#include <cvirte.h>    /* Needed if linking in external compiler; harmless otherwise */
#include <userint.h>
#include "example0.h"

static int panelHandle;
```

Program 5-1: First Program Segment

The second segment, shown in Program 5-2, is the main-function of C. Ignore the arguments of the main function and the first if-statement block in "main," but pay attention to the last 5 lines of code.

- The LW "LoadPanel" function loads the uir-file created in the User Interface Editor. (Important: if you ever change the name of your uir-file in a project, make sure that the file name agrees with the one displayed on this line.)
- The "DisplayPanel" function displays your GUI on the desktop.
- The "RunUserInterface" runs your GUI's event handler routine. This is probably the most important line of code because it directs all user events to the appropriate callback functions. RunUserInterface( ) executes until the QuitUserInterface( ) is called from within a callback function.
- Finally, if a QuitUserInterface( ) call was received, the "return 0" statement terminates the program.

```
int main (int argc, char *argv[])
    {
    if (InitCVIRTE (0, argv, 0) == 0)
                          /* Needed if linking in external compiler; harmless otherwise */
        return -1;                    /* out of memory */

    if ((panelHandle = LoadPanel (0, "example0.uir", PANEL)) < 0)
        return -1;
    DisplayPanel (panelHandle);
    RunUserInterface ();
    return 0;
    }
```

Program 5-2: Main Function

The last segment, shown in Program 5-3, consists of the callback function "Bye". This function is being called by the "RunUserInterface" function when a user event occurs in the panel, which "Bye" is associated with, the QUITBUTTON panel. From the function arguments you can see that the callback function receives information about the panel and the type of user event generated.

The function itself contains a "case-switch" structure that filters the events. As it stands, it only processes EVENT_COMMIT user events, i.e. either left mouse button clicks or return key hits. Any other event will be ignored and control is returned back to the RunUserInterface function, which patiently waits for the next event to occur.

When an EVENT_COMMIT user event occurs, the code between the case statement and the break statement will be executed. In this particular case, the QuitUserInterface( ) function will be executed, which as previously stated, terminates the RunUserInterface( ) and ends the program.

```
int CVICALLBACK Bye (int panel, int control, int event, void *callbackData,
int eventData1, int eventData2)
    {
    switch (event)
        {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
        }
    return 0;
}
```

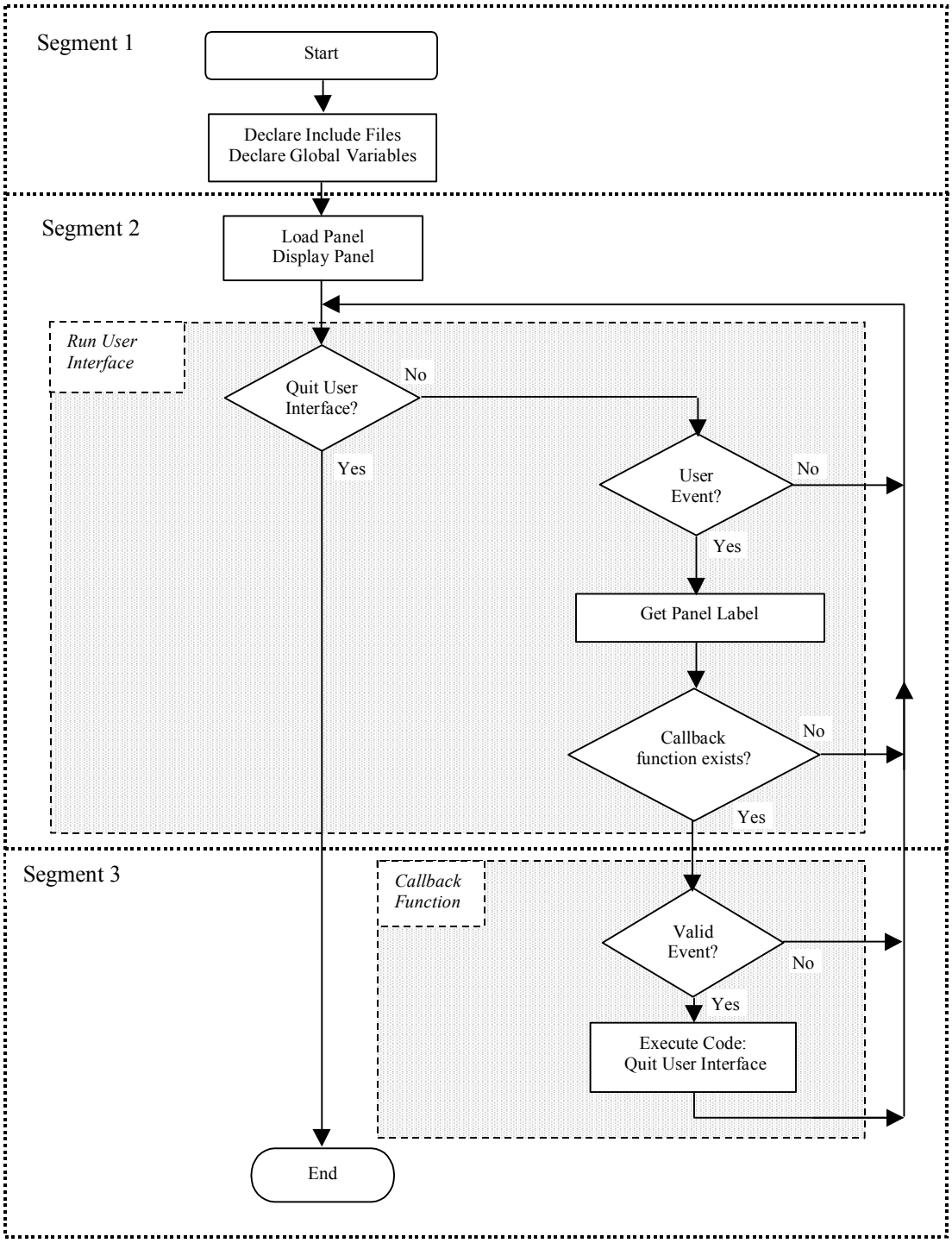
Program 5-3: Third Program Segment, Callback Function "Bye."

When you generate the callback function code, a blank line will be inserted between the EVENT_COMMIT case statement and the break statement. It is to remind you that this is the location where you will add any code that you wish to execute when a valid user event occurs. You will see more of that in the next exercise.

In this case, the QuitUserInterface( ) function has already been inserted by LW at this location. This happened when the code generator asked you a while ago (see Figure 5-7) if you wanted to specify a callback function to terminate the RunUserInterface event handler. By adding the checkmark in the window in Figure 5-7, LW automatically inserted this function for you.

5.10. Flowchart of the C-Code

To express what has been said in this chapters in a different way, study the flow chart below, which corresponds to the code above.



5.11. Conclusion

A lot of new material has been presented. Try to remember the most important steps:
First you use the uir-editor to design the GUI. Second, you generate the C-code. Finally, you (would) add code to the callback functions.

6. EXAMPLE 2: CONTROLS AND INPUTS

In this exercise you will enhance the GUI and code developed previously. You will add a toggle switch to the GUI and write some code in the new callback function that increments a variable when you operate the switch. Because you have not learned yet how to display numerical values in LW using display panels, you will use the ANSI "printf" function and the standard-output window.

6.1. Controls Panels Overview

LW provides various types of panels that can be implemented as controls. If you are curious, go back to the uir-editor and select the Create menu. The three most commonly used control panels are:

- Command Buttons
- Toggle Buttons
- Binary Switches

The functionality of these switches is pretty similar. Other, more complex controls, also to be found in the uir-editor under the Create menu, are:

- Numeric
- Ring
- List Box

Of these, the "numeric" panel is often used because it allows entering and displaying numerical values. Therefore, depending on its intended usage it will act either as a control or as a display or both and may or may not require a callback function. If it is used for output and input, then it must be implemented as a control because it will require a callback function for the input.

6.2. Adding a Binary Switch

Go back to uir-editor. The GUI from the previous exercise should still be displayed. (If this is not the case, find the uir-file in the project window and double click on it.)

- Click inside of the parent panel to activate it and then add a binary switch; from the uir-editor menu select:
  Create/Binary Switch (select any one you like, they all work the same)

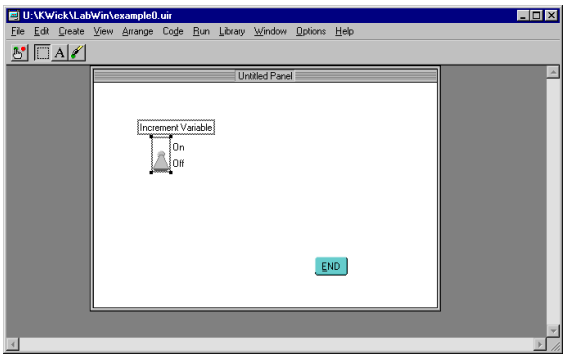


Figure 6-1: GUI with the Binary Switch

Change some of the properties of the binary switch:

- Add a callback function called "FunctionIncrement."
- Change the switch label to "Increment Variable" or some other descriptive comment.

To make these changes remember that you first must double click on the binary switch button to open the properties window of the binary switch. It is similar to the one displayed for the command button, see Figure 5-5. Unless you feel strongly about the default label BINARYSWITCH do not bother to change it.

Before generating the new C-code, be sure to save the modified uir-file.

- In the uir-editor, select from the menu:
  File / Save

6.3. Adding Skeleton C-Code for the New Callback Function

In the previous example, the entire C-skeleton code was generated from scratch, even the QuitUserInterface function was added at the appropriate place. This time, you will learn how to add only the skeleton code required for operating the binary switch to the already existing code.

It is tempting to regenerate the entire skeleton code, just as you did in the previous exercise. Nevertheless, you should be very careful doing so because a complete code generation will overwrite any changes previously made to the skeleton code. At times, this can be very annoying, especially if you have added code to your callback functions! Therefore, you should get used to generating code only for new or altered control panels.

To generate the code for the callback function for the binary switch you need to identify the target file where the new code is to be added to.

- In the uir-editor select:
  Code / Set Target File

Select the c-program from the previous exercise as the target file.

There are two methods to generate the code for the binary switch. Choose either one.

- Go back to the uir-editor and either left-click once on the binary switch and select from the menu:
  Code / Generate / Control Callbacks

  Or right-click in the uir-editor on the binary switch and select:
  Generate Control Callbacks

If you pay close attention, you might notice that the mouse cursor temporarily changes to a lightning bolt indicating that code is being generated.

