intel®

C-86 COMPILER USER'S GUIDE

Order Number: 122085-001

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

AEDIT	iDIS	Intellink	MICROMAINFRAME
BITBUS	iLBX	iOSP	MULTIBUS
BXP	i_m	iPDS	MULTICHANNEL
COMMputer	iMMX	iRMX	MULTIMODULE
CREDIT	Insite	iSBC	Plug-A-Bubble
i	int$_e$l	iSBX	PROMPT
I^2ICE	int$_e$lBOS	iSDM	Ripplemode
iATC	Intelevision	iSXM	RMX/80
ICE	int$_e$ligent Identifier	Library Manager	RUPI
iCS	int$_e$ligent Programming	MCS	SYSTEM 2000
iDBP	Intellec	Megachassis	UPI

A948/ 583 / 1.5K DD

REV.	REVISION HISTORY	DATE
-001	Original issue.	6/83

PREFACE

Notational Conventions

italic	Italic indicates a meta symbol that may be replaced with an item that fulfills the rules for that symbol. The actual symbol may be any of the following:
filename	Is a valid name for the part of a *pathname* that names a file.
[ ]	Brackets indicate optional arguments or parameters.
{ }...	At least one of the enclosed items must be selected unless the field is also surrounded by brackets, in which case it is optional. The items may be used in any order unless otherwise noted.
[,...]	The preceding item may be repeated, but each repetition must be separated by a comma.
punctuation	Punctuation other than ellipses, braces, and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered: `SUBMIT PLM86(PROGA,SRC,'9 SEPT 81')`
`input lines`	In interactive examples, user input lines are printed in white on black to differentiate them from system output.
`<cr>`	Indicates a carriage return.

CONTENTS

CHAPTER 1 INTRODUCTION

This manual is a user's guide for the C compiler and runtime system for the Intel 8086, 8087, and 8088 microprocessors. This compiler may be used as a cross compiler running under RSX-11M compatibility mode VAX/VMS, as a native compiler running under ISIS on the Intel series III microcomputer development system, or under iRMX86 on supported boards and systems.

CC86 compiles programs written in the C programming language, as described in the reference *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). The fundamental data types supported include char (8 bits), short (16), int (16), long (32), float (32), and double (64); the modifier unsigned may be applied to char, short, int, and long. Storage classes supported include auto, extern, register, static, and typedef. The modifier readonly may be applied to objects in extern and static classes to indicate that no value will be written to the object. Additional data types may be derived from the fundamental types by means of arrays, functions, pointers, structures, and unions.

Lines beginning with # are preprocessor directives. The CC86 preprocessor supports the directives #define, #else, #endif, #if, #ifdef, #ifndef, #include, #line, and #undef, as described in *The C Programming Language.*

CC86 supports several advanced features in addition to the full range of features described in The C Programming Language. The data type void is a special type that may not be used in expressions; typically, it is used in the definition of a function that returns no value, to prevent its use in a value context. The derived type enum specifies an enumerated data type. CC86 also supports structure assignment and allows functions to take structure arguments and to return structure values.

CC86 translates programs into either relocatable object files or assembly language source files. The generated relocatable object code may then be transferred to the Series III development system, where it can be linked with the standard C runtime support libraries (using LINK86) and, if necessary, con verted into an absolute module (using LOC86).

CC86 supports both the SMALL and LARGE models of segmentation. A SMALL model program can have up to 64K bytes of code and 64K bytes of data. All pointers occupy two bytes (16 bits). This model is recommended for programs that can satisfy these size requirements, because the two byte pointers permit the generation of very compact and efficient code.

The LARGE segmentation model is used by programs that require access to the full addressing capabilities of the 8086 and 8088 processors. In this model, each source file generates a distinct pair of code and data segments. A single source file can generate up to 64K bytes of code and 64K bytes of data. All pointers occupy four bytes (32 bits). The generated code in the LARGE model is not as compact or efficient as that in the SMALL model: the large pointers are more difficult to manipulate and the compiler must generate code to adjust the segmentation registers whenever it detects a reference to an object in an unknown segment.

CC86 does not support the MEDIUM or COMPACT models of segmentation.

The runtime system includes a full implementation of the standard I/O package, a large number of generally useful routines for manipulating strings and a complete set of routines for interfacing with the DQ$ entry points of Intel's Universal Development Interface (UDI) libraries.

There are two different versions of the runtime library: one is used by SMALL model programs and the other by LARGE model programs. The libraries are completely compatible; in fact, they are just two compilations (one SMALL, the other LARGE) of the same C source code.

The C language has been slightly changed to make programming easier in the 8086 and 8088 environment. There is no limit to the number of characters in an identifier, other than the 39-character limit imposed by LINK86 and LOC86. The dollar sign $ is accepted in identifiers in exactly the same fashion as it is in PLM86 (it is silently thrown away). This makes it possible for calls to the system interface library routines to look exactly like the corresponding PLM86 system calls.

CHAPTER 2
COMPILING A PROGRAM

2.1 Under RSX-11M Compatibility Mode of VAX/VMS

The CC86 (Intel C Compiler) command provides an MCR interface to the C compiler. This command is simply a driver task that runs the actual compiler phases using the RSX-11M SPAWN directives.

The general syntax of the CC86 command is

```
CC86 [switches] file [file ...]
```

The command syntax is modeled after the DCL command language on the VAX, RSTS/E and RT-11. Any switches present on the command line persist for the entire command.

/DEBUG	The /DEBUG switch causes CC86 to place debugging information (local symbol and line number records) into the object module. These symbols are of use to DEBUG86 and ICE86.
/KEEP	The /KEEP switch suppresses the deletion of the temporary files used to pass in formation between the compiler phases. This option is used to retain a temporary file for examination, should one of the compiler's phases malfunction.
/LARGE	The /LARGE switch causes CC86 to generate code that uses the LARGE model segmentation assumptions.
/ASM86	The /ASM86 switch causes CC86 to generate assembly language. The output file has a file type of .A86 (instead of .OBJ). The code is formatted in a style that is easily understandable by users of ASM86; however, it is not an acceptable ASM86 program.
/VERBOSE	The /VERBOSE switch causes the CC86 command to print a running trace of the compiler phases as they are executed. This is useful if one wishes to see the exact command line that is passed to a phase (most often the preprocessor) or to provide a step-by-step trace of progress through a very large compilation on a very slow system.
/INCLUDE:*name*	The directory name is added to the list of directories searched by the preprocessor to locate #include files. The extra directories are searched in the order specified on the command line, before the default directory.
/DEFINE:*name* [:*value*]	The name *name* is defined to have value *value* just as if a #define line appeared in the source program. If the value parameter is omitted, the name is defined to have value 1 (so it can be used as a flag in a #if preprocessor line).

/UNDEFINE:*name*	The specified name is undefined, just as if a #undef preprocessor directive appeared in the source program. This is most often used to remove one of the preprocessor's built-in definitions.

2.2 Under RUN on the Series III Development System

The C compiler is a native mode program for the resident 8086 processor in the Series III development system. The Series III must be in 8086 execution mode when the compiler is invoked. Details on how to place the Series III into 8086 execution mode and how to execute commands in this mode may be found in the *Series III Console Operating Instructions.*

The general syntax of the invocation line is:

```
[:Fn:]CC86[.86] inputfile [TO outputfile] [controls]
```

The inputfile is a standard ASCII file containing a C source program prepared with one of the standard text editors (EDIT or CREDIT).

Normally, the output of the compiler, be it object code or an assembly language program, is written to a file on the same device and with the same name as the source file, but with the file type changed to .OBJ or .A86 (using a standard DQ$CHANGE$EXTENSION call). The TO outputfile control may be used to place the output in any desired file.

CC86 uses two temporary files. These are allocated on the :WORK: device. The WORKFILES control (described below) may be used to place the temporary files in specific places.

The controls may be one or more of the following compiler control arguments:

DEBUG	The DEBUG control causes CC86 to place debugging information (local symbol and line number records) into the object module. These symbols are of use to DEBUG86 and ICE86.
KEEP	The KEEP control suppresses the deletion of the temporary files used to pass in formation between the compiler phases. This option is used to retain a temporary file for examination should one of the compiler's phases malfunction. CC86 uses the :WORK: device for temporary files by default, so an explicit WORKFILES control (described below) must be used in conjunction with the KEEP control.
LARGE	The LARGE control causes CC86 to generate code that uses the LARGE model segmentation assumptions.
ASM86	The ASM86 control causes CC86 to generate assembly language. The output file has a file type of .A86 (instead of .OBJ). The code is formatted in a style that is easily understandable by users of ASM86; however, it is not an acceptable ASM86 program.

VERBOSE	The VERBOSE control causes the CC86 command to print a running trace of the compiler phases as they are executed. This may be used to get a step-by-step trace of progress through a very large compilation on a very slow system.
INCLUDE(*name*)	The INCLUDE control directs the preprocessor in its search for #include files. Because the UDI specification does not allow programs to know the syntax of file names, the search rules are slightly different from those specified by the language. In the source file, #include requests are of two types: #include "*file*" and #include < *file* >. In both cases, there may be additional names supplied by INCLUDE directives: INCLUDE (*name*1), ..., INCLUDE (*name*N). In either case, if no additional names have been supplied, the preprocessor will attempt to open file. If names have been supplied by INCLUDE controls, the treatments differ as follows. In the case #include "*file*", the preprocessor will first attempt to open file. If it fails, it then prefixes the names specified in the INCLUDE directives to the name file and attempts, in sequence, to open the files *name*1file, ..., *name*Nfile. In the case #include < *file* >, the preprocessor will first attempt to open the files *name*1file, ..., *name*Nfile, and will attempt to open file last.
DEFINE(*name* [,*value*])	The name is defined to have the given value, just as if a #define line appeared in the source program. If the value parameter is omitted, the name is defined to have value 1 (so it can be used as a flag in a #if preprocessor line).
UNDEFINE(*name*)	The specified name is undefined, just as if a #undef preprocessor directive appeared in the source program. This is only used to remove one of the preprocessor's built-in definitions.
WORKFILES(*file1*[,*file2*])	The WORKFILES control is used to specify the names of the two temporary files used by CC86. If only one name is specified, that file name is used for the first temporary file and the :WORK: device issed for the second temporary file.
ROM	The ROM control specifies that the compiled object will be read only. The impact of this control on the segmentation of the generated code is described in Chapter 5, "Runtime Issues."

2.3 Under iRMX™ 86

The invocation and control instructions for CC86 under iRMX86 are the same as under RUN on Series III.

