intel®

iC-86 COMPILER USER'S GUIDE

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Order Number: 122085-002

iC-86 COMPILER USER'S GUIDE

Order Number: 122085-002

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

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

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

Intel retains the right to make changes to these specifications at any time, without notice. Contact your local sales office to obtain the latest specifications before placing your order.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

The following are trademarks of Intel Corporation and its affiliates and may only be used to identify Intel products:

BITBUS	i_m	iRMX	Plug-A-Bubble
COMMputer	iMMX	iSBC	PROMPT
CREDIT	Insite	iSBX	Promware
Data Pipeline	int_el	iSDM	QueX
Genius	int_elBOS	iSXM	QUEST
i	Intelevision	Library Manager	Ripplemode
$\overset{\wedge}{\text{I}}$	$int_eligent$ Identifier	MCS	RMX/80
I^2ICE	$int_eligent$ Programming	Megachassis	RUPI
ICE	Intellec	MICROMAINFRAME	Seamless
iCS	Intellink	MULTIBUS	SOLO
iDBP	iOSP	MULTICHANNEL	SYSTEM 2000
iDIS	iPDS	MULTIMODULE	UPI
iLBX			

VAX is a registered trademark of Digital Equipment Corporation.

A1234 / 984 / 2K / DD / KH
SOFTWARE

REV.	REVISION HISTORY	DATE	APPD.
-001	Original issue.	6/83	
-002	Update to document new records and correct any previous errors in text.	9/84	C.C.

PREFACE

This manual describes the C programming language compiler for the Intel iAPX 86 family of microprocessors. It is intended to support new users as well as those already familiar with the C programming language.

This manual consists of eight chapters and four appendixes:

- Chapter 1, "Introduction," presents an overview of this particular implementation of the C programming language.
- Chapter 2, "Compiling a Program," details iC-86 compilation under VAX/VMS, Series III, Series IV, and iRMX-86.
- Chapter 3, "Linking C Programs," explains the method for linking C programs and special consideration applying to programs that use floating point.
- Chapter 4, "Using the Standard Libraries," describes the run- time libraries.
- Chapter 5, "Run-Time Issues," provides information on interfacing iC-86 code with code generated by other Intel translators, such as ASM86 or PL/M-86. It explains the calling sequence used by C functions, machine register conventions and other low-level issues.
- Chapter 6, "Special Considerations," describes miscellaneous issues regarding the compiler run-time environment such as absolute addressing and programming debugging.
- Chapter 7, "The Standard (libc) Library," describes the library routines used to perform many of the common programming tasks.
- Chapter 8, "The System Interface (DQ$) Library," explains the system interface (DQ$) routines.
- Appendix A, "Keywords," provides a list of identifiers that are used as keywords by iC-86.
- Appendix B, "Error Messages," is a list of the error messages generated by the compiler.
- Appendix C, "ASCII Codes," provides the standard ASCII to Hex conversion tables.
- Appendix D, "iRMX-86 Libraries," describes the RMX-86 interface library and linking under iRMX-86.

Notational Conventions

`UPPERCASE`	Characters shown in uppercase must be entered in the order shown. You may enter the characters in uppercase or lowercase.
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:
directory-name	Is that portion of a *pathname* that acts as a file locator by identifying the device and/or directory containing the *filename*.
filename	Is a valid name for the part of a *pathname* that names a file.

pathname	Is a valid designation for a file; in its entirety, it consists of a *directory* and a *filename.*
pathname1, pathname2, ...	Are generic labels placed on sample listings where one or more user-specified pathnames would actually be printed.
system-id	Is a generic label placed on sample listings where an operating system-dependent name would actually be printed.
V*x.y*	Is a generic label placed on sample listings where the version number of the product that produced the listing would actually be printed.
[ ]	Brackets indicate optional arguments or parameters.
{ }	One and only one of the enclosed entries must be selected unless the field is also surrounded by brackets, in which case it is optional.
{ }...	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 vertical bar separates options within brackets [ ] or braces { }.
...	Ellipses indicate that the preceding argument or parameter may be repeated.
[,...]	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.
BOLDFACE	Names of routines which must be entered exactly as they appear in the manual.