```
#include <cvirte.h>    /* Needed if linking in external compiler; harmless otherwise */
#include <userint.h>
#include "example0.h"

static int panelHandle;

int main (int argc, char *argv[])
    {
    if (InitCVIRTE (0, argv, 0) == 0) /* Needed if linking in external compiler;
                  harmless otherwise */
        return -1; /* out of memory */
    if ((panelHandle = LoadPanel (0, "example0.uir", PANEL)) < 0)
        return -1;
    DisplayPanel (panelHandle);
    RunUserInterface ();
    return 0;
    }

int CVICALLBACK Bye (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event)
        {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
        }
    return 0;
    }


int CVICALLBACK FunctionIncrement (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event)
        {
        case EVENT_COMMIT:
```

```
            break;
        }
    return 0;
    }
```

Program 6-1: Third Program Segment, Callback Function "Bye."

The new code for the entire application is listed in Program 6-1. When you compare it with the code generated previously, see Figure 5-8, you will notice that the code is identical except for the new callback function "FunctionIncrement" that has been appended. During program execution, when you left-click on the binary switch in the GUI, this function will be called by the RunUserInterface function and everything between the lines “case EVENT_COMMIT:” and “break;” will be executed. As you can see from the above code, LW has inserted an empty line at that location to remind you to insert any code.

The purpose of this exercise is to increment a variable each time the binary switch is operated. The C-code for incrementing a variable named *x* is:

```
x = x + 1;
```

(This could also be written in C as: `x++;`)

- In the C-compiler, add this line of code (at the blank line) to the callback function FunctionIncrement.

Before you can use a variable in C, it must be declared in your code. Therefore, declare "*x*" globally.

- Towards the top of the code, right after: `static int panelHandle;` enter:

```
static int x;
```

Save the code and see if it executes without errors. Unfortunately, you will not see much happening when you operate the binary switch because you have not specified how to display the numerical value that the variable "*x*" takes on. You will have to wait till the next exercise to learn how to use panels to display numerical values. For now, you will use a quick and easy solution to output a variable, namely the good, old ANSI C-function “printf”.

To display an integer variable called “x” using "printf" use the following syntax:

```
printf("%d\n",x);
```

- Add this line of code to the FunctionIncrement, right after the `x = x+1;` statement.

Save the code and run the program again. While compiling the code, LW will ask you if you want to add the ansi_c.h include file to your code; answer with "yes."

When you now operate the binary switch, a new window, called the “standard input/output” window, opens up and displays the current value of "*x*." Click a few more times on the binary switch and watch the value of the variable x incrementing.

While this method of displaying numerical values is not as elegant as the one shown in the next exercise, it is quicker to set up, especially if you are familiar with ANSI C. It also illustrates that you still can run standard ANSI C-code in the LW environment. Therefore, if you do not know how to do something using a GUI you can always go back and do it in ANSI C. Also, you should remember this method because it can come in very handy for debugging code.

Finally study the flowchart shown in Figure 6-2. Compare it with the one shown in the previous chapter and notice the new callback function. To emphasize the callback function, this flowchart has been simplified; checks for valid user events and checks for callback functions associated with panel were omitted.

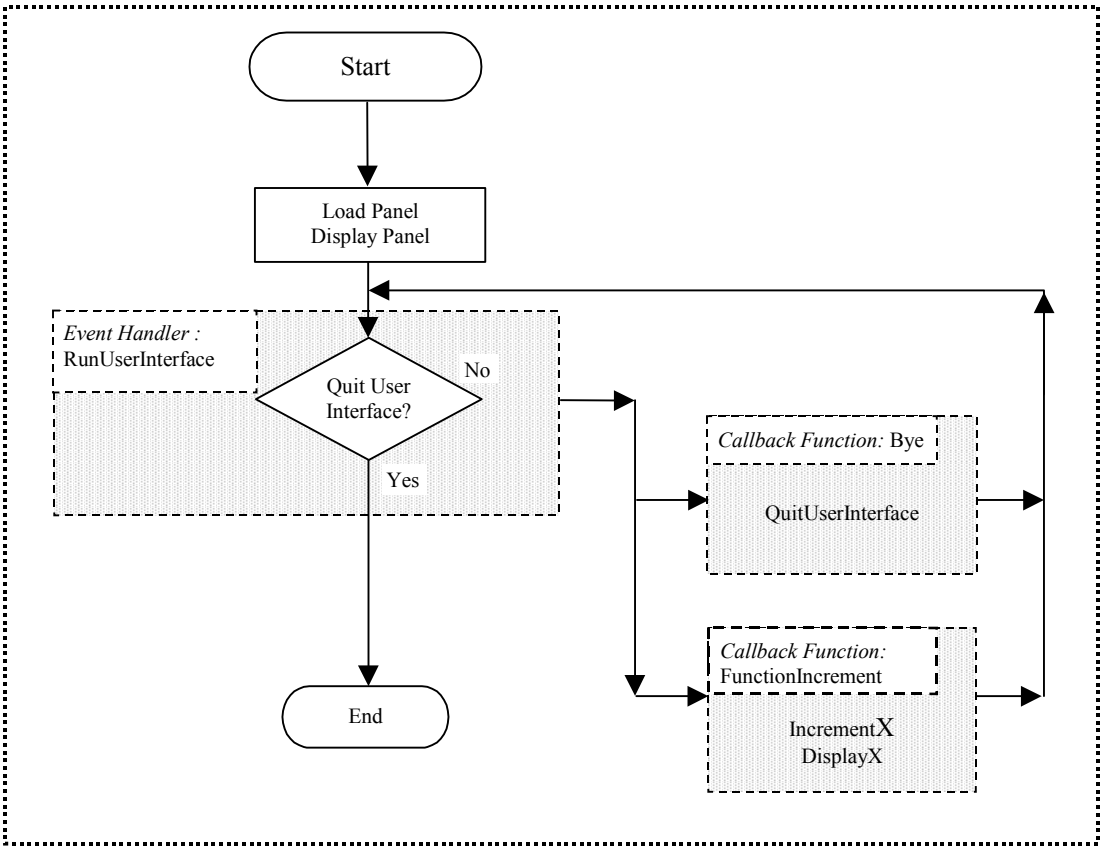


Figure 6-2: Simplified Flowchart

7. EXAMPLE 3: OUTPUT AND THE SETCTRLVAL FUNCTION

In this exercise, in addition to learning how to display numerical values in the GUI you will also learn how to use the LW function help utility. Because GUI functions are often very complex and involve many arguments, it is important that you become familiar with it.

7.1. Display Panels Overview

Display panels output information and require no input. Therefore, they do not require their own callback function and are usually updated by the callback function of a control. The most common types of displays are:

- LED,
- Numeric,
- String,
- Message and
- Graphs.

To explore them, select the CREATE menu in the uir-editor.

7.2. Adding A Display

Add a numeric display panel to the GUI to display the value of the incremented variable x (from the previous exercise) without having to resort to the standard input/output window.

- In the uir-editor, click inside the parent panel and then select
  Create / Numeric

Select the first choice, i.e. the panel shown on the right. 

You need to change some of the properties of the numeric display panel so double click on it.

- Change the *Control Mode* from “Hot” to “Indicator” to specify that it is a display and not a control.
- Change the data type so that it agrees with the type already specified for the variable “x” in the previous exercise, namely “Int”.
- Finally, change the numeric panel label to something meaningful such as “Incremented Variable”.
- When you are done, save your GUI.

It should look similar to the one shown in Figure 7-3.

Since this panel requires no callback function you do not need to generate any new code. Theoretically, you could immediately run your application but not much will happen when you toggle the binary switch because you have not yet linked the variable “x” to the display panel. You need a function that writes the content of the variable to the display panel, the LW equivalent of a "printf" function. They are provided by LW through the SetCtrlVal( ) function, and its related function, GetCtrlVal( ), which are the primary means of communicating with the panels.

7.3. LW Library Utility

Both the SetCtrlVal and the GetCtrlVal functions are relatively straight forward. Since they only have three arguments it is conceivable to memorize the type and order of arguments required. On the other hand, LW contains hundreds of functions, some of which have five or more arguments. Memorizing or looking up how to use them can become quickly cumbersome. Furthermore, as your application grows, the number of variables will grow rapidly and you will have to keep track of them. Wouldn't it be nice if you had a utility that would simplify all these tasks? The answer to these problems is the LW library utility, which not only displays information about each function and its argument, it declares and lists suitable variables for each argument and generates the C-code for you.

Typically, using the LW library utility is a two step process. First you select the function and fill in the various arguments; while you are doing this, LW writes the corresponding code for this function in a temporary window. During this process you may also create new variables and LW will, if you chose so, write declarations for the new variables directly to your program code. Second, when you have added all the necessary arguments to your function, you have the option of pasting the temporary code into your existing program.

Start the LW library utility.

- Selecting the following menu choice in the C-code editor:
  Library / User Interface / Controls, Graphs, Strip Charts… / General Functions / Set Control Value

It is unfortunate that a function such SetCtrlVal, which is used so frequently, is buried under all these subject listings. Nevertheless, once you get to it you should see a panel like the one shown in Figure 7-1. The top of the panel consists of the function arguments. Windows with a grayed out box below them, such as the "Status" one correspond to function return values; windows without that box, such as PanelHandle, ControlID and Value, correspond to function arguments. The white window at the very bottom displays the temporary C-code generated for the function. As you modify the arguments, this window is automatically updated.

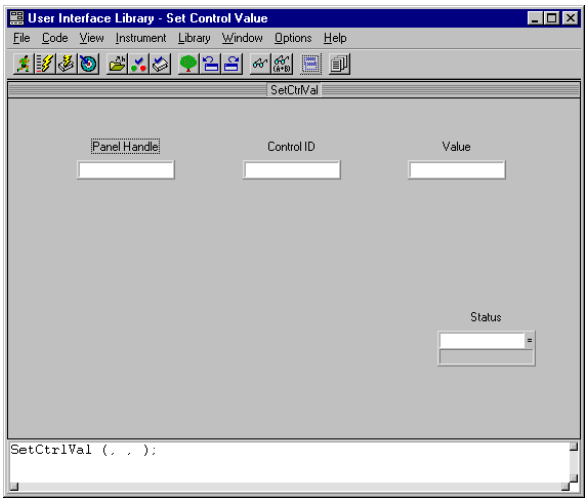


Figure 7-1: Library Utility for the SetCtrlVal Function

General information about the function can be obtained by right-clicking anywhere in the gray area. When you right-click in one of the function argument boxes, you receive specific information about the selected argument.

Now enter the appropriate values for the function arguments. Begin with the PanelHandle argument, which requires you to enter the name of the variable that stores information about the parent panel. You

could determine the name of that variable from studying the existing C-code but LW gives you an alternate method and instead lists all suitable variables so you can select the correct one.

- To do so, left-click in the window corresponding to the PanelHandle argument.
- From the library utility menu select:
  Code / Select Variable
- Click on: *Show Project Variables*

You should now see a listing of the suitable project variables. In case you forgot, the C-compiler has previously assigned the variable panelHandle as the variable that stores the constant value associated with the parent panel.

- Select it by double clicking on panelHandle.

Note how the new argument was added to the temporary function code in the bottom panel of the library utility window.

The second function argument requires you to enter the constant name of the panel which will display the numerical value. Unless you did change the default label that LW assigned to the numerical panel, the name should be NUMERIC. You could just enter that name (with some modification as you will see in a second), but it is better if you use the library utility and let it list all the constant names.

- Click inside ControlID window and select the following menu choice:
  Code / Select UI Constant

A list of all the constant labels is displayed.

- Double click on: PANEL_NUMERIC.

Note that while the child panel was called NUMERIC, LW automatically added the PANEL suffix to it, identifying it as the child panel of the parent panel called: PANEL. It may appear somewhat confusing to keep the names of the labels straight, i.e. is it PANEL_NUMERIC or simply PANEL? So remember that when you are working with code, you should always use the complete name, i.e. parent panel and child panel. This is yet another reason why it is advantageous to use this library utility because it automatically adds the correct name.

Finally, the last function argument requires you to enter the variable whose value you want to display, which is "x."

- Again, click in Value window and then select:
  Code / Select Variable

Double click on the variable "x."

The "Status" argument corresponds to the return value of the function. This value usually corresponds to an error code whose properties are listed when you right click in the return argument window. Among other things, it will tell you if the function executed successful or not. Whether or not you choose to store and process the return value is up to you. Clearly it is essential when you need to debug your code. In this particular exercise, you do not have to store the returned value, i.e. we simply assume that the function will execute correctly.

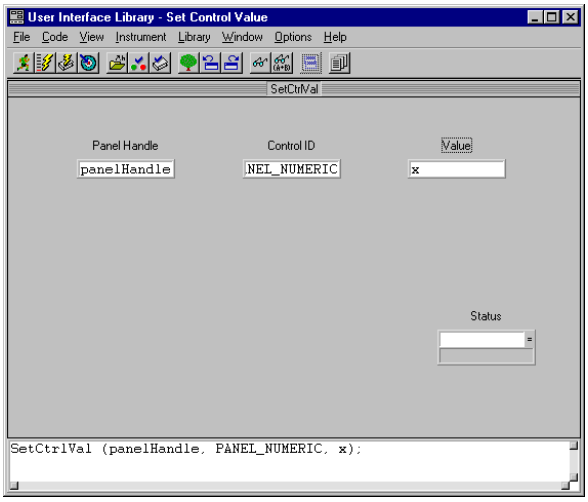


Figure 7-2: Library Utility for the SetCtrlVal Function Completed

Your library utility panel should now look like the one shown in Figure 7-2. As you can see, all appropriate arguments have been added to the function code and you are now ready to paste this line of code into your program. Before you can do so, you need to place the mouse cursor at the appropriate location in your program code where you want this line of code to be inserted.

Since the purpose of this exercise is to replace the printf function from the previous exercise with the SetCtrlVal function, move your cursor to that location:

- Go temporarily back to the C-editor and find the printf statement. Delete it. It is no longer needed.
- Insert an empty line at that location. This is where the new code will be inserted.
- Make sure the cursor stays at the beginning of this line before you return to the library utility panel.

- In the library utility panel, select from the menu:
  Code / Insert Function Call

The line of code shown at the bottom of the utility window has now been inserted in your program.

- Close the library utility panel by clicking on the X in the upper right hand corner of the panel.

The callback function for the binary switch, FunctionIncrement, should now look similar to the one shown in Program 7-1. As you can see, the library utility inserted one new line of code, the SetCtrlVal function.

Looking back, it certainly appears like a lot of complicated work has been done for inserting just one line of code. It would have been much simpler just to type the line of code into your program. This may be true in this particular case where the function is relatively simple and the choice of variables pretty straightforward. Nevertheless, once you get used to using the library utility, it can save you a lot of time and trouble, as you will see.

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            x = x + 1;
            SetCtrlVal (panelHandle, PANEL_NUMERIC, x);

            break;
        }
    return 0;
    }
```