NOTE

CC86 requires version 5.0 or later of iRMX86.

CHAPTER 3 LINKING C PROGRAMS ON THE SERIES III

The C compiler distribution kit includes two C runtime libraries (SCLIB.LIB for the SMALL segmentation model and LCLIB.LIB for the LARGE segmentation model). It also includes two runtime startoff routines (SQMAIN.OBJ for the SMALL model and LQMAIN.OBJ for the LARGE model). The object modules produced by the C compiler must be linked with the appropriate runtime star toff routine, the appropriate C library and the appropriate Intel interface library (SMALL.LIB or LARGE.LIB).

A typical command sequence for linking a SMALL model program called SMALLP with the standard libraries would be:

8087.LIB

```
>LINK86 SMALLP.OBJ,SQMAIN.OBJ,&
>SCLIB.LIB,SMALL.LIB,87NULL.LIB &
>BIND MAP SEGSIZE (STACK(800H),MEMORY(3000))
```

There are a number of things to note. First, note that a load time locatable image (LTL) is being created (the BIND control is used). This is necessary in the SMALL model for the dynamic memory allocation routines, which are used by the standard I/O library to function correctly. The size of the MEMORY segment is determined by a call to DQ$GET$SIZE, which does not return a reasonable value on absolute programs in the SMALL model. The amount of raw memory available for the dynamic allocation routines is determined by the SEGSIZE control that adjusts the size of the MEMORY segment. If there is insufficient free space available, it will not be possible to open files or perform other operations that require space in the dynamic storage allocation pool.

The stack used by the program is normally defined by the SYSTEMSTACK module in the SMALL library. The size of the stack can be determined from the link map. If more stack is required by an application, a SEGSIZE control may be used to increase the size of the STACK segment.

The following rough guidelines may be of use in estimating stack requirements. In the LARGE model, the overhead is 14 bytes per function call, plus 2 bytes per int and char, 4 bytes per long, float and pointer, and 8 bytes per double. In the SMALL model, the overhead is 8 bytes per function call, plus 2 bytes per int, char, and pointer, 4 bytes per long and float, and 8 bytes per double. Additional stack space will be needed for recursive function calls (the same amount of space is required for each level of recursion) and the local buffers used by UDI system calls.

The link command for a LARGE model program is similar. Here is an example:

```
-RUN LINK86 LARGEP.OBJ, &
>  LQMAIN.OBJ,&
>  LCLIB.LIB,&
>  LARGE.LIB &
>  LARGE.LIB,87NULL.LIB &
```

Once again, the BIND control has been used to create a load time locatable module. This speeds program development because a LOCATE step is not required; however, the actual loading of the program (by RUN) is quite slow because all of the absolute segment bases in the image must be fixed up. Absolute programs do, however, work properly in the LARGE model, since system calls (DQ$ALLOCATE and DQ$FREE) are used to perform all dynamic memory allocation.

The SEGSIZE control that adjusts the size of the STACK segment is almost always required. The default stack provided by the SYSTEMSTACK module in LARGE.LIB is seldom large enough for the (substantially) larger stack frames in the LARGE model.

Of course, the more elaborate features of LINK86 and LOC86 all work with the object modules produced by the C compiler. Detailed descriptions of the fancy features (such as building libraries, creating overlaid programs or writing code that is scattered all over physical memory) are beyond the scope of this manual. The Intel publications provide numerous examples of the complex features of LINK86 and LOC86.

3.1 Floating Point

Several special considerations apply to programs using floating point. The link command must include the appropriate floating point library, either 8087.LIB for hardware floating point with the 8087 or E8087.LIB for software emulation. The floating point library should be included after the C library (either SCLIB.LIB or LCLIB.LIB) in the LINK86 command.

NOTE

If your program does not use floating point, include the library 87NULL.LIB instead to avoid loading the emulator.

With either hardware or software floating point, the runtime startoff routine issues a call to INIT87 for initialization. A control word of 3BFH is loaded to mask exceptions and select round to nearest mode. Code generated by C routines may change the 8087 rounding mode, but the mode will always be restored.

Operation of the 8086 and 8087 processors is unsychronized. All C code manipulating floating point objects directly will check for termination of previous 8087 operations correctly. However, any C code that manipulates floating point objects in non-floating point contexts (for example, by casting a float * to char * and accessing the object through the char *) must explicitly check for termination of 8087 operation to assure correct results.

The code to output floating point numbers is quite bulky. Since most C programs do not need floating point output, the conversion routine in the standard library is a fake, which always prints the string {Float}". The real floating point output conversion routines are contained in the files SDTEFG.OBJ (SMALL model) and LDTEFG.OBJ (LARGE model). The appropriate object file should be included in the LINK86 command line immediately before the standard library.

Some users will always want floating point output conversion. These users may wish to delete the fake floating point output module (FDTEFG) from the standard library and replace it with the real version.

CHAPTER 4
USING THE STANDARD LIBRARIES

The standard C runtime libraries provide a large number of useful routines that make it easy to manipulate some common data structures (such as strings), dynamically allocate memory, and perform I/O operations to files on all devices supported by the operating system.

The purpose of this section is to provide a quick overview of the features and facilities of the library. The library routines are all listed in later chapters (along with their calling sequences and the types of their arguments).

The routines in the standard library on all hosts are identical, so it is easy to write programs that can be transported from system to system without change.

4.1 Standard Definitions

There are a number of header (.h) files supplied with the libraries. These files, which are intended to be included (using the #include preprocessor directive) by applications programs, provide a number of useful definitions for using the routines in the standard libraries. The most important of these is stdio.h. This header file contains all of the definitions used by the I/O routines, some symbolic constants (the value of the NULL pointer, for example) and external definitions for the library routines that return non-integer objects.

4.2 Overall Structure of Programs

A C program consists of a set of functions, one and only one of which must be called main. This function is called from the runtime startoff routine (SQMAIN.OBJ or LQMAIN.OBJ) after all of the required initialization of the runtime environment has been performed.

Programs may terminate in two ways. The easiest is to simply have the main routine terminate, returning control to the runtime startoff code, which performs some cleanup operations and returns control to the operating system. In some situations (errors, perhaps), it may be necessary to terminate a program, and it may not be desired (or even possible) to return to the main routine. In these situations, the exit routine can be used. This routine performs the standard cleanup and returns control to the operating system.

There is also a second exit routine called _exit that quickly returns control to the operating system without performing any cleanup. It should be used only in disaster situations, since bypassing the cleanup will leave files open and buffers of write data in memory.

4.3 Strings

A very common data structure in C programs is the character string. The usual runtime representation for a string is an array of characters delimited by a 0 byte ('\0'). This is, in fact, the representation used by the C compiler when a program contains a string constant (like "I am a string constant"). The address of the first character in the string is used as the handle for the string. Note that an array of 20 characters will hold a string of 19 (not 20) nonnull characters, delimited by a 0 byte.

Often strings can be assigned simply by shuffling pointers. If, however, it is necessary to actually move the characters, the library routine strcpy can be used. This is a function of two arguments; the first (a pointer to a string) points to the destination array and the second (also a pointer to a string) points to the source array. All characters, up to and including the 0 byte, are copied and the first argument is returned.

```
extern char *strcpy();
char buf[20];
strcpy(buf, "hello");
```

The length of a string may be determined by using the library routine strlen. This function takes one argument, a pointer to a string, and returns the number of characters in the string, up to but not including the 0 byte.

The strcat library function performs simple string concatenation. It takes two strings as arguments and appends a copy of the second string to the end of the first string. The first string is assumed to have enough extra space at the end to hold the new characters. Strcat returns a pointer to the new result, 0 byte delimited.

A typical use of strcat would be the creation of file names. Here a specific file type must be appended to a name that changes at runtime. For example, the following program fragment puts the file name "mumble.c" in the buffer buf.

```
char buf[20];
extern char *strcpy(), *strcat();
strcpy(buf, "mumble");
strcat(buf, ".");
strcat(buf, "c");
```

Often strings must be compared. This must be done, for example, if a list of strings is being sorted. The strcmp library function performs string comparison. It takes two arguments (both pointers to strings) and returns an integer. This integer is less than 0 if the first string is less than the second string (using native machine character comparisons), equal to 0 if the two strings are equal and greater than 0 if the first string is greater than the second string.

Applications that deal with fixed length strings can use the strncat, strncpy, and strncmp routines. These routines perform the same functions as their variable length (without the n) counterparts; however, they all take an additional (third) argument that specifies the maximum length of the string.

4.4 Input/Output

The standard library provides routines that do I/O at a number of levels to all devices supported by the operating system. Facilities exist for byte by byte, word by word, string, block, and formatted transfers. All I/O modes may be freely intermixed.

4.4.1 The FILE type

Included in the standard I/O header file stdio.h is a type definition (typedef) for the FILE type. A FILE is a structure that contains all of the information needed by the I/O routines to perform I/O operations on a connection. A pointer to a FILE is the external name of an I/O stream (much like the file variables of PASCAL or the unit numbers of FORTRAN) and is passed to the various routines in the I/O library to specify which stream participates in the transfer.

4.4.2 Opening (Creating) a FILE

A file is opened (and a FILE allocated) by the routines fopen and freopen. The most frequently used open routine is fopen. It takes two arguments. The first is a string that contains the name of the file to be opened. The second is a mode string that specifies the access mode required. The mode string is one of r (for plain reading), w (for plain writing), r+w (read and write, or update) or a (append). In addition, the mode string can contain the character b (for binary), which specifies that this is a binary (as opposed to an ASCII) stream, and specifies that newline characters should not be mapped into a carriage return/line feed sequence.

If the mode is w or a and the named file does not exist, it will be created. If the mode is w and the file does exist, it will be truncated to zero length.

If the open is successful, fopen will return a pointer to a FILE object. If unsuccessful, it will return NULL.

When control is passed to the main routine, the runtime startoff has already created three FILE objects. The first, called stdin (for standard input), is always attached to the :CI: device. The second, called stdout (for standard output), is always attached to the :CO: device. The third, called stderr (for standard error), is always attached to the video display device :VO:. A write to the standard error stream will always be seen, no matter how the :CO: stream is redirected.

The three names stdin, stdout, and stderr are defined as macros in the stdio.h header file. They cannot appear on the left-hand side of an assignment.

The alternate open routine freopen is just like fopen except that it takes a third argument. This argument is a pointer to a FILE that is closed and reopened, using the file name and access mode specified in the freopen call. It is most often used to redirect one of the standard streams to another file.