Related Publications

For further information on issues raised in this manual, refer to the following Intel publications:

- *Pascal-86 User's Guide* (121539)
- *PL/M-86 User's Guide* (121636)
- *PSCOPE 86 User's Guide* (121790)
- *8086/8087/8088 Assembly Language Reference Manual for 8086-Based Development Systems* (121627)

- *Guide to Using iRMX 86 Languages* (143907)
- *Introduction to the iRMX 86 Operating System* (9803124)
- *iRMX 86 Debugger Reference Manual* (143323)
- *iRMX 86 Nucleus Reference Manual* (9803122)
- *iRMX 86 Systems Programmer Reference Manual* (142721)
- *Intellec Series III Console Operating Instructions* (121609)
- *Intellec Series III Programmer Reference Manual* (121618)
- *Intellec Series IV Operating and Programming Guide* (121753)
- *iAPX 86,88 Family Utilities User's Guide* (121616)
- *Run-Time Support Manual for iAPX 86,88 Applications* (121776)

CONTENTS

FIGURES

CHAPTER 1
INTRODUCTION

This user's guide describes the C compiler and run-time system for Intel iAPX 86 family of microprocessors. The C compiler may be used as a cross compiler running under VAX/VMS, or may be used as a native compiler running under ISIS on the Intel series III or Series IV microcomputer development system or under iRMX 86 on supported boards and systems.

The Intel C compiler compiles programs written in the C programming language, as described in *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. Supported storage classes 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 is written to the object. Additional data types may be derived from the fundamental types by using arrays, functions, pointers, structures, and unions.

Lines beginning with # are preprocessor directives. The iC-86 preprocessor supports the directives #define, #else, #endif, #if, #ifdef, #ifndef, #include, #line, and #undef, as described in *The C Programming Language.*

iC-86 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 used in the definition of a function that returns no value, data type void is used to prevent the use of a null value in a value context. The derived type enum specifies an enumerated data type. iC-86 also supports structure assignment and allows functions to take structure arguments and to return structure values.

iC-86 translates programs into relocatable object files or assembly language-like source files. Once generated, relocatable object code may be linked with the standard C run-time support libraries (using LINK86) and, if necessary, converted into an absolute module (using LOC86). iC-86 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). Two byte pointers permit extremely compact and efficient code to be generated; thus, this model is recommended for programs that can satisfy the size requirements just given.

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.

iC-86 does not support the MEDIUM or COMPACT models of segmentation.

The run-time 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.

The run-time library comes in two different versions: one is used by SMALL model programs, 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.

iC-86 can optionally specify symbols, lines and type information for use by Intel debuggers such as PSCOPE-86, I²ICE and ICE-86A.

The C language has been enhanced slightly to make it easier to program 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 exactly as it is accepted in PL/M-86 (it is silently thrown away). This makes it possible for calls to the system interface library routines to look exactly like the corresponding PL/M-86 system calls.

1.1 Recent Additions to C Language

The C language as defined in the book "The C Programming Language" by Brian Kernighan and Dennis Ritchie has undergone some extensions since the publication of this book. The purpose of this section is to describe the recent additions to the language supported by the Intel C compiler.

1.1.1 The Void Type

A recent addition to C is the addition of the type **void**. Declaring a function to be of the type **void** indicates that the function does not return a value. This is useful in error checking and enhances readability of programs. Only functions may be declared to be **void**.

1.1.2 The enum Type

Another recent addition to C is the enumerated type. This is similar to the sub-range types of Pascal. An enumerated type declaration lists a set of identifiers, which may then be used as values for data of that enumerated type. An enumerated type declaration may or may not associate a tag with the type, similar to structure or union tags. For example, the declaration:

```
enum opinion { yes, maybe, no } x;
```

declares an enumerated type **opinion** with three values: **yes**, **no**, and **maybe**, and declares a variable of the type named **x**. This type may be used in further declarations analogously to structure and union tags. The declaration:

```
register enum opinion *op;
```

declares a register pointer to the enumerated type **opinion**.

To add this type to the formal definition of the language in *The C Programming Language*, amend the list of type-specifiers in section 8.2 of Appendix A of that book to include the entry *enum-specifier*. The syntax of this type-specifier is shown in Figure 1-1.

enum-specifier:
enum { *enum-list* }
enum *identifier* { *enum-list* }
enum *identifier*

enum-list:
enumerator
enum-list, *enumerator*
enumerator:
identifier
identifier = *constant-expression*

Figure 1-1. enum-specifier

The identifier in the enum-specifier, as already indicated, behaves exactly like the structure or union tag in a structure-specifier or union-specifier. The identifiers in the enum-list must all be distinct, and must be distinct from other, ordinary identifiers. They are constants, and may appear in any context where constants are appropriate. Values are assigned from left to right, beginning with 0 and increasing by 1. If an enumerator containing an equal sign ('=') appears in the list, the identifier in that enumerator is set to the value of the constant-expression, and subsequent enumerators increase by 1 from that value.