Program 7-1: The Callback Function "FunctionIncrement."

Save the modified code and run the program. A GUI similar to the one shown in Figure 7-3 will appear. Notice how the variable increases when you flip the binary switch.

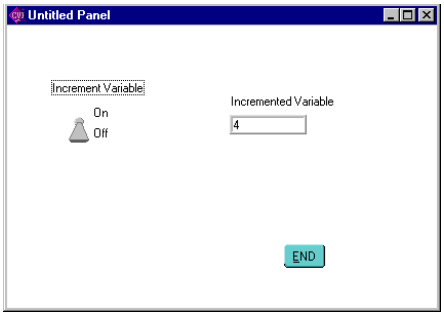


Figure 7-3: GUI with the Binary Switch and Numerical Display.

8. EXAMPLE 4: INPUT AND THE GETCTRLVAL FUNCTION

In this exercise you will learn how read a value from a panel. You will use this method if you want to read a numerical value that was entered into a numerical panel or, as in this exercise, the status of a switch. You will also become more familiar with the library utility.

8.1. The GetCtrlVal Function

While the application from the previous exercise was executing without any errors, it still had one shortcoming. The variable "*x*" was incremented each time you changed the state of the binary switch, i.e. when switching from ON to OFF and when switching from OFF to ON. Can you change the code so that the variable increments only when you switch from OFF to ON? Though it may sound like just another exercise, this type of situation is fairly typical. Usually, you want something to happen only when a switch is in the ON position, for example, collect data when the switch is ON. It should be easy to change the code by including an if-statement, if you can tell whether the switch is ON or OFF. This can be accomplished by GetCtrlVal function, which reads a value back from a panel. This function is very similar to the SetCtrlVal function from the previous exercise.

You may wonder what kind of value is returned by the binary switch when it is ON or OFF. Double click on the binary switch in the uir-editor; notice from the property panel shown in Figure 8-1 that in the section labeled "Control Settings" the binary switch was assigned a default data type of integer with the ON state being 1 and the OFF state being 0.

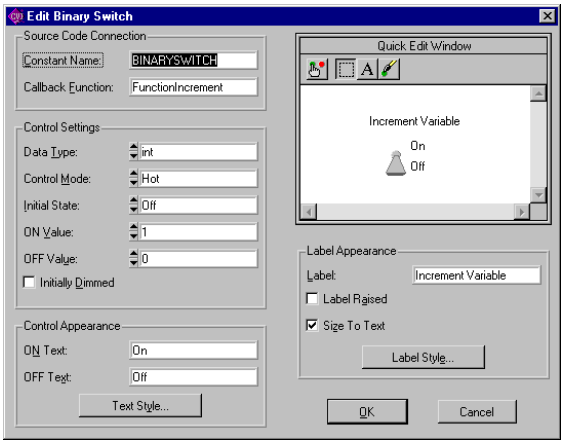


Figure 8-1: Binary Switch Properties Panel

- To add the GetCtrlVal function to your code, use the library utility by selecting:
  Library / User Interface / Controls, Graphs, Strip Charts…/ General Functions / Get Control Value

From the previous exercise, you should be able to determine the values for the first two arguments.

- Set the PanelHandle to PanelHandle.
- Select PANEL_BINARYSWITCH for the ControlID.

8.2. Declaring Variables through the Library Utility

The third argument, “Value”, relates to the variable that will store the state of the binary switch. You want to store it in a new variable, called “iOnOff,” that needs to be declared first. Instead of entering the code for the variable declaration by hand as you did a couple of examples ago for the variable "*x*", you will use the library utility to do so.

- Click in the third argument window, the “Value” window, of the library utility panel and enter: iOnOff.

- To declare the variable "iOnOff", select from the panel’s menu:
  Code / Declare Variable

A new panel, like the one shown in Figure 8-2 opens.

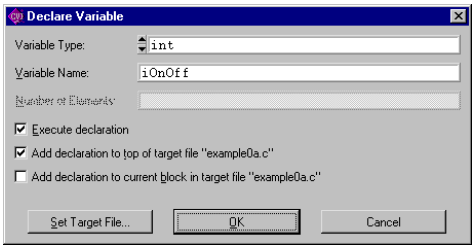


Figure 8-2: Declaring Variables through the Library Utility

You have already seen (in the properties panel, Figure 8-1) that the variable containing information about the state of the binary switch is of type “int.”

- Therefore, check that in the variable declaration window, Figure 8-2, iOnOff matches this type.

- If it isn’t already selected, make sure that the checkmark “Add declartion to the top of target file ” has been selected. Using this option will make the variable declaration a global one. (Be sure to have the C-compiler windows open.)
- Click on “OK”.

The declaration for the variable “iOnOff” has now been inserted at the top of your code.

When you return to the library utility panel note how the LW library utility automatically inserted the address operator, &, in front of the iOnOff variable in the third function argument. It did so because the GetCtrlVal function requests a pointer to the third argument and LW automatically converted the iOnOff variable to the appropriate type for you.

8.3. Adjusting the Code in the Callback Function

You are now ready to insert the new function call into your code. Before doing so, think for a while and try to figure out where it should be inserted. Since the purpose of this exercise is to increment the variable only when the switch is going from OFF to ON, you must check on the status of the switch right before the increment statement and then, depending on the status of the switch, increment the variable.

After you have selected the appropriate location in your C-code, insert the code for the GetCtrlVal from the library utility:

- Select Code/Insert Function Call.

Go back to the c-code editor and check if the changes made to your code by the library utility were appropriate. You should see the following line of code that has been inserted at the beginning of your program:

```
        static int iOnOff;
```

This corresponds to the (global) variable declaration for iOnOff.

With the GetCtrlVal function you have now a method to determine whether the binary switch ON or OFF. Nevertheless, you still need to add an if-statement if you want the variable "*x*" to be incremented only when the switch is in the ON position. Modify the code in the FunctionIncrement callback function as shown in Program 8-1:

```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panelHandle, PANEL_BINARYSWITCH, &iOnOff);
            if( iOnOff)
                {
                x = x + 1;

                SetCtrlVal (panelHandle, PANEL_NUMERIC, x);
                }

            break;
    }
    return 0;
}
```

Program 8-1: The Callback Function "FunctionIncrement."

Save and run the program. It should now increment only when the switch is going from OFF to ON.