4.4.3 Closing a FILE

When all processing on a FILE is completed the stream must be closed by calling fclose. This routine takes one argument, a pointer to a FILE. Any data buffered in the stream is flushed; any buffers are released and the connection is detached.

All open files are automatically closed when a program terminates (via internal calls to fclose in exit).

4.4.4 Byte by Byte I/O

The lowest level of I/O is the byte-by-byte level. At this level, a call to the I/O routine either reads a single character from a FILE or writes a single character to a FILE.

All higher level I/O routines use these byte-by-byte routines to actually read and write data.

The most basic read routine is getc(fp). This function takes a single argument, a pointer to a FILE, and returns either the next character from the FILE or EOF. The definition of EOF is in the stdio.h header file. In ASCII mode, all carriage return characters (0DH) are thrown away, and the line feeds at the end of the lines (0AH) mark the end of the lines (the \'n' in C is equal to 0AH). In binary mode, all characters are passed without interpretation.

There is also a routine getchar that is equivalent to getc(stdin); it reads characters from the standard input FILE, which is normally the keyboard.

The routine ungetc(c, fp) returns c to the FILE fp. This is useful for looking ahead at the next input character and then returning it to the input file. Only a single character can be unread with ungetc.

The most basic write routine is putc(c, fp). The function takes two arguments: c is an integer containing the byte to be written and fp is a pointer to an output FILE. The first argument is returned unless there is some kind of write error, in which case EOF is returned.

There is also a routine putchar(c) that is equivalent to putc(c, stdout). It writes characters to the standard output FILE, which is normally the video display.

Here is a simple example that uses the I/O routines discussed so far. It copies the characters in the file "mumble.c" to the display. It prints a rude diagnostic if the file cannot be opened.

```
#include <stdio.h>

main()
{
        FILE *fp;
        int c;

        fp = fopen("mumble.c", "r");
        if (fp == NULL) {
                putchar('N');
                putchar('o');
                putchar('!');
                putchar('\n');
                exit(1);
        }
        while ((c=getc(fp)) != EOF)
                putchar(c);
        fclose(fp);
}
```

4.4.5 Word by Word I/O

A program may read the next word (16-bit object, low byte first) from a FILE by using the routine getw(fp). This routine takes one argument, a pointer to a FILE. The word read is returned.

Note that all bit patterns are legal return values for getw. A special token like EOF cannot be returned on end of file.

Instead, the program must explicitly test for end of file by using the macro feof(fp) (from stdio.h). This macro looks at the FILE pointed to by fp and returns true if the last call to getw ran into end of file. If a file has an odd size, the last call to getw will return the data and an error will be posted to the FILE. This error may be detected by using the ferror(fp) macro. End of file is posted only if a call to getw gets no data.

Of course, there is a similar routine putw(w, fp) that writes a word to a file. The ferror macro must be used to check for I/O errors.

4.4.6 String I/O

There are a number of routines that perform I/O on strings.

The most basic string read routine is fgets(b, n, fp). This routine reads a newline delimited string from the FILE pointed to by fp and stores it into the array of characters b. The newline character is transferred to the buffer. A 0 byte is placed in the buffer immediately after the newline. The integer n specifies the length of the buffer; this prevents fgets from writing beyond the array if a long line is encountered in the input.

Thére is also a routine gets(b). This routine reads a newline delimited string from the standard input stream and stores it into the array of characters b. The newline is deleted (this is different from fgets). A 0 byte is placed in the buffer immediately after the last byte read from the FILE.

The most basic string output routine is fputs(s, fp). This routine writes out the string s to the FILE pointed to by fp. There is also a routine puts(s) that writes the string pointed to by s, followed by a newline, to the standard output.

Here is another example. This program reads a filename from the keyboard, opens the file and copies it, line by line, to the video display.

```
#include <stdio.h>

char b[128];
char f[128];

main()
{
        FILE *fp;
        char *p;
        int c;
        puts("Enter a file name");
        gets(f);
        if ((fp=fopen(f, "r")) == NULL) {
                puts("Go away");
                exit(1);
        }
        while (fgets(b, sizeof(b), fp) != NULL) {
                p = b;
                do {
                        c = *p++;
                        putchar(c);
                } while (c != 'n');
        }
}
```

4.4.7 Block I/O

The standard library provides facilities for transferring blocks of memory to and from user programs. These are most often used on binary streams to move raw binary information to and from files. However, they may be used on ASCII streams with no ill effects, with the possible exception of newline interpretation.

The function fread(b, size, nitems, fp) reads nitems objects of size size bytes into the buffer pointed to by b from the FILE pointed to by fp. The number of items actually read is returned.

The analogous routine fwrite(b, size, nitems, fp) writes nitems objects, each of size size byte from the buffer b to the FILE pointed to by fp. The number of items actually written is returned.

The feof and ferror macros can be used to check for end of file and transmission errors on block reads and writes.

4.4.8 Formatted I/O

Routines are provided that permit formatted I/O to and from FILE streams. Data may be read in and written out in a number of formats and bases (decimal, octal, hexadecimal), strings may be truncated or padded, and fields may be justified to the left or to the right.

Although these routines are usually used on ASCII streams, they work perfectly well on binary streams (they are, after all, just interfaces to putc and getc). This facility is sometimes useful when dealing with strange command sequences that get sent to terminals, which often are mixtures of ASCII characters and binary values.

The formatted I/O routines printf and scanf are complex (to say the least!). The details of all their formatting options are described in detail in section 7.8.

Briefly, all formatted I/O routines work by interpreting one of their arguments as a format string. This string consists of format specifications (introduced by a '%' character) and ordinary characters (everything else). For each format specification encountered in the format string, an argument is extracted from the parameter list of the formatted I/O routine and interpreted in a fashion determined by the format specification. The type of the argument must agree with that expected by the format specification. If this is not the case (for example, a long integer is placed in the argument list where a normal integer is expected), the result is undefined.

The most commonly used format directives are %d (for decimal numbers), %o (for octal numbers), %x (for hexadecimal numbers) and %s (for strings).

Here is an example that uses the basic formatted output routine printf. This program prints out the numbers from 0 to 100 in decimal, octal, and hexadecimal.

```
#include < stdio.h >

main()
{
        int i;

        for (i=0; i < =100; + +i)
                printf("%d, %o, %x"n", i, i, i);
}
```

Note that the format string contains one format directive for each argument in the list, and that it also contains some literal characters that get copied directly into the output.

4.4.9 Random Access

All of the examples seen so far deal with sequential access FILE streams. However, the I/O library supports random access transfers as well. Associated with every FILE is a seek pointer. This pointer starts off at the beginning of the file (except, of course, when a stream is opened for append, where it starts off at the end of the file) and moves along as data is read from or written to the FILE.

The value of this pointer (as a 32-bit long integer) can be obtained with the routine ftell(fp). This routine returns the current value of the seek pointer for the FILE pointed to by fp.

The seek pointer can be moved about in the file by using the routine fseek(fp, where, how). This resets the seek pointer in the FILE pointed to by fp to where (also a 32-bit long integer). The how argument specifies if the seek is front of file relative (how = 0), current seek position relative (how = 1) or end of file relative (how = 2). Fseek has no defined return value.

Of course, some FILE streams (like the standard output, which is attached to the video display) cannot perform random access operations. If ftell is pointed at one of these streams, it returns garbage.

The special case of returning the seek pointer to the start of a file is made a little easier by the routine rewind(fp). This routine is equivalent to fseek(fp, 0L, 0).

Another example. This program opens a file on the disk and then lets the user display eight-byte fixed length records, by number.

```
#include <stdio.h>

char rec[8];

main()
{
        FILE *fp;
        int rn;
        char b[20];

        fp = fopen("database", "rb");
        if (fp == NULL) {
                puts("No database");
                exit(1);
        }
        while (gets(b) != NULL) {
                rn = atoi(b);
                fseek(fp, (long)8*rn, 0);
                fread(rec, sizeof(char), 8, fp);
                printf("Record %d:", rn);
                print();
        }
        exit(0);
}

atoi(s)
char *s;
{
int c, n;

n = 0;
while ((c=*s++) != 0)
        n = 10*n + c - '0';
return (n);
}

char hex[]      = {
        '0', '1', '2', '3', '4', '5', '6', '7',
        '8', '9', 'A', 'B', 'C', 'D', 'E', 'F'
};
```

```
print()
{
        int i;
        int byte;

        for (i=0; i < 8; ++i) {
                if (i != 0)
                        putchar(' ');
                byte = rec[i];
                putchar(hex[(byte >> 4)&0x0F]);
                putchar(hex[byte&0x0F]);
        }
putchar('\n');
}
```

4.5 Sorting

Often it is necessary to sort data. However, good sorting routines are tricky and difficult to debug, so the standard library contains two sort functions. These functions are general in that they implement only the skeleton of the sort algorithm. The user must provide a comparison function and tell the sort function the size of the objects being sorted.

The qsort(b, n, size, f) routine implements Hoare's quicksort. The argument b points to the base of the block of data being sorted. The n argument specifies number of elements to be sorted. Each of these objects has size size (the routine needs the size to be able to move the objects around and to update its internal pointers). The f is a pointer to a function that performs comparisons. It is called with two arguments (pointers to objects being compared) and it returns an integer that is less than 0, equal to 0 or greater than 0 to indicate the ordering.

The shellsort(b, n, size, f) routine has exactly the same calling sequence but uses Shell's sorting method. For most purposes, qsort is preferable.

The quicksort routine is recursive; it also uses a somewhat surprising amount of stack if presented with data that is almost sorted. The SEGSIZE control can be used on the LINK86 command line to allocate enough stack.

Here is an example. The quicksort routine is used to sort an array of integers.

```
#define   NINTS     20

int   ints[NINTS];

main()
{
        int compare();

        qsort((char *)ints, NINTS, sizeof(int), &compare);
}

compare(p1, p2)
char *p1, *p2;
{
        return (*(int *)p1 - *(int *)p2);
}
```

4.6 Dynamic Memory Allocation

When building linked data structures or when dealing with arrays whose size can only be determined at runtime, it is very useful to be able to allocate blocks of memory dynamically. The standard functions malloc, calloc, and free implement a general purpose memory allocation system. This system is used by user programs and by the I/O library routines to allocate buffers.

The basic allocator is malloc(n). This routine allocates a block of memory of at least n bytes and returns a (character) pointer to it. The block may be larger than requested, if allocating the exact size would create a very small (and probably unusable) block on the list of free memory. The block contains garbage; it is not initialized in any way. If there is no memory left in the free space pool, a NULL pointer will be returned.