1.1.3 Structure Assignment and Passing

The definition of C in *The C Programming Language* prohibits the assignment of structures, the passing of structures to functions, and the returning of structures by functions. Another recent change to C is the lifting of these restrictions. iC-86 allows structures to be assigned, provided they are of the same type, and allows structures to be passed to and returned from functions.

1.1.4 `'\v'` Vertical Tab Literal Character

iC-86 recognizes the literal character `'\v'` for the ASCII vertical tab (octal 013) character. This literal character may be used within strings, the same as the literal characters `'\n'` and `'\t'`. It is included in the definition of white space; in particular, the **ctype** macro **isspace** is true for `'\v'`.

1.1.5 Initialization of Automatic Aggregates

One of the restrictions on C lifted by iC-86 is the prohibition of initialization of automatic aggregates, made in section 8.6 of Appendix A of *The C Programming Language*. iC-86 allows automatic arrays and structures to be initialized, provided the size of the array, or any array contained within a structure, is known. The initialization has the same form as the initialization of the external aggregate, but is performed on entry to the routine instead of at compile time.

1.1.6 Addition Type Specifiers

Section 8.2 of Appendix A of *The C Programming Language* implies that only one type-specifier may be used in a declaration, except for those cases shown in the left hand column of Figure 1-2, below. iC-86 recognizes several additional cases, all involving the adjective "unsigned" as listed in the right-hand column of Figure 1-2.

The first pair of "unsigned" terms have the same meaning, as do the second pair. The type unsigned char is an addition to the language. If it is used in arithmetic expressions it is automatically cast to unsigned int.

short int	unsigned long int
long int	unsigned long
unsigned int	unsigned short int
long float	unsigned short
	unsigned char

Figure 1-2. Type Specifiers

1.1.7 The #assert Preprocessor Directive

The only language construct recognized by iC-86 that is an entirely new construct is the **#assert** preprocessor directive, which has the form:

#assert *constant-expression*

The preprocessor evaluates the constant expression. If it is false (zero), it prints a diagnostic message. However, the failure of a "**assert** directive is not considered a fatal error, and does not terminate the compilation process.

1.2 Additional Features of iC-86

1.2.1 Sizes and Formats of Types

The sizes of various types in bits in iC-86 are shown in Figure 1-3.

There are no alignment restrictions on data in iC-86, because the 8086 processor imposes no such restrictions. The order of significance of bytes and words in iC-86 is that adopted by the 8086 processor, namely that the significance of bytes increase with address. Dependence on these facts may result in code that is not portable to other processors. Floating point types use the 8087 formats.

1.2.2 Type Conversions

In arithmetic expressions signed types promote to signed types by sign extension, and unsigned types to unsigned types by zero padding. For example, the type **char** promotes to the type **int** by sign extension, but the type **unsigned char** promotes to the type **unsigned int** by zero padding.

char	8
short	16
int	16
long	32
pointers	16 (small model)
pointers	32 (large model)
enum types	16
float	32
double	64

Figure 1-3. Bit Size in iC-86

1.2.3 Register Variables

iC-86 allows up to two register variables. These register variables may be used to hold any 16 or 8 bit quantity; however, because of inefficiencies of the architecture the type **unsigned char** cannot be supported in a register.

1.2.4 The argc, argv Parameters to main

The command line arguments, called the "command tail" are placed in a known location in low memory. The iC-86 runtime startup parses this command tail in the standard **argc**, **argv** pair expected by many C programs as arguments to **main**. This enhances the portability of C programs written under iC-86.

A caution concerning the cases of arguments. UDI changes all alphabetic characters in the command tail to upper case. Because most C programs expect arguments to be in lower case, the iC-86 runtime start-off maps all alphabetic characters back to lower case. Although a compromise, this approach seems to be the most helpful in enhancing the portability of programs developed under iC-86.