Keep in mind that in this exercise, you read the state of a binary switch. Reading the input provided by a numerical panel would have been a similar procedure. You use the GetCtrlVal function to read the numerical value that was input and then you store that value in a variable.

9. EXAMPLE 5: TIMER

In the previous examples the callback functions were always responding to user events created by the mouse or the keyboard. In this example you will make use of another type of valid user event, the clock ticks (called EVENT_TIMER_TICK) caused by an adjustable software timer that is part of LW. After you have added the timer to your GUI, it will call its callback function at preset time intervals and thereby executes any code in the callback function. As you will see, it's rather easy to implement and set up a timer. In this example you will again modify the application from the previous example so that it increments the variable "*x*" automatically each second when the binary switch is in the ON position.

9.1. Adding a Timer to Your GUI

Go back to the uir-editor and add the timer.

- Select:
  Create/Timer

You may place this control anywhere on your GUI because this is the ONLY control that will NOT be displayed while the program executes! Its presence in the uir-editor is a reminder that this control will run in the background.

Since this is a control, you need to associate a callback function with the timer which will be called at regular time intervals.

- To set the properties of the timer, in the uir-editor, double click on the timer icon.

The properties panel as shown in Figure 9-1 will be displayed.

- In the Callback Function window enter: FunctionClock

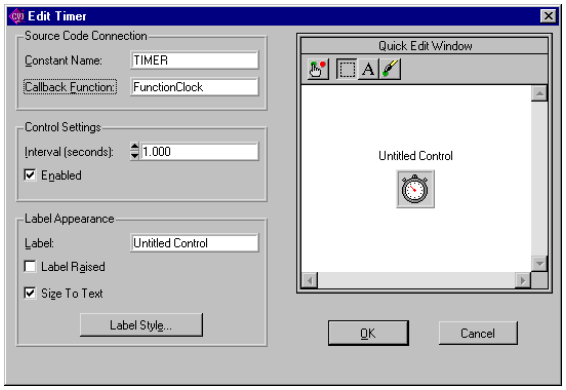


Figure 9-1: Timer Property Window

While you already have the properties window open you should examine the other choices. The timer interval is already set to 1.0 second, so there is no need to change that. Also, since the timer is not displayed during program execution, there is no need to change the label of the timer.

- Close the property panel by clicking on the OK button.

- Generate the callback function code for the timer by right-clicking on the timer icon and by selecting:
  Generate / Control Callback

9.2. Adding Code to the Timer Callback Function

A new callback function has now been added to your code. It should look like the one shown below:

```
int CVICALLBACK FunctionClock (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_TIMER_TICK:

            break;
        }
    return 0;
    }
```

Program 9-1: Callback Function "FunctionClock"

If you were to execute the program at this point, the application would call the FunctionClock function every second and execute the statements between “case EVENT_TIMER_TICK:” and “break;”. As you can see from the code segment in Program 9-1, no code exists between these lines so you will need to add the appropriate statements.

First, you want to increment the variable "*x*" at each timer tick.

- Therefore, you need to add the statement:

```
    x = x + 1;
```

immediately after the case EVENT_TIMER_TICK statement.

Second, you want this statement to be executed only when the binary switch is on. Similar to the previous example you need to enclose the increment statement inside an if-statement that checks if the binary switch is in the ON position. The partial program listing, Program 9-2, should accomplish all these tasks. Study it and make the appropriate changes to your code.

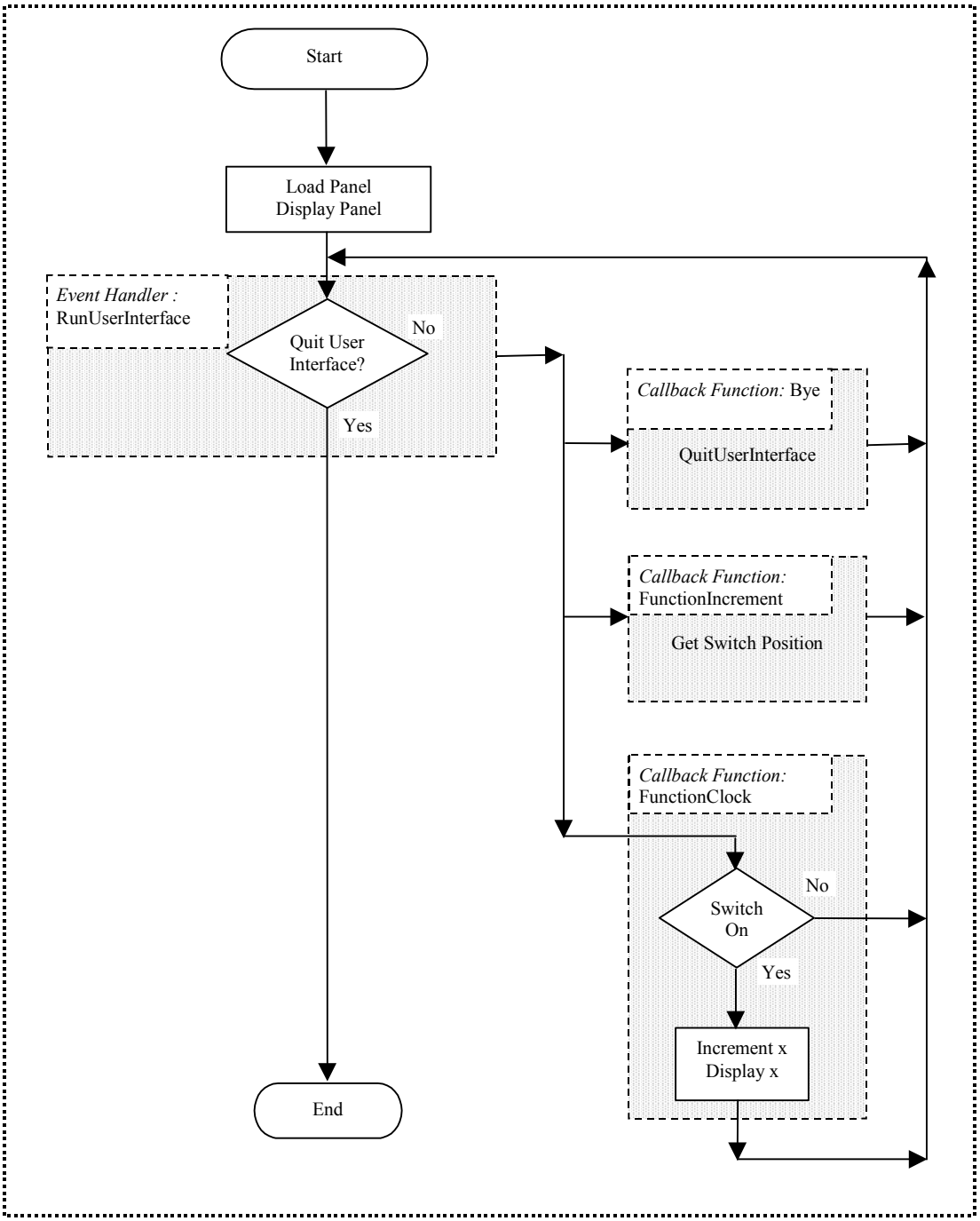
```
int CVICALLBACK FunctionIncrement (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panelHandle, PANEL_BINARYSWITCH, &iOnOff);

            break;
        }
    return 0;
    }

int CVICALLBACK FunctionClock (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_TIMER_TICK:
            if( iOnOff)
                {
                x = x + 1;

                SetCtrlVal (panelHandle, PANEL_NUMERIC, x);
                }

            break;
        }
    return 0;
    }
```

Program 9-2: Complete Callback Functions Code

Since the variable is now incremented in the timer callback function "FunctionClock," you no longer need to increment it in the callback function "FunctionIncrement." Therefore, as shown in Program 9-2, remove the increment statement and any other code directly related to it from the "FunctionIncrement" function.

Save the code and execute it. Test the GUI. You should see the variable incrementing every second when the binary switch is in the ON position. Also observe, as mentioned previously, that you do not see the timer icon in the GUI while it is executing.