The function calloc(n, size) allocates (with malloc) a block of memory large enough to hold n objects of size size; this memory is zeroed. If there is insufficient free memory, a NULL pointer is returned.

Blocks of memory that are no longer needed may be returned to the free pool by passing a pointer to the block to free(p). This routine puts the block back in the free list and merges adjacent free areas into single, larger free areas. It is a grave error to pass a nonsense pointer to free. No checking is done; a subsequent call to one of the allocation functions will probably return a very strange value.

4.7 The System Interface

The standard library provides a complete set of routines for dealing with the system. These permit the renaming and deletion of files, the catching of exceptions (including the control C key) and other low level system operations.

All of these routines, along with their calling sequences, are described in Section 8 below. More detail may be found in the Intel *Series III System Programmer's Reference Manual.*

4.8 Odds and Ends

There are some other routines in the library that perform conversions between character strings and binary values, generate random numbers and perform other commonly required actions. The Chapter 7 reference pages describe all of these routines.

CHAPTER 5
RUNTIME ISSUES

This section of the manual is intended to assist those users who must interface code generated by CC86 with code generated by other Intel translators such as PL/M-86 or ASM86. It describes, in detail, the calling sequence used by C functions, the conventions regarding the use of machine registers, how the segment registers are set up and other low level issues. This section may be ignored by most users.

The runtime environment used by the SMALL model of segmentation is quite different from that used by the LARGE model. Mixtures of the two models may work (and, in fact, be necessary) in some circumstances. However, mixing models should not be attempted by anyone short of a very experienced user.

5.1 Small Model

5.1.1 Segment Names and Attributes

In the SMALL model, a program has two segments, each 64K bytes (maximum) in size. One of these, mapped by the CS segment register and spanned by the group CGROUP, contains all of the machine code generated by CC86. The second, mapped by the DS, ES, and SS segment registers (which must contain the same value at all times) and spanned by the group DGROUP, contains all the pure and impure data, the stack, and the pool of free memory (the MEMORY segment) used by the dynamic storage allocation functions malloc and free.

CC86 places all instructions in a segment called CODE, which has a class name of CODE and is a member of the CGROUP. All pure data and readonly data is placed in a segment called CONST that has a class name of CONST. If the ROM control is specified, strings are also placed in CONST. All impure data (including strings, unless the ROM control is specified) is placed in a segment called DATA that has a class name of DATA. The CODE, CONST, and DATA segments, along with the machine stack (in a segment called STACK) and the free memory pool (in a segment called MEMORY), are members of the DGROUP.

Users of PL/M-86 will recognize these names as those used by the PL/M-86 compiler in the SMALL model. CC86 is completely compatible with PL/M-86 in terms of segment names, class names, groups, and attributes. The rules in the ASM86 manual that describe how to set up the segments for assembly language subroutines for PL/M-86 also apply to C.

5.1.2 Calling Sequence

Although call compatibility with PL/M-86 was an early design goal of the C implementation, this goal was not achieved. The C calling sequence is different from that used by PL/M-86 (and other Intel translators) for a number of reasons. First and foremost, the C language does not require that the number of arguments passed to a function be the same as the number of arguments specified in the function's declaration. Routines with a variable number of arguments are not uncommon. The two formatted I/O routines in the standard library (printf and scanf) are, in fact, routines that take a variable number of arguments. Given this requirement, the PL/M-86 convention of having the called routine remove the arguments from the stack is not usable. Further, the standard PL/M-86 calling sequence pushes the arguments from left to right, making it difficult to locate the first argument if the number of arguments is unknown. A secondary consideration in the design of the C calling sequence was the availability of variables of the register storage class.

Therefore, the following calling sequence is used. The function arguments are pushed, right to left, onto the stack. Long integers are pushed high half first; this makes the word order compatible with the DD assembly directive in ASM86. Doubles are pushed so that the byte order on the stack is compatible with the 8087. The function is then called with a NEAR CALL instruction (either directly or indirectly). An ADD instruction after the call removes the arguments from the stack.

For example, the function call

```
int     a;
long    b;
char    c;

f(a, b, c);
```

generates the code

```
movb    al,c
cbw
push    ax
push    b+2
push    b
push    a
call    f_
add     sp,8
```

Note that an underbar character ('_') has been appended to the function name. This serves two functions. First, it makes it harder to call a PL/M-86 routine by accident. Secondly, it means that there can be two routines, both with (apparently) the same name, that are called from C and PL/M-86 in (apparently) identical fashions. This facility is used in the UDI support library, where the DQ functions in the C library (whose names end in an underbar) and are simply interfaces to the routines in the Intel library that reverse the argument list and (sometimes) convert null terminated C strings to leading count PL/M-86 strings.

The parameters and local variables in the called function are referenced as offsets from the BP; the arguments begin at offset 8 and continue toward higer addresses, whereas the locals begin at offset −2 and continue toward lower addresses. The SP points at the local variable with the lowest address.

Functions return integers in the AX register, long integers in the DX:AX register pair, pointers in the AX register, and doubles on the top of the stack of the 8087.

```
f(a, b, c)
int a;
int b;
int c;
{
        return (a*b - c);
}

f_      proc   near

        push   si
        push   di
        push    bp
        mov     bp,sp

        sub     sp,N_autos

        mov     ax,a
        imul    b
        sub     ax,c

        mov     sp,bp
        pop     bp
        pop     di
        pop     si
        ret

f_      endp

a       equ     word ptr [bp+8]
b       equ     word ptr [bp+10]
c       equ     word ptr [bp+12]
```

5.1.3 Stack Allocation

The stack used by C programs is provided by the SYSTEMSTACK module in SMALL.LIB. The C runtime startoff routine contains a zero length stack segment with a symbol at the end of it, which gets relocated to the very top of the stack segment by LINK86. The stack may be set to any size by using the SEGSIZE directive in LINK86.

5.1.4 Segment Register Initialization

The runtime startoff routine always initializes the segment registers DS, ES and SS. It also sets up SP. Interrupts are disabled before touching any of these registers and are unconditionally enabled when the initialization is completed. The same startup routine will handle both absolute images and LTL images.

5.1.5 Command Line Processing

The command line that invoked the C program is obtained by calling DQ$GET$ARGUMENT until an argument string delimited by a carriage return is encountered. These command arguments are collected in a static buffer in SQMAIN. When control is passed to the user's main routine, three arguments are passed to it. The first, argc, is the number of arguments. The second, argv, is a pointer to an array of character pointers that point to the beginnings of the command strings in SQMAIN's buffer. This argument array is also statically allocated in SQMAIN. The third argument, envp, is always 0.

5.1.6 Heap Allocation

The MEMORY segment provides the raw material for the dynamic storage management functions. The size of the MEMORY segment is determined by subtracting the base address of the segment in the DGROUP (obtained by simply placing a label into the segment in SQMAIN) from the size of the data segment (obtained by a call to DQ$GET$SIZE). All of the MEMORY segment is linked into the free memory pool on the first call to malloc.

The size of the MEMORY segment may be adjusted by using the SEGSIZE control on LINK86.

Note that DQ$GET$SIZE does not return a useful result if the program has been bound as an absolute image (that is, has been processed by LOCATE86). Consequently, the storage allocator will malfunction if used by an absolute program. This means that absolute programs may not use the standard I/O package without providing their own versions of malloc and free, since the I/O routines use the dynamic space allocator to obtain and release I/O buffers.

5.1.7 Interfacing with Intel Supplied Routines

Most routines supplied by Intel use the PL/M-86 calling con ventions. As was previously mentioned, the code generated by the C compiler cannot, because of the semantics of C, use these conventions. If it is necessary to call such routines (for example, the interface routines of RMX86), the linkage must be written in ASM86.

Here is a simple example. Assume that it is necessary to call the PL/M-86 function USEFUL, which has the following declaration:

```
USEFUL:   PROCEDURE (A, B, C) EXTERNAL POINTER;
          DECLARE    (A, B, C) INTEGER;
          END;
```

The ASM86 linkage to this function would look like this:

```
          name      useful

cgroup    group     code
dgroup    group     const, data, stack, memory
          assume    cs:cgroup
          assume    ds:dgroup, es:dgroup, ss:dgroup

code      segment   public 'code'
          public    useful_              ; Note the "_"
          extrn     useful:near

useful_   proc      near
          push      si                   ; C save code
          push      di
          push      bp
          mov       sp, bp
          sub       sp, N_autos          ; Claim locals

          push      word ptr [bp+ 8]     ; Push parameters
          push      word ptr [bp+10]     ; from left
          push      word ptr [bp+12]     ; to right, and
          call      useful               ; call routine.
          mov       ax, bx               ; Move return value.

; At this point, the SI and DI registers
; may have been altered.

          mov       sp, bp               ; C return code
          pop       bp
          pop       di
          pop       si
          ret
```

```
useful_     endp

code        ends
            end
```

5.2 Large Model

5.2.1 Segment Names and Attributes

Object modules generated by CC86 in LARGE model always contain two segments. One of these holds all of the code produced by the functions in the file. The other generally contains all of the data actually allocated by the functions in the file. These segments are called 'name_CODE' and 'name_DATA', where name is the name of the source file (with all leading devices, UIC, and directory information stripped off). The code segment always has class name CODE, and the data segment always has class name DATA; these are the same naming conventions used by PL/M-86 in the LARGE model.

The 'name_CODE' segment includes code, linkage vectors, literals, switch tables, and readonly data. If the ROM control is specified, it also includes strings. The 'name_DATA' segment in cludes ordinary external and static impure data items. If the ROM control is not specified, it also includes strings.

There are no group definitions in the object code produced by the LARGE model CC86.

5.2.2 Calling Sequence

The LARGE model calling sequence is similar in spirit to the SMALL model sequence. Arguments are passed in exactly the same way, except that pointers are two-word objects. The base part of the pointer is pushed first, followed by the offset part. This makes the pointer object on the stack compatible with the standard 8086 pointer.

As a consequence of the fact that the return address pushed by the FAR CALL instruction is now a double word, the first argument is at offset 10 from the BP (rather than at offset 8 as in the SMALL model).

Functions return pointer objects in the DX:AX register pair. This is different from PL/M-86, which returns pointer objects in the ES:BX register pair.