1.2.5 I/O Redirection

Another of the services provided by **main** is the processing of I/O redirection arguments on the command line, and the initialization of the standard streams, **stdin**, **stdout** and **stderr**. I/O redirection works as follows:

< *file* — An argument of this form causes the standard input stream, **stdin**, to take its input from *file*.

> *file* — An argument of this form causes the standard output stream, **stdout**, to be redirected to the named *file*. If *file* does not exist, it is created; if it does exist, the previous contents are lost.

> > *file* — This is identical to the last form, except that if *file* exists, the standard output stream is appended to the file, instead of overwriting it.

Redirection specifications are not included in the argument count argc or the argument vector argv passed to main. The standard error output **stderr** can not be redirected, and always goes on to the console.

If any of the above forms is preceded by a sharp character (**'#'**), the appropriate stream will be redirected in binary mode.

CHAPTER 2
COMPILING A PROGRAM

2.1 Compilation on the Series III, Series IV 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. For details on placing the Series III into 8086 execution mode and executing commands in this mode, see the *Series III Console Operating Instructions*. The same instructions apply to the operation of iC-86 on the Series IV.

The general syntax of the invocation line is

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

where

:Fn:	specifies the drive number if not on zero. The default drive is zero.
CC86	is the name for the compiler.
inputfile	is a filename which you specify. It contains a C source program prepared with one of the standard text editors.
outputfile	is the optional file name you enter for output.
controls	optionally specifies one or more compiler controls you may enter.

The input file is a standard ASCII file that contains a C source program prepared with one of the standard text editors.

The output of the compiler, whether object code or an assembly language program, is normally written to a file on the same device as the source file and with the same name as the source file, but with the file type changed to .OBJ or .A86. The TO output file control may be used to place the output in any desired file.

iC-86 uses two temporary files, allocated on the :WORK: device.

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

DEBUG	The DEBUG control causes iC-86 to place debugging information (symbols, their types, and line number records) into the object module. The default is no debugging information.
LARGE SMALL	This segmentation control causes iC-86 to generate code that uses the assumptions of the specified model of segmentation. The default is SMALL model.
ASM86	The ASM86 control causes iC-86 to generate assembly language. The output file has a file type of .A86 (instead of default.OBJ). The code is formatted in a style easily understood by ASM86 users; 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 trace may be used to obtain a step by step trace of progress through a large compilation on a 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. There is a limit of 3 nested #include controls. In the source file, #include has two types of requests: #include "*file*" and #include < *file* >. In both cases, additional names may be supplied by INCLUDE directives (e.g., INCLUDE (*name1*), ..., INCLUDE (*nameN*). In either case, if no additional names are supplied, the preprocessor attempts to open file. If names are supplied by the INCLUDE controls, the treatments differ. In the case #include "*file*", the preprocessor first attempts 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 *name1*file, ..., *nameN*file. In the case# include < file >, the preprocessor first attempts to open the files *name1*file, ..., *nameN*file, and then attempts 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 control is used only to remove one of the preprocessor's built-in definitions.

ROM
RAM

This control directs the placement of constants in the object module. The impact of this control on the segmentation of the generated code is described in Chapter 5. The default is RAM control.

PRINT(*file*)

The PRINT control directs the iC-86 compiler to send all messages, normally sent to the console by default, to the specified file.

OPTIMIZE(*n*)

The OPTIMIZE control governs the kinds of optimization to be performed in generating object code. *n* may be 0, 1, or 2. OPTIMIZE(0) is the least level of optimization (constant folding) and is recommended when debugging programs with PSCOPE OR I²ICE. OPTIMIZE(2) is the highest level of optimization that can be specified and includes constant folding, deletion of unused labels and dead or useless instructions, and simplification of common code sequences and jumps. It also performs peephole optimizations. OPTIMIZE(1) specifies all level 2 optimizations except peephole optimization.

2.2 Compilation Under iRMX™ 86

The invocation and control for iC-86 under iRMX86 are the same as those for iC-86 on Series III.

2.3 Compilation Under VAX®/VMS

The invocation, control instructions and their meanings under VAX/VMS are the same as those on Series III above except that the default object file extension is .086 instead of .OBJ.

CHAPTER 3
LINKING C PROGRAMS

The C compiler distribution kit includes two C run-time libraries (SCLIB.LIB for the SMALL model of segmentation and LCLIB.LIB for the LARGE model of segmentation). The kit includes two run-time start-off 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 run-time start-off routine, the appropriate C library, and the appropriate Intel interface library (SMALL.LIB or LARGE.LIB).