Shown below is a flowchart of your application. You can see the how the callback functions interact with the RunUserInterface.



9.3. Multiple Timers

Before you go on to the last example, there are two important issues about timers that you should understand. The first involves the use of multiple timers.

Situations can arise when you need to call one function at one time interval while you want to call another function at another time interval. For example, you might want to take data at very short time intervals but you want to display a running average of the data only every few seconds. There are two ways to deal with such a situation: you either use multiple independent timers or you use one timer and the execute the various events after a predetermined number of clock ticks.

Usually it is much easier to work with multiple independent timers. In that case each timer has its own callback function and executes its code at its preset rate. This method works fine as long as it is not important if the different timers are not exactly in phase or that their phase may change during program execution due to one timer slowing down or speeding up.

Sometimes you need to trigger two events at different rates but with the trigger of one event depending on the trigger of the other. For example, assume that whenever the first function has been called four times, the second function should be called once. Using two independent timers with one timer interval being four times longer that the other might work under most circumstances. Nevertheless, there is no guarantee that one of the timers could not fall behind and that you could receive sometimes five clock ticks (or three) during one interval.

If that is a serious concern then add only one timer to the GUI and in the callback function count the clock ticks. Execute one of the events each time but the other only if the number of clock ticks elapsed is a multiple of four.

9.4. Time Intervals

The second issue involves the danger associated with setting short time intervals. If the time interval is close to zero, the computer will dedicate all its CPU time for the timer's callback function. In such a case, the computer will have no time left to process any other user events such as the ones responsible for stopping or adjusting the timer or terminating the application. Under such circumstances it is very likely that the computer will lock up and you may have to reboot the machine.

In addition, you should also be aware that with shorter time intervals the accuracy of the clock decreases because other processes with higher priority may prevent the CPU from returning to your process.

Finally, you should not only think in terms of the time interval set but also how long it takes to execute the task. For example, under normal circumstances, setting the time interval to 1/10 of a second should pose no problems for the lab computers. On the other hand, if the task that you execute every 1/10 of a second takes close to 1/10 of a second (or even longer), then you may lock up the computer. In that case the CPU may receive a new CLOCK_TICK message before it has completed the old one. So think carefully before setting the timer interval and prevent your application from being able to set it to zero!

10. EXAMPLE 6: READING AND SETTING PANEL ATTRIBUTES

The final example teaches you how to adjust the attributes of panels. You will add a control knob to your GUI allowing you to adjust the timer interval while the program is running. Later you will learn how to change the attributes of a panel.

As you have already learned, panels can be used to input or output numerical values using the GetCtrlVal and SetCtrlVal functions. In addition, each panel also has a set of attributes that can be read back and changed using the appropriate functions. Probably the most obvious attributes of panels are color, size and position. Nevertheless, it is rare that you want to change these attributes while executing code. An example of an attribute, which is often changed through code, is the possibility of *dimming* of a panel. If this is applied to an LED panel, it appears as if you are turning the LED on and off. You can also dim certain controls to indicate that this particular choice is not available. The two functions for reading and controlling attributes are GetCrtlAttribute and SetCtrlAttribute. You will use them to adjust the time interval of the timer while the program is executing.

10.1. Adding a Control Knob

Again you will use the program developed in the previous example and modify it. First add a control knob to the GUI so that you can set the time interval. Since you have already added various other controls to your GUI there is really nothing new in this part, just the combination of various tasks that you have already learned.

- First go back to the uir-editor and select: *Create / Numeric / Knob*

Change the properties of this knob:
- Add a callback function; call it "FunctionSetTimeInterval" or any other appropriate name.
- Change the label to Clock Time Interval.
- Set the data type to double.

Edit the control knob's range values. They will determine the timer interval's maximum and minimum values:
- Click on Range Values.
- Set the Minimum to 0.1 seconds. Do not forget this or the computer will lock up if the minimum is left at 0!
- Maximum: 10
- Set the Increment: 0.1. This corresponds to the amount that the numerical value will increase if the user chooses to click on the arrows in the numerical display located directly below the knob. (As you can see from Figure 10-1, the user has two methods for selecting a value. She can either turn the knob with the help of the mouse or she can click on the arrows on the display directly below it.)
- Set the Range Checking to "Ignore." (This option is only relevant when a variable is setting the numerical value and could fall outside the range of allowable values.)

After closing various property panels, your GUI should look similar to the one shown in Figure 10-1. Note how the knob's ranges have already been adjusted in the display.

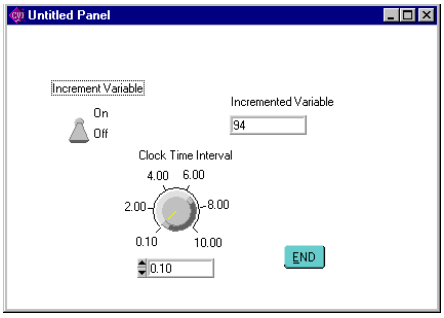


Figure 10-1: GUI with the Binary Switch

- Generate the code for the callback function FunctionSetTimeInterval and, similar to example 4, read the knob setting using the GetCtrlVal function into a variable (that you have to declare) of type double. (If you forgot go back to example 4.)

In case you are not sure if everything is working correctly you might want to add a printf statement to the code, similar to example 2, at the location where you read in the knob value. Run the application and check that, each time you change the knob settings, the new value is printed.

10.2. Setting the Control Attribute

You can not use the SetCtrlVal function to set the timer interval because it is considered to be an attribute. Therefore, you must use the SetCtrlAttribute function to change the timer's attribute.

- In the C-editor select the following menu to open the library utility:
  Library / User Interface / Controls, Graphs, Strip Charts…/ General Functions / Set Control Attribute

- Enter the appropriate values for the PanelHandle and the ControlID. They should be familiar to you. The PanelHandle again refers to the parent panel and the ControlID refers to the panel whose attribute you want to change, i.e. the TIMER.

Changing the attributes of a panel is a little bit trickier because there are numerous attributes for each panel.

- To get a listing of all the attributes and to find the one we want to change, left click in the library utility Control Attribute Window.

A new panel opens, similar to the one shown in Figure 10-2. It lists the control attributes for various panels that can be changed. Select the timer panel.

- At the very top of the panel, select the "Control type": Timer
- In the “Attributes” window select: Control Settings / Timer Interval. As shown in Figure 10-2 you will be given a short description of this attribute.
- Hit ok to close the panel and to return to the library function utility.

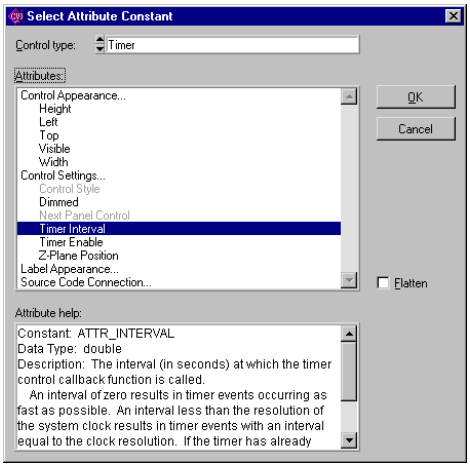


Figure 10-2: GUI with the Binary Switch

Back in the library utility, link the attribute argument to the variable that stores the timer interval value as set by the control knob, i.e. the variable that you declared in the first part of this exercise for the GetCtrlValue.

- In the Attribute Value window select:
  Code/Select Variable

Select the appropriate variable.

- Finally, in the C-compiler, move the mouse cursor to the callback function associated with this control knob and insert this piece of code at the appropriate place:
  Code/Insert Function Call

Save your code and run your application. If you have problems compare your code with the one shown in Program 10-1.