The following example copies a character string from the location pointed to by p1 to that pointed to by p2, changing all upper case letters to the '!' character.

```
f(p1, p2)
char *p1, *p2;
{
        int c;

        while ((c=*p1++) != 0) {
            if (c >='A' && c <='Z')
                c = '!';
            *p2++ = c;
        }
        *p2 = 0;
}
```

```
f_      proc    far

        push    si
        push    di
        push    bp
        mov     sp,bp

        sub     sp,2

p1      equ     dword ptr [bp+10]
p2      equ     dword ptr [bp+14]
c       equ     word ptr [bp-2]

10:     les     si,p1
        inc     word ptr p1
        mov     al,es:[si]
        cbw
        mov     c,ax
        or      ax,ax
        je      12

        cmp     c,'A'
        jl      11
        cmp     c,'Z'
        jg      11
        mov     c,'!'

11:     les     si,p2
        inc     word ptr p2
        mov     ax,c
        movb    es:[si],al
        jmp     10

12:     les     si,p2
        mov     es:byte ptr [si], 0

        pop     bp
        pop     di
        pop     si
        ret

f_      endp
```

5.2.3 Runtime Startoff

The runtime startoff routine works exactly the same way in the LARGE model as it does in the SMALL model. Only the SS and the SP registers are set up (the DS and ES registers are set up to access internal data while the LQMAIN routine is running).

5.2.4 Heap Allocation

The standard allocation routines malloc and free are simply interfaces to the library functions DQ$ALLOCATE and DQ$FREE. LARGE model programs may be bound in any fashion in which these Intel supplied routines function correctly.

5.2.5 Interfacing with Intel Supplied Routines

The LARGE model interface to Intel supplied routines is similar to that used in the SMALL model. Because of differences between the C and PL/M-86 calling sequences, the linkage must be written in ASM86.

Here is an example. Assume a LARGE model interface is required for the same USEFUL PL/M-86 routine used as an example in the SMALL model. The following ASM86 routine will perform the linkage:

```
                name        useful

                extrn       useful:far
useful_code segment public 'code'
                assume      cs:useful__code
                public      useful_                 ; Note the "_"

useful_         proc        far
                push        si                      ; C save code
                push        di
                push        bp
                mov         sp, bp
                sub         sp, N_autos             ; Claim locals

                push        word ptr [bp+10]        ; Push parameters
                push        word ptr [bp+12]        ; from left
                push        word ptr [bp+14]        ; to right, and
                call        useful                  ; call routine.
                mov         dx, es                  ; Return pointer
                mov         ax, bx                  ; in dx:ax

; At this point, the SI, DI, DS and ES registers
; may have been altered.

                mov         sp, bp                  ; C return code
                pop         bp
                pop         di
                pop         si
                ret

useful_         endp

useful_         code ends

                end
```

CHAPTER 6
SHORTCOMINGS AND CAVEATS

This section is an attempt to document the known shortcomings in the compiler and its runtime system and to warn the new user of some of the more common difficulties.

6.1 Binary Files

The ISIS file structures maintain a distinction between ASCII and binary files. In a binary file, all characters are simply read and written as encountered. However, in an ASCII file, all newlines must be expanded to carriage return/line feed sequence on output, and the carriage return/line feed sequence must be converted to newline on input. The fopen routine takes an extra format specifier in the mode field (a b) to specify a binary stream; forgetting to specify the b will make extra 0Dh bytes appear in output files and will make 0Dh bytes disappear on input files. This class of problems happens most frequently when one is moving a program from COHERENT (where there is no distinction between ASCII and binary I/O) to ISIS.

6.2 Running out of Memory

Care should be taken when writing programs that allocate memory with the dynamic memory allocation functions malloc and free. Typically, a program simply prints a message and exits when it discovers that no more dynamic memory is available. However, I/O buffers are claimed on demand. If the error message is the very first write to a stream, there may not be enough space to claim the I/O buffer. To make programming a little easier, the standard error stream preallocates its buffers. However, programs that write diagnostics to the standard output or to some other stream should be cautious.

The **setbuf** routine may be used to force the allocation of the buffer.

6.3 Fields

The C language requires only that fields be implemented in integers. It also allows the implementation considerable liberty with respect to the zero or sign extension of fields.

CC86 allows fields of char, of unsigned char, of short, of unsigned short, of int, and of unsigned int. Fields in signed types are sign extended to integers when referenced. Fields in unsigned types are zero extended to integers when referenced.

No attempt has been made to implement fields in long integers or unsigned long integers.

CHAPTER 7
THE STANDARD (libc) LIBRARY

The standard libraries contain a large number of routines that perform many common programming tasks. This section describes each of the routines in the libraries. For each routine, it describes the calling sequence (the type of the return value and the types of each of the arguments) and gives a quick explanation of the routine's function.

The library routines are divided into functional groups. These groups correspond (roughly) to the topics in Chapter 4 of this manual.

7.1 Character Classification

The include file ctype.h contains definitions for a number of character classification macros. These macros permit the lexical class of a character to be easily determined.

The macros use a character classification table so that all class determinations are short and efficient.

The isascii macro is defined on all integers. All other macros are defined only on the special value EOF and legal ASCII characters (as determined by isascii).

isalnum(*c*); int *c*;

The **isalnum** macro tests if *c* is either an alphabetic character or a numeric character (as defined by the **isalpha** and **isdigit** macros).

isalpha(*c*); int *c*;

The **isalpha** macro tests if *c* is alphabetic. In this context, alphabetic means the upper and lower case letters and the underbar(_).

isascii(*c*); int *c*;

The **isascii** macro tests if the integer *c* is in the legal ASCII range (0 to 127 decimal). It is normally used to check the legality of a character before presenting it to one of the other macros, which malfunction on out of range arguments.

iscntrl(*c*); int *c*;

The **iscntrl** macro tests if *c* is a rubout (7FH) or a control character (less then 20H).

isdigit(*c*); int *c*;

The **isdigit** macro tests if *c* is a digit (between 0 and 9).

islower(*c*); int *c*;

The **islower** macro tests if *c* is a lower case letter (between a and z).

isprint(*c*); int *c*;

The **isprint** macro tests if *c* is a printing character (between a blank space and ~).

ispunct(*c*); int *c*;

The **ispunct** macro tests if *c* is a punctuation character. A punctuation character is defined as a character that is neither a control character nor an alphanumeric character.

isspace(*c*); int *c*;

The **isspace** macro tests if *c* is a whitespace character (space, tab, carriage return, newline, line feed or form feed).

isupper(*c*); int *c*;

The **isupper** macro tests if *c* is an upper case letter (A through Z).

7.2 String Manipulation

The string manipulation routines work on 0 byte terminated strings stored in arrays of characters. They all assume that their arguments are well formed. If any of the routines are called with ill-formed strings (strings without the 0-byte termination), they may test, compare or move all of memory!

char ***strcat**(*s1*, *s2*); char **s1*, **s2*;

The **strcat** routine concatenates a copy of the string pointed to by *s2* to the end of the string pointed to by *s1*. The destination string is assumed to have enough memory allocated past its end to hold the extra characters. The *s1* argument (a pointer to the result) is returned.

char ***strncat**(*s1*, *s2*, *n*); char **s1*, **s2*, *n*;

The **strncat** routine is just like **strcat**, except that it will never copy more than *n* characters from the second string.

int **strcmp**(*s1*, *s2*); char **s1*, **s2*;

The **strcmp** routine performs lexicographic string comparison. It takes pointers to two strings as arguments and returns an integer that is less than zero if the first string is less than the second string, equal to zero if the first string is the same as the second string or greater than zero if the first string is greater than the second string.

int **strncmp**(*s1*, *s2*, *n*); char **s1*, **s2*; int *n*;

The **strncmp** routine is just like **strcmp**, except that it does not compare more than *n* characters.

int **strlen**(*s1*); char **s1*;

The **strlen** routine returns the number of characters in the string pointed to by *s1*.

char ***strcpy**(*s1*, *s2*); char **s1*, **s2*;

The **strcpy** routine copies the string pointed to by *s2* into the string pointed to by *s1*. The *s1* argument (a pointer to the result string) is returned.

char ***strncpy**(*s1*, *s2*, *n*); char **s1*, **s2*; int *n*;

The **strncpy** routine is just like **strcpy**, except that no more than *n* characters are copied.

char ***index**(*s1*, *c*); char **s1*; int *c*;

The **index** routine returns a pointer to the first occurrence of the character *c* in the string *s1*. A NULL pointer is returned if the character is not present in the string.

char ***rindex**(*s1*, *c*); char **s1*; int *c*;

The **rindex** routine returns a pointer to the last occurrence of the character *c* in the string *s*1. A NULL pointer is returned if the character is not present in the string.

7.3 Creating, Deleting and Manipulating FILE Objects

FILE ***fopen**(*name*, *mode*); char **name*, **mode*;
FILE ***freopen**(*name*, *mode*, *fp*); char **name*, **mode*; FILE **fp*;

The **fopen** routine creates a new FILE object and attaches the device and/or file specified by the name argument to it.

The *name* argument is a string. Any device and/or file name, as defined by the operating system, is acceptable.

The *mode* string must be one of one of **r** (for reading), **w** (for writing), **r+w** (for updating) or **a** (for appending). If the file does not exist and the mode is **w** or **a**, it will be created. If the mode is **w** and the file does exist, it will be truncated to zero length (the old contents are destroyed).

The *mode* string may also contain the character **b** to specify that the new FILE should be set up for binary I/O. A binary FILE is the same as a default (ASCII) file, except that the special processing of the newline character (0AH) is disabled.

A pointer to the new FILE object is returned. A NULL pointer is returned on any kind of error.

The **freopen** routine is like **fopen** routine, except that it takes a third argument *fp*. This FILE object will be closed, and the named file will be attached to it. This routine is normally used to associate one of the standard streams (stdin, stdout or stderr) with a specific file.

int **fclose**(*fp*); FILE **fp*;

The **fclose** routine destroys the FILE object pointed to by *fp*, after finishing up any I/O operations associated with the FILE, releasing any I/O buffers and detaching the connection. It returns 0 if all went well, or −1 on any type of error.

int **fflush**(*fp*); FILE **fp*;

The **fflush** routine writes out any data that has been buffered in a FILE object. It returns 0 if all went well, and −1 on any kind of error. The **fflush** performs no operation on an input stream; it always returns a successful status.

void **setbuf**(*fp*, *b*); FILE **fp*; char *b*[BUFSIZ];

The **setbuf** routine causes the buffer *b* to be associated with the specified FILE. It must be called before buffers are dynamically allocated to the FILE (that is, before the first read or write operation is performed).

This routine is most often used to prevent I/O buffers from being allocated in the dynamic storage pool in programs that require very precise control of their memory usage.

feof(*fp*); FILE **fp*;

The **feof** macro tests the _FEOF flag in the FILE *fp*. This flag is set when an input FILE hits end of file.

ferror(*fp*); FILE **fp*;

The **ferror** macro tests the _FERR flag in the FILE *fp*. This flag is set on any kind of I/O error.

clearerr(*fp*); FILE **fp*;

The **clearerr** macro clears the _FERR flag in the FILE *fp*. It is used by programs that recover from I/O errors.

fileno(*fp*); FILE **fp*;

The **fileno** macro extracts the operating system's connection number from the FILE *fp*. It might be used, for example, to obtain the connection number so that it could be passed to **dq$special** or **dq$get$connection$status**.

7.4 Byte by Byte I/O

int **fgetc**(*fp*); FILE **fp*;

The **fgetc** routine reads and returns the next byte from the input FILE *fp*. The special value EOF (−1) is returned on end of file or error.

int **fputc**(*c*, *fp*); int *c*; FILE **fp*;

The **fputc** routine writes the byte *c* onto the output FILE *fp*. The *c* argument is returned, if all went well. An EOF is returned on any kind of error.

The **fgetc** and **fputc** routines are the actual, low level byte-by-byte I/O functions. However, they are normally not called by users. User programs call these routines through four standard macros.

getchar()

The **getchar**() macro is identical to **fgetc(stdin)**.

getc(*fp*)

The **getc**(*fp*) macro is identical to **fgetc**(*fp*).

putchar(*c*)

The **putchar**(*c*) macro is identical to **fputc**(*c*, *stdout*).

putc(*c*, *fp*)

The **putc**(*c*, *fp*) macro is identical to **fputc**(*c*, *fp*).

int **ungetc**(*c*, *fp*); int *c*; FILE **fp*;

The **ungetc** routine pushes the character *c* back into the input FILE fp. Only one character may be pushed back. This routine is useful in situations (such as the reading of numbers) where an extra character must be read in order to determine that the end of the input has been reached.

7.5 Word by Word I/O

int **getw**(*fp*); FILE **fp*;

The **getw** routine reads and returns the next (16-bit) word from the input FILE fp. It returns EOF on end of file. However, since **EOF** is a legal word value, the **feof** or **ferror** macros must be used to determine the success or failure of a **getw**.

int **putw**(*i*, *fp*); int *i*; FILE **fp*;

The **putw** routine writes the (16-bit) word *i* to the output FILE *fp*. It returns *i* if the write was successful, and **EOF** on any kind of error. Since **EOF** is a legal word, the **ferror** macro must be used to check the success of a **putw**.

7.6 String I/O

char ***fgets**(*b*, *n*, *fp*); char **b*; int *n*; FILE **fp*;

The **fgets** routine reads characters from the input FILE *fp* and stores them into the buffer *b*. It stops reading on end of file, when a newline character is read or after *n*-1 bytes have been stored in the buffer. Newlines are stored in the buffer. A 0 byte is stored in the buffer immediately after the last character read.

The *b* argument is returned unless reading was terminated by end of file, in which case NULL is returned.

char ***gets**(*b*); char **b*;

The **gets** routine is much like **fgets**, except that it always reads from the standard input FILE. There is no *n* parameter to specify the length of the buffer, and delimiting newlines are NOT stored in the buffer.

int ***fputs**(*b*, *fp*); char **b*; FILE **fp*;

The **fputs** routine writes the 0 byte terminated string in the buffer *b* onto the output FILE *fp*.

int ***puts**(*b*);

The **puts** routine writes the 0 byte terminated string in the buffer *b*, followed by a newline, to the standard output FILE.

7.7 Block I/O

int **fread**(*b*, *s*, *n*, *fp*); char **b*; int *s*, *n*; FILE **fp*;

The **fread** routine reads (up to) *n* objects, each of size *s* bytes, from the input FILE *fp* into the buffer *b*. The number of items actually read is returned.

The **feof** and **ferror** macros must be used to check for end of file or error conditions.

int **fwrite**(*b*, *s*, *n*, *fp*); char **b*; int *s*, *n*; FILE **fp*;

The **fwrite** routine writes *n* items, each of size *s* bytes, from the buffer *b* onto the output FILE *fp*. The number of items actually written is returned.

The **ferror** macro must be used to check for error conditions.

7.8 Formatted I/O

printf(*format* [, *list*]); char **format*;
fprintf(*fp*, *format* [, *list*]); FILE **fp*; char **format*;
sprintf(*sp*, *format* [, *list*]); char **sp*, **fp*;

These three routines perform formatted output conversion. The **printf** routine writes characters to the standard output FILE, the **fprintf** routine writes characters to the FILE *fp*, and the **sprintf** routine stores characters into the string *sp*.

The *format* argument is a character string that controls the interpretation of the additional arguments in the comma separated list. Ordinary characters (characters that are not part of a format specification) are simply copied to the output.

Format specifications are introduced by a percent sign (%). After the % there may be:

1. A minus sign (—) that specifies left adjustment of the data in the output field, instead of the default right adjustment.
2. A string of decimal digits that specify the width of the output field. Normally, a field is padded to its field width with space characters (blank spaces). However, if the first character of the field width is a 0, the field will be padded with 0 characters; the leading 0 does not cause the field width specification to be taken as an octal number. If the field width is an *, the next int from the *list* is used as the field width.
3. A period (.) that serves only to separate the two decimal digit strings.
4. A string of decimal digits that specifies the precision of an **e**, **f**, or **g** conversion item, or the maximum number of characters that will be output by an **s** conversion item. If the maximum number is an *, the next **int** from the *list* is used as the maximum width.
5. An l that specifies that the argument from the list is a **long** object rather than an **int** object. Making the conversion character uppercase has the same effect.
6. A conversion character that specifies the exact form of the data conversion. The legal conversion characters are:

%	The character % is output; the sequence %% is used to print a single % character.
c	The next int from the list is output as a character.
d(D)	The next **int (long)** from the *list* is output in decimal.
e	The next **float** or **double** from the *list* is output in the format [—]*d.ffffffE* [+—]*ee*, where the length of the fraction string *ffffff* is given by the precision (default 6).
f	The next **float** or **double** from the *list* is output in the format [—]*ddd.ffffff*, where the length of the fraction string *ffffff* is given by the precision (default 6).
g	The next **float** or **double** from the *list* is output in the shorter of either the **e** or the **f** conversion format.
o(O)	The next **int (long)** from the *list* is output in octal.
r	The next **char** * from the *list* is taken as a pointer to the argument list of a function. A recursive invocation of **printf**, **fprintf**, or **sprintf** is created to process this list as a **printf** argument list, with the pointer pointing at the *format* argument. This format item is used to implement functions that take **printf** style format lists as arguments.
s	The next item from the *list* is taken to be a (character) pointer to a string. This string is output, subject to the maximum length specification.
u(U)	The next **int (long)** from the *list* is output as an unsigned decimal integer.
x(X)	The next **int (long)** from the *list* is output in hexadecimal. The characters A through F (uppercase) are used for the digits with values 10 through 15.

Users requiring floating point output should read the remarks in section 3.1 above. Floating point output may print several strings in addition to the usual numbers. The string { Float } indicates that the real floating point output routine was not included in the link, as described above. The string { *s* Unnormal } , where *s* is + or —, indicates that the floating point object is unnormalized. The string { *s* NAN } indicates that the floating point object is not a legitimate floating point number. The string { *s* Infinity } indicates that the floating point object represents infinity or —infinity. The string { *s* Denormal } indicates that the floating point object is denormalized.

scanf(*format* [, *list*]); char **format*;
fscanf(*fp*, *format* [, *list*]); FILE **fp*; char **format*;
sscanf(*sp*, *format* [, *list*]); char **sp* **format*;

These three routines perform formatted input conversion. The **scanf** routine reads characters from the standard input FILE, interprets them according to the given format and stores the results in the argument *list*. **fscanf** reads from the FILE *fp*, and **sscanf** reads from string *sp*.

The *format* argument is a character string that controls the interpretation of the input. The *list* arguments must be pointers that indicate where the corresponding input item will be stored. White space characters (space, tab, newline) in format are ignored. Other characters except % match non-white space characters in the input. The % character identifies the start of a conversion specification. Each conversion may use one or more of the remaining *arg* arguments. It is essential for users to ensure type matching between the arguments and the conversion specifications.

Each routine terminates when it encounters the end of the *format* string or when the input does not match a specification. Each returns the number of successful assignments.

After the % character, there may be characters indicating the width of the input field and the conversion type. A field is delimited by white space (space, tab, newline) or by the given field width, if any. Newlines are white space, so the input can include more than one line. The following modifiers, in this order, may precede the conversion type:

1. An optional *, indicating that the next input field should be skipped (rather than assigned to the next variable in *list*).
2. An optional string of decimal digits, specifying a maximum field width.
3. An l, specifying that the next input item is a **long** object rather than an **int** object. Making the conversion character uppercase has the same effect.
4. A conversion character that specifies the exact form of the data conversion. The legal conversion characters are:

c	The next input character is assigned to the next list member, which should be **char** *.
d(D)	The next input field is a decimal (*long*) integer; the next *list* member should be **int** * (**long** *).
e	The next input field is a floating point number; the next *list* member should be **float** * or **double** *.
f	Same as e.
o(O)	The next input field is an octal (**long**) integer; the next *list* member should be **int** * (**long** *).
s	The next input field is a string; the next *list* member should be **char** *.

7.9 Random Access

Associated with every FILE is a long integer containing the *seek pointer*. This pointer is an origin 0 offset, in bytes, from the start of the file. It specifies the next byte that will be read or written and is advanced as I/O is actually performed. This seek pointer may be manipulated by programs to perform random access file operations.

int **fseek**(*fp*, *offset*, *how*); FILE **fp*; long *offset*; int *how*;

The **fseek** routine adjusts the seek pointer associated with the FILE *fp*. If *how* is 0, the seek pointer is set to *offset*. If *how* is 1, *offset* is added to the seek pointer (permitting relative seeking). If *how* is 2, the seek pointer is set to the sum of *offset* and the size of the file (in bytes). This permits seeking relative to the end of file.

long **ftell**(*fp*); FILE **fp*;

The **ftell** routine returns the seek pointer associated with the FILE *fp*.

FILE ***rewind**(*fp*); FILE **fp*;

The **rewind**(*fp*) routine is identical to **fseek**(*fp*, 0L, 0). It is provided only for programming convenience.