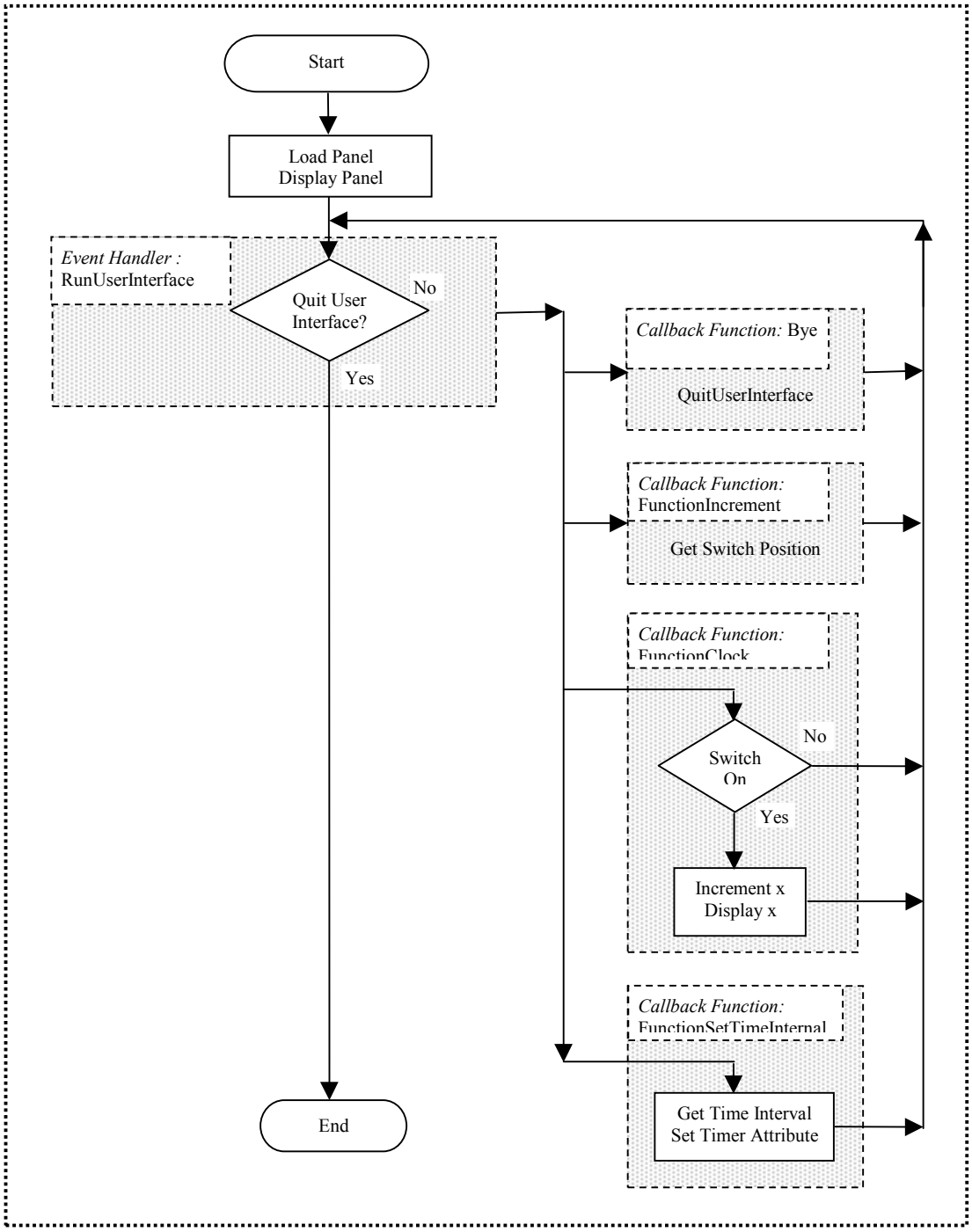
```
static double dTimeInc;
static int iOnOff;
#include <ansi_c.h>
#include <cvirte.h>        /* Needed if linking in external compiler; harmless otherwise */
#include <userint.h>
#include "example0.h"

static int panelHandle;
static int x;

int main (int argc, char *argv[])
    {
    if (InitCVIRTE (0, argv, 0) == 0) /* Needed if linking in external compiler; harmless
otherwise */
        return -1; /* out of memory */
    if ((panelHandle = LoadPanel (0, "example0.uir", PANEL)) < 0)
        return -1;
    DisplayPanel (panelHandle);
    RunUserInterface ();
    return 0;
    }

int CVICALLBACK Bye (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            QuitUserInterface (0);
            break;
```

```
        }
    return 0;
    }


int CVICALLBACK FunctionIncrement (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panelHandle, PANEL_BINARYSWITCH, &iOnOff);

            break;
        }
    return 0;
    }

int CVICALLBACK FunctionClock (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_TIMER_TICK:
            if( iOnOff)
                {
                x = x + 1;

                SetCtrlVal (panelHandle, PANEL_NUMERIC, x);

                }

            break;
        }
    return 0;
    }

int CVICALLBACK FunctionSetTimeInterval (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
    {
    switch (event) {
        case EVENT_COMMIT:
            GetCtrlVal (panelHandle, PANEL_NUMERICKNOB, &dTimeInc);
            SetCtrlAttribute (panelHandle, PANEL_TIMER, ATTR_INTERVAL, dTimeInc);
            break;
        }
    return 0;
    }
```

Program 10-1: Complete Program Listing

In case you had some problems with LW, you will find a working copy of this code and the GUI in the "U:\pub\LW\Examples" directory on your U-drive. The files are called: Example6. They are write protected meaning you can read but not modify them. If you want to modify them, open the files using LW and then use the File / Save Copy As… option and save them to your directory.

Shown below is the flowchart for the entire application.



11. CONCLUSION AND CONVENTIONS

11.1. Beginning a New Project

You have been exposed to a lot of new information. Here are some guidelines on how you should approach a new project.

First think what your inputs and outputs are. It may help to make flow charts like the ones shown in the previous examples to determine what should be done in the callback functions.

Second, start in LW by designing the GUI in the uir-editor. As you arrange all the panels you get a clearer idea about which ones will act as controls and which ones will be used as displays. Add the names for the callback functions to all your controls. Do not forget include a control to terminate your application!

Third, generate all the C-code. If this is the first time that you generate the code for your project, then specify the name of the control that will be used for the QuitUserInterface function. You may want to run the application just to see if it executes and terminates without errors. (Of course at this stage, none of the other controls operate yet.)

Fourth, add any code to the callback functions. Run the application as you add new code to each callback function to minimize confusion in case it does not work.

Finally, remember to save your application under different names, i.e. use version numbers, so that you can always go back when you mess up.

11.2. C-Naming Conventions

What follows is a list of C-naming conventions that are widely used today. Though they are only conventions you should use them to minimize confusion. Nevertheless, you are free to violate them at your own risk.

Constants in C are always capitalized, for example: *CONSTANT*.

The first letter of a function name is capitalized, such as in *Function*.

A variable has the first letter of its name capitalized and has a lower case letter prefix to indicate the variable type. For example, *iNumber,* is a variable of type Integer and *fValue*, is variable of type float.

APPENDIX

TABLE 1: Predefined Functions

The C run-time library contains about 400 functions, divided into the categories listed below: (Note that functions beginning with an underscore are QuickC functions and are not part of the ANSI C set.)