7.10 Sorting

The standard library provides two completely general sorting routines. These routines implement only the framework of the sort. The user program must provide a routine to perform key comparison.

void **shellsort**(*b*, *n*, *s*, *p*); char **b*; int *n*, *s*; int (**p*)();

The **shellsort** is a general purpose sorting function that uses Shell's sorting algorithm. The argument *b* is a pointer to the base of the data block to be sorted. The block contains *n* items, each of size *s* bytes. The *p* argument is a pointer to a function that takes two arguments (both pointers to the objects being compared) and returns an integer that is less than zero if the first object is less than the second, equal to zero if the objects are identical, and greater than zero if the first object is greater than the second object.

void **qsort**(*b*, *n*, *s*, *p*); char **b*; int *n*, *s*; int (**p*)();

The **qsort** routine is just like the **shellsort** routine, except that it uses C. A. R. Hoare's quicksort algorithm.

7.11 Dynamic Memory Allocation

The standard library provides a general purpose dynamic memory allocation system. This system is used both by user programs and by the I/O routines contained within the standard library to dynamically allocate and release blocks of memory.

char ***calloc**(*n*, *s*); unsigned int *n*, *s*;

The **calloc** routine allocates (via an internal call to **malloc**) enough memory to contain *n* objects each of size *s* bytes. It clears this memory to binary zeros and returns a pointer to it. It returns NULL if the memory cannot be allocated.

void **free**(*p*); char **p*;

The **free** routine takes a pointer *p* to a block of memory, which has been allocated by **malloc** or **calloc**, and returns the block to the free memory pool. Passing random pointers, or pointers to blocks of memory not allocated by **malloc** or **calloc** to **free**, brings speedy disaster.

char ***malloc**(*n*); unsigned int *n*;

The **malloc** routine allocates and returns a pointer to a block of memory at least *n* bytes in length. The memory is not cleared. It returns NULL if the memory cannot be allocated.

7.12 Odds and Ends

The standard library contains routines to convert numbers (stored in character strings) from ASCII to binary, to generate random numbers and to perform nonlocal flow control.

int **abs**(*i*); int *i*;

The **abs** routine computes the absolute value of its argument *i*. No overflow checking is performed; the absolute value of the largest negative number is itself.

double **atof**(*s*); char **s*;
int **atoi**(*s*); char **s*;
long **atol**(*s*); char **s*;

The **atof**, **atoi** and **atol** routines convert a number stored as an ASCII character string to a **double**, an **int**, or a **long** respectively. Leading whitespace is ignored. Leading signs ('+' and '−') are accepted and correctly interpreted. The first unrecognized character (usually the 0 byte at the end of the string) stops the conversion. No overflow checking is performed.

int **rand**();
void **srand**(*seed*); int *seed*;

The **rand** routine is a random number generator. Every time it is called, it returns a new random number in the range 0 to 2ˆ15–1. The generator has a period of 2ˆ32. The **srand** routine can be called to seed (reset) the random number generator. Often a timing device (Intel 8253, for example) can be used as a source of random seeds.

int **setjmp**(*env*); *jmp_buf env*;
void **longjmp**(*env*, *value*); *jmp_buf env*; int *value*;

The **setjmp** and **longjmp** routines manipulate machine environments and provide a simple scheme for performing nonlocal transfers of control. An environment (*env*) is an array of some sort. The include file **setjmp.h** contains a *typedef* (*jmp_buf*) for this object.

The **setjmp** routine saves the state of the runtime stack (SP, BP, and IP, plus the CS in the LARGE model) in the supplied environment and returns 0.

The **longjmp** routine restores the state of the runtime stack from the *env*, and then makes the call to setjmp that set up the environment return again. However, this time, the **setjmp** routine returns *value*.

The caller of **setjmp** must not have returned when **longjmp** is called, or the runtime stack will be destroyed.

CHAPTER 8
THE SYSTEM INTERFACE (DQ$) LIBRARY

Both C libraries contain a complete set of system interface (DQ$) routines. These routines have the same names as their PL/M-86 counterparts described in the *Series III System Programmer's Reference Manual*. In almost all cases, the calling sequences are identical.

The interface routines perform some minor transformations upon their parameters to make it easier to call the system from C programs. In particular, they transform the 0-byte terminated strings of C into the leading count strings of PL/M-86 by moving the data into a buffer on the stack.

The header file udi.h contains definitions and macros useful for dealing with the system interface. Included in this file are symbolic names for the system error codes, some structures for dealing with the time, date, and status of a connection, and definitions for the types (such as **token** and **Boolean**) used by the interface routines.

The following subsections contain brief descriptions of each routine. Experienced Series III programmers will find this information sufficient. Less experienced programmers are well advised to refer to the Intel publications for more elaborate descriptions.

8.1 Segment Management

token **dq$allocate**(*size*, *excep$p*);
unsigned int **size**; int **excep$p*;

This function allocates a new segment at least *size* bytes in length (with 0 meaning 64K) and returns a token representing the base of the new segment. If the operation fails, a token of 0xFFFF is returned. This routine is probably of very little use to programs running in the SMALL segmentation model, since the new segment may not be addressable. However, this routine is used (almost directly) as a dynamic memory allocator by LARGE model programs.

void **dq$free**(*segment*, *excep$p*);
token *segment*; int **excep$p*;

This routine returns the segment (previously obtained via a call to **dq$allocate**) whose base is segment to the system's free memory pool.

unsigned **dq$get$size**(*segment*, *excep$p*);
token *segment*; int **excep$p*;

This function obtains the size in bytes (with 0 representing 64K) of the segment whose base is *segment*.

Programs using the **SMALL** segmentation model can use this function to obtain the size of their expanding **DATA** segment. This is, in fact, how the standard memory allocation routines (**malloc** and **free**) determine the size of the free storage pool.

8.2 Exception Handling

int (***dq$trap$exception**(*handler$p*, *excep$p*))();
int (**handler$p*)(); int **excep$p*;

This function makes the function pointed to by *handler$p* the current exception handler. The exception handling function is called, with a single integer argument (the exception code), when an exception occurs. A pointer to the old exception handling function, or **NULL** if no handler has yet been established, is returned.

This function has the same calling sequence in both segmentation models. The actual exception handler is, in both cases, a **FAR** procedure concealed in the interface routine. This hidden routine makes an indirect call to the user's handler (using either a **NEAR** or **FAR** call, as is appropriate). The hidden routine saves all of the 8086 registers. It does not, however, save or restore the status of the numeric coprocessor (8087).

int (***dq$get$exception$handler(***excep$p*))();
int **excep$p*;

This function returns a pointer to the current execption handling function, or **NULL** if no handler has yet been established. It is not a system interface function. It simply returns the pointer to the exception handler that has been saved by **dq$trap$exception**.

The *excep$p* argument is present only for calling sequence compatibility; it is completely ignored.

void **dq$decode$exception(***code, buf, excep$p*);
int *code*; char *buf*[81]; int **excep$p*;

This routine obtains, from the system, an error message describing the error code passed in code and stores the message, as a PL/M-86 string, in the buffer *buf*.

int (***dq$trap$cc(***handler$p, excep$p*))();
int (***handler$p**)(); int **excep$p*;

This function makes the function pointed to by *handler$p* the current control C trap handling function. It returns a pointer to the old handler, or **NULL** if no handler has yet been established. The handler function is called with no arguments.

As with **dq$trap$exception**, this routine is the same in both segmentation models; it handles all of the register saving and long pointer fabrication.

8.3 Exit

void **dq$exit(***code*);
int *code*;

This routine terminates the current program. All connections are detached and all resources are released. The *code* is a completion status, which is thrown away by the system.

8.4 Get Time and Date

void **dq$get$time(***gt$p, excep$p*);
struct *gt *gt$p*; int **excep$p*;

This routine asks the system for the current time and date. This information is returned, as PL/M-86 format strings, in the supplied gt structure (which is defined in udi.h).

8.5 Get System Identification

void **dq$get$system$id**(*id*, *excep$p*);
char *id*[21]; int **excep$p*;

This routine obtains the system identification and stores it, as a standard C string, in the supplied *id* buffer.

8.6 Delete a File

void **dq$delete**(*path$p*, *excep$p*);
char **path$p*; int **excep$p*;

This routine deletes the file whose pathname is the string *path$p*. This C string is transformed into a PL/M-86 string by the interface routine via a buffer on the stack.

8.7 Rename a File

void **dq$rename**(*old$p*, *new$p*, *excep$p*);
char **old$p*; char **new$p*; int **excep$p*;

This routine renames the file whose pathname is in the C string *old$p* to the new name in the C string *new$p*.

8.8 Connection Management

connection **dq$attach**(*path$p*, *excep$p*);
char **path$p*; int **excep$p*;

This function establishes a connection to an existing file. An error will be returned if the file does not exist. The *path$p* argument is a C string containing the pathname of the file.

connection **dq$create**(*path$p*, *excep$p*);
char **path$p*; int **excep$p*;

This function establishes a connection to a new file. If the named file exists, it is deleted and recreated (truncating it to 0 length). The *path$p* argument is a C string containing the pathname of the new file.

void **dq$open**(*conn*, *mode*, *num$buf*, *excep$p*);
connection *conn*; int *mode*, *num$buf*; int **excep$p*;

This routine takes a *connection* object and prepares it for I/O operations. This involves checking access rights, allocating buffers, and, in general, preparing for actual read and/or write commands.

The *conn* argument is a connection object returned by a call to **dq$attach** or **dq$create**.

The *mode* argument specifies the desired access mode. Legal modes are 1 **(DQ$MREAD)** for read access only, 2 **(DQ$MWRITE)** for write access only, and 3 **(DQ$MUPDATE)** for read and write access. The symbolic definitions of the access modes are in the **udi.h** header file.

The *num$buf* argument specifies the number of buffers. The console is usually run unbuffered (*num$buf* = 0). Double buffering (*num$buf* = 2) is appropriate for sequentially processed connections. Single buffering (*num$buf* = 1) may be more appropriate for connections used in a random fashion.

void **dq$close**(*conn*, *excep$p*);
connection *conn*; int **excep$p*;

This routine undoes the actions of a **dq$open**. All buffers are flushed and released.

void **dq$detach**(*conn*, *excep$p*);
connection *conn*; int **excep$p*;

This routine undoes the actions of a **dq$attach** or **dq$create**. If the connection is open, it is automatically closed before it is detached.

8.9 Read from a File

unsigned **dq$read**(*conn*, *buf$p*, *count*, *excep$p*);
connection *conn*; char **buf$p*; unsigned *count*; int **excep$p*;

This function obtains up to *count* bytes from the connection *conn* and stores them into successive bytes starting at *buf$p*. The number of bytes actually read is returned. On end of file, a count of 0 is returned.

The number of bytes read is never larger than *count*, although on line edited connections it may be less than *count*.

8.10 Write to a File

void **dq$write**(*conn*, *buf$p*, *count*, *excep$p*);
connection *conn*; char **buf$p*; unsigned *count*; int **excep$p*;

This routine writes count bytes beginning at *buf$p* to the connection specified by conn. Files are automatically extended if the write goes beyond end of file.

8.11 Seek a Connection

void **dq$seek**(*conn*, *mode*, *offset*, *excep$p*);
connection *conn*; int *mode*; long *offset*; int **excep$p*;

This system interface routine moves the seek pointer in the connection specified by *conn* to the position specified by the *mode* and *offset*. The *mode* may be 1 (**DQ$BACK**) to seek backwards by *offset* bytes, 2 (**DQ$SET**) to set the seek pointer to *offset*, 3 (**DQ$FORWARD**) to seek forwards by *offset* bytes, or 4 (**DQ$ENDBACK**) to seek backwards by *offset* bytes from the end of file.

Note that the *offset* is a long integer. This is different from the PL/M-86 interface, where the high and low halves of the offset are passed as separate arguments.

8.12 Truncate a File

void **dq$truncate**(*conn*, *excep$p*);
connection *conn*; int **excep$p*;

This routine truncates the file open on the connection *conn* at the current seek position. The connection must be open for write or update.

8.13 Get Connection Status

void **dq$get$connection$status**(*conn*, *gs$p*, *excep$p*);
connection *conn*; struct gs **gs$p*; int **excep$p*;

This routine fills in the supplied *gs* structure with status information obtained from the connection *conn*.

The *gs* structure definition is in the **udi.h** header file and looks like this:

```
struct      gs     {
            char         gs_open;          /* Open flag */
            char         gs_access;        /* Access modes */
            char         gs_seek;          /* Seek modes */
            long         gs_offset;        /* Seek pointer */
};
```

If the connection is open, the **gs_open** field is set true (not zero); if the connection is not open, the field is set false (zero).

The **gs_access** field indicates the access mode of the connection. The **gs_seek** field indicates the seek operations that are legal on the connection. The **udi.h** header file contains the symbolic names of the bits in these bytes.

The **gs_offset** field is set to the current seek position. If the connection is not open or cannot perform a backward seek, it is set to garbage.

8.14 Change Extension

void **dq$change$extension**(*path$p*, *new*, *excep$p*);
char **path$p*; char *new*[3]; int **excep$p*;

This routine changes the extension of the filename in the string *path$p* to that specified by the *new* argument. If *new*[0] is a blank, the extension is stripped from the *path$p*.

8.15 Load an Overlay

void **dq$overlay**(*link$p*, *excep$p*);
char **link$p*;
int **excep$p*;

This routine loads the overlay whose link name is contained in the C string *link$p* from the current load file.

8.16 Perform Special I/O Function

void **dq$special**(*type*, *parm$p*, *excep$p*);
int *type*; connection **parm$p*; int **excep$p*;

This routine permits the setting and/or resetting of the line edit mode on the console. The *type* argument is either 1, which makes console input transparent, or 2, which makes it line edited. The **dq$special** routine does not check that the *type* argument is one of these values. Any additional codes accepted by the operating system are acceptable to this routine.

The *parm$p* argument is a pointer to a connection that represents a **dq$attach** of the :CI: device.

8.17 Command Tail Parsing

int **dq$get$argument**(*buf*, *excep$p*);
char *buf*[81]; int **excep$p*;

This routine gets the next argument from the command tail and stores it into the supplied buffer as a PL/M-86 format string. It returns the character that terminated the argument.

This routine is not normally used by C programs. Instead, the command tail has been preparsed by the runtime startoff and passed as arguments to the **main** routine.

unsigned **dq$switch$buffer**(*buf$p*, *excep$p*);
char **buf$p*; int **excep$p*;

This routine switches the input buffer used by **dq$get$argument** to a user specified area in memory. It is useful for parsing imbedded '$' control lines and other related tasks.

The first time that this routine is called, it returns 0. On subsequent calls, it returns the offset, in bytes, from the start of the buffer of the first character past the last delimiter returned by **dq$get$argument**.

APPENDIX A KEYWORDS

CC86 uses the following identifiers as keywords. They may not be used for any other purpose.

auto	extern	short
break	float	sizeof
case	for	static
char	goto	struct
continue	if	switch
default	int	typedef
do	long	union
double	readonly	unsigned
else	register	void
entry	return	while
enum		

APPENDIX B ERROR MESSAGES

The following error messages may be printed by CC86. '%s' will be replaced by a string and '%d' by a decimal number.

```
arg. list syntax
array bound must be a constant
array bound must be positive
array row has 0 length
bad argument storage class
bad base type for field
bad external storage class
bad field width
bad filler field width
call of non function
cannot add two pointers
cannot assign unlike structures
cannot declare flexible automatic array
cannot initialize fields
cannot initialize unions
cannot specify class in cast
'case' not in 'switch'
class not allowed in structure body
compound statement required
constant expression required
declarator syntax
'default' not in 'switch'
end of file in comment
enumeration constant '%s' is changing value
enumeration list syntax error
expression syntax
external syntax
extra 'long' or 'short'
field too wide
function cannot be an argument
'goto' statement syntax
identifier '%s' is not a label
identifier '%s' is not a tag
identifier '%s' is undefined
identifier '%s' not a formal
identifier '%s' not an enumeration tag
identifier '%s' not legal in expression
identifier '%s' redeclared
identifier '%s' reinitialized
identifier '%s' semantically forbidden
illegal character (%d)
illegal character constant
illegal label '%s'
illegal operation on 'void' type
illegal pointer subtraction
illegal use of 'void'
illegal use of 'void' in cast
illegal use of floating point
illegal use of pointer
illegal use of structure
indirection through non pointer
initializer too complex
left context required
left side of '->' not usable
```

```
member '%s' is changing offset
member '%s' is changing width
member '%s' is undefined
mismatched conditional
misplaced ':' operator
misplaced 'long'
misplaced 'short'
misplaced 'unsigned'
missing %s
missing member
missing right brace
missing semicolon
multiple 'default' labels
multiple classes
multiple types
no 'break' context
no 'continue' context
non scalar field
nonterminated string or character constant
number too long
registers lack an address
return(e) illegal in 'void' function
size of %s '%s' is not known
structure or union in truth context
tag mismatch
too many case labels
too many initializers
too many structure initializers
type clash
type required in cast
undefined label '%s'
unexpected end of enumeration list
unexpected end of file
```

The following warning messages may be printed by CC86. '%s' will be replaced by a string.

```
divide by zero
empty switch
missing '='
nested comments
possible missing initializer
sizeof(function) set to 1
sizeof(void) set to 0
switch of non integer
symbol '%s' truncated to 39 characters
zero modulus
```

The following strict warning messages may be printed by CC86. '%s' will be replaced by a string.

```
%s '%s'%s is unused
constant '%s' is long
construction not in Kernighan and Ritchie
identifier '%s' not bound to register
questionable structure access
risky type in truth context
structure '%s' does not contain member '%s'
union '%s' does not contain member '%s'
```

APPENDIX C ASCII CODES

ASCII CHARACTER	HEX	PL/M-286 CHARACTER?	ASCII CHARACTER	HEX	PL/M-286 CHARACTER?
NUL	00	no	@	40	yes
SOH	01	no	A	41	yes
STX	02	no	B	42	yes
ETX	03	no	C	43	yes
EOT	04	no	D	44	yes
ENQ	05	no	E	45	yes
ACK	06	no	F	46	yes
BEL	07	no	G	47	yes
BS	08	no	H	48	yes
HT	09	no	I	49	yes
LF	0A	no	J	4A	yes
VT	0B	no	K	4B	yes
FF	0C	no	L	4C	yes
CR	0D	no	M	4D	yes
SO	0E	no	N	4E	yes
SI	0F	no	O	4F	yes
DLE	10	no	P	50	yes
DCI	11	no	Q	51	yes
DC2	12	no	R	52	yes
DC3	13	no	S	53	yes
DC4	14	no	T	54	yes
NAK	15	no	U	55	yes
SYN	16	no	V	56	yes
ETB	17	no	W	57	yes
CAN	18	no	X	58	yes
EM	19	no	Y	59	yes
SUB	1A	no	Z	5A	yes
ESC	1B	no	[	5B	no
FS	1C	no	\	5C	no
GS	1D	no	]	5D	no
RS	1E	no	∧(↑)	5E	no
US	1F	no	—	5F	yes
space	20	yes	`	60	no
!	21	no	a	61	yes
"	22	no	b	62	yes
#	23	no	c	63	yes
$	24	yes	d	64	yes
%	25	no	e	65	yes
&	26	no	f	66	yes
'	27	yes	g	67	yes
(	28	yes	h	68	yes
)	29	yes	i	69	yes
*	2A	yes	j	6A	yes
+	2B	yes	k	6B	yes
,	2C	yes	l	6C	yes
−	2D	yes	m	6D	yes
.	2E	yes	n	6E	yes
/	2F	yes	o	6F	yes
0	30	yes	p	70	yes
1	31	yes	q	71	yes
2	32	yes	r	72	yes
3	33	yes	s	73	yes
4	34	yes	t	74	yes
5	35	yes	u	75	yes
6	36	yes	v	76	yes
7	37	yes	w	77	yes
8	38	yes	x	78	yes
9	39	yes	y	79	yes
:	3A	yes	z	7A	yes
;	3B	yes	{	7B	no
<	3C	yes	|	7C	no
=	3D	yes	}	7D	no
>	3E	yes	~	7E	no
?	3F	no	DEL	7F	no

INDEX

REQUEST FOR READER'S COMMENTS

Intel's Technical Publications Departments attempt to provide publications that meet the needs of all Intel product users. This form lets you participate directly in the publication process. Your comments will help us correct and improve our publications. Please take a few minutes to respond.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this publication. If you have any comments on the product that this publication describes, please contact your Intel representative. If you wish to order publications, contact the Intel Literature Department (see page ii of this manual).

1. Please describe any errors you found in this publication (include page number).

2. Does the publication cover the information you expected or required? Please make suggestions for improvement.

3. Is this the right type of publication for your needs? Is it at the right level? What other types of publications are needed?

4. Did you have any difficulty understanding descriptions or wording? Where?

5. Please rate this publication on a scale of 1 to 5 (5 being the best rating). ______

NAME ______ DATE ______
TITLE ______
COMPANY NAME/DEPARTMENT ______
ADDRESS ______
CITY ______ STATE ______ ZIP CODE ______
(COUNTRY)

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.