Category	Contents	Functions
Buffer manipulation	Manipulate areas of memory on a character basis	_fmemccpy _fmemicmp memccpy memicmp movedata _fmemchr _fmemmove memchr memmove swab _fmemcmp _fmemset memcmp memset
Character classification	Test individual characters	isalnum iscntrl islower isspace toascii toupper isalpha isdigit isprint isupper tolower _toupper isascii isgraph ispunct isxdigit _tolower
Data conversion	Convert numbers to strings and vice versa	atof ecvt ltoa atoi fcvt strtod atol fieeetomsbin strtol _atold fmsbintoieee strtoul dieeetomsbin gcvt ultoa dmsbintoieee itoa
Directory control	Manipulate directory structure and information	chdir getcwd _makepath _searchenv _chdrive _getdcwd mkdir _splitpath _fullpath _getdrive rmdir
File handling	Manipulate files	access fstat _makepath rename stat chmod _fullpath mktemp_searchenv umask chsize isatty remove setmode unlink filelength locking _splitpath
I/O - Streams	Stream I/O routines	clearerr fgetpos fread getchar rmtmp fclose fgets freopen gets scanf fcloseall fileno fscanf getw setbuf fdopen flushall fseek printf setvbuf feof fopen fsetpos putc tempnam ferror fprintf _fsopen putchar tmpfile fflush fputc ftell puts ungetc fgetc fputchar fwrite putw vfprintf fgetchar fputs getc rewind vprintf
I/O - Low-level	Low-level I/O routines	close dup2 open tell creat eof read umask dup lseek sopen write
I/O - Console and Port	Console and port I/O routines	cgets cscanf getche inpw outp putch cprintf getch inp kbhit outpw ungetch cputs
Internationalization	Localization routines	localeconv setlocale strcoll strftime strxfrm
Math	Math routines	abs coshl hypot modf tanl acos cosl hypotl modfl y0, y1, yn acosl dieeetomsbin j0, j1, jn pow _y0l, _y1l, _ynl asin div _j0l, _j1l, _jnl powl asinl dmsbintoieee labs rand atan exp ldexp _rotl atanl expl ldexpl _rotr atan2 fabs ldiv sin atan2l fabsl log sinh bessel fieeetomsbin logl sinhl cabs floor log10 sinl cabsl floorl log10l sqrt ceil fmod _lrotl sqrtl ceill fmodl _lrotl srand _clear87 fmsbintoieee matherr _status87 _control87 _fpreset _matherrl tan cos frexp max tanh cosh frexpl min tanhl

Memory allocation	Allocate, free, and reallocate memory	alloca _bmalloc _fheapset _heapmin _nfree _bcalloc _bmsize _fheapwalk _heapset _nheapchk _bexpand _brealloc_fmalloc _heapwalk _nheapmin _bfree calloc _fmsize hfree _nheapset _bfreeseg _expand free malloc _nheapwalk _bheapadd _fcalloc _frealloc _memavl _nmalloc _bheapchk _fexpand _freect _memmax _nmsize _bheapmin _ffree halloc _msize _nrealloc _bheapseg _fheapchk _heapadd _ncalloc realloc _bheapset _fheapmin _heapchk _nexpand stackavail _bheapwalk
Miscellaneous	Miscellaneous functions	assert putenv srand getenv rand strerror longjmp _searchenv _strerror perror setjmp swab
Process control	Manipulate processes and programs	abort execve spawnl atexit execvpe spawnle _c_exit exit spawnlp _cexit _exit spawnlpe execl getpid spawnv execle longjmp spawnve execlp raise spawnvp execlpe setjmp spawnvpe execv signal system
Searching and sorting	Binary search, linear search, and quicksort routines	bsearch lfind lsearch qsort
String manipulation	String functions	_fstrcat _fstrncmp _fstrtok strcspn strncmp _strtime _fstrchr _fstrncpy _fstrupr _strdate strncpy strtok _fstrcmp _fstrnicmp _nstrdup strerror strnicmp strupr _fstrcpy _fstrnset sprintf _strerror strnset toascii _fstrcspn _fstrpbrk sscanf strftime strpbrk tolower _fstrdup _fstrrev strcat stricmp strrchr _tolower _fstricmp _fstrrchr strchr strlen strrev toupper _fstrlen _fstrset strcmp _strlwr strset _toupper _fstrlwr _fstrspn strcmpi strncat strspn vsprintf _fstrncat _fstrstr strcpy strncmp strstr
System calls	DOS &BIOS interrupt services	bdos _dos_creatnew _dos_open _harderr _bios_disk dosexterr _dos_read _hardresume _bios_equiplist _dos_findfirst _dos_setblock _hardretn _bios_keybrd _dos_findnext _dos_setdate inp _bios_memsize _dos_freemem _dos_setdrive inpw _bios_printer _dos_getdate _dos_setfileattr int86 _bios_serialcom _dos_getdiskfree _dos_setftime int86x _bios_timeofday _dos_getdrive _dos_settime intdos _chain_intr _dos_getfileattr _dos_setvect intdosx _disable _dos_getftime _dos_write outp _dos_allocmem _dos_gettime _enable outpw _dos_close _dos_getvect FP_OFF segread _dos_creat _dos_keep FP_SEG
Time Manipulate	system time	asctime difftime localtime strftime tzset clock ftime mktime _strtime utime ctime gmtime _strdate time
Variable-length args	Variable-length argument lists	va_arg vfprintf va_end vprintf va_start vsprintf

TABLE 2a: Operators by Category

This table lists operators by category.

Category	Symbol	Name or Meaning	Comments
	+	Addition	
	-	Subtraction	
	*	Multiplication	
	/	Division	
	%	Modulus	
Relational	<	Less than	
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
	==	Equal	
	!=	Not equal	
Assignment	=	Assignment	
	+=	Addition	
	-=	Subtraction	
	*=	Multiplication	
	/=	Division	
	%=	Modulus	
	<<=	Left shift	
	>>=	Right shift	
	&=	Bitwise AND	
	^=	Bitwise exclusive OR	
	|=	Bitwise OR	
Increment & Decrement	++	Increment	
	--	Decrement	
Bitwise	&	Bitwise AND	
	^	Bitwise exclusive OR	
	|	Bitwise OR	
	<<	Left shift	
	>>	Right shift	
	~	One's complement	
Relational	&&	Logical AND	
	||	Logical OR	
	!	Logical NOT	
Pointer	&	Address	
	*	Indirection	
	:>	Base	Example: myseg:>bp
Conditional	? :	Conditional	Example: (val >= 0) ? val : -val
Miscellaneous	( )	Function call	
	[ ]	Array element, structure or union member	
	->	Pointer to structure member	
	(type)	Type cast	
	sizeof	Size in bytes	

TABLE 2b: Operators by Precedence

The table below lists the C operators and their precedence and associativity values. The lines separate precedence levels. The highest precedence level is at the top of the table.

Symbol	Name or Meaning	Associativity
++	Post-increment	Left to right
--	Post-decrement	
( )	Function call	
[ ]	Array element	
->	Pointer to structure member	
.	Structure or union member	
:>	Base	
++	Pre-increment	Right to left
--	Pre-decrement	
!	Logical NOT	
~	Bitwise NOT	
-	Unary minus	
+	Unary plus	
&	Address	
*	Indirection	
sizeof	Size in bytes	
(type)	Type cast [for example, (float) i]	
*	Multiply	Left to right
/	Divide	
%	Remainder	
+	Add	Left to right
-	Subtract	
<<	Left shift	Left to right
>>	Right shift	
<	Less than	Left to right
<=	Less than or equal to	
>	Greater than	
>=	Greater than or equal to	
==	Equal	Left to right
!=	Not equal	
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	
|	Bitwise OR	
&&	Logical AND	
||	Logical OR	
? :	Conditional	Right to left
=	Assignment	Right to left
*=, /=, %=, +=, -=, <<=, &=, ^=, |=	Compound assignment	
,	Comma	Left to right

Table 3: 'printf( )' Type Specifiers and Escape Sequences

The format for the printf family of functions is as follows:

%<flags>< width . precision> <type prefix><format type>

Flags	-	left justify
	+	prefix with sign
	blank	prefix with blank
	#	modifies o,x,X, e,E,f,g,G
Type Prefix	F	far pointer
	N	near pointer
	h	short int
	l,L	long int or double
Format Type	d,i	signed decimal
	u	unsigned decimal integer
	o	unsigned octal integer
	x,X	unsigned hex integer
	f	fixed-point float
	e,E	scientific notation
	g,G	(%e or %f; whichever is shorter)
	c	single character
	s	string
	p	pointer
	n	character count

The C escape sequences are as follows:

Seq.	Name	Seq.	Name
\a	Alert (bell)	\v	Vertical tab
\b	Backspace	\'	Single quotation mark
\f	Form feed	\"	Double quotation mark
\n	Newline	\\	Backslash
\r	Carriage return	\ddd	ASCII character in octal notation
\t	Horizontal tab	\xdd	ASCII character in hex notation