Following is a typical command sequence for linking a SMALL model program called SMALLP with the standard libraries:

```
-RUN LINK86 SQMAIN.OBJ, SMALLP.OBJ, &
>SCLIB.LIB, SMALL.LIB, 87NULL.LIB TO SMALLP.86 &
>BIND MAP SEGSIZE(STACK(+200H), MEMORY(3000))
```

A load time locatable (LTL) image is being created (the BIND control is used). In the SMALL model the LTL is necessary for the dynamic memory allocation routines, used for correct functioning by the standard I/O library. The size of the MEMORY segment is determined by a call to DQGETSIZE, 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 sufficient free space is unavailable files cannot be opened and operations that require space in the dynamic storage allocation pool cannot be performed.

It is recommended that SQMAIN.OBJ be linked as the first file to avoid a pointer to DS:0.

The iRMX 86 operating system allocates memory for file connections, file buffers, and other operating system objects from a memory pool. The MEMPOOL control of LINK86 specifies the minimum amount of memory required to load the program and the above-mentioned objects. Since the size of the buffers is device-dependent and the number of buffers is program-dependent, a trial and error approach works best.

The maximum memory pool size needs to be coded to avoid partition fragmentation; it should be set to 0FFFF0H. For SMALL programs specify:

```
MEMPOOL (150000, 0FFFF0H)
```

For LARGE programs total the segment sizes, add sizes of dynamically allocated memory, and then add approximately 15K for UDI overhead.

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 on the Series III. 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. For RMX 86,

400H bytes is needed for operating system processing. If no other stack requirements are made, this can be specified as

```
SEGSIZE(STACK(+400H))
```

on the link invocation. Additional stack space is needed for recursive function calls (the same amount of space is required for each level of recursion) and for the local buffers used by UDI system calls.

The link command for a LARGE model program is similar to that of a SMALL model program. Following is an example of a typical command sequence for linking a LARGE model program called LARGEP:

```
-RUN LINK86 LARGEP.OBJ, LQMAIN.OBJ, &
>   LCLIB.LIB, &
>   LARGE.LIB, &
>   E8087.LIB, E8087, &
>   LARGE.LIB, 87NULL.LIB &
>   BIND MAP SEGSIZE(STACK(1000H))
```

Once again, the BIND control has been used to create a load time locatable (LTL) module. This speeds program development because a LOCATE step is not required; however, the actual loading of the program is quite slow since all of the absolute segment bases in the image must be fixed up. Absolute programs do, however, work properly in the LARGE model, because 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.

The more elaborate features of LINK86 and LOC86 all, of course, work with the object modules produced by the C compiler. Detailed descriptions of the 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.

3.1 Floating Point

Several special considerations apply to programs using floating point. The link command must include the appropriate floating point library: 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 (SCLIB.LIB or LCLIB.LIB) in the LINK86 command.

NOTE

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

With either hardware or software floating point, the run-time start-off 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.

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 decoy, which always prints the string {Float}. The real floating point output conversion routines are 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. To guarantee yourself floating point conversion, delete the decoy 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 run-time 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.

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

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

4.1 Standard Definitions

A number of header (.h) files are supplied with the libraries. These files, 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 header files and their applications are listed below.

stdio.h This is the most important and most often used of the header files. It contains all of the definitions used by the I/O routines, a number of symbolic constants (the value of the **NULL** pointer, for example) and external definitions for the library routines that return non-integer objects.

udi.h This header file contains typedefs, structures, macros and symbolic constants used to interface with the UDI.

assert.h This defines the **assert** macro used for program verification.

ctype.h This defines the character classification macros, such as **isascii** and **isupper**.

rmu.h This header file contains flags and register definitions for using the 8087 numeric data processor.

setjmp.h This defines a type needed for use by the pair of routines, **setjmp** and **longjmp**.

4.2 Overall Structure of Programs

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

Programs may terminate in two ways. The easiest way is to simply terminate the main routine, returning control to the run-time start-off code, that performs some cleanup operations and returns control to the operating system. Some situations (errors, perhaps) may require a program to be terminated, and returning to a the main module may not be desirable (or even possible). When these conditions arise,

use the exit routine. It performs the standard cleanup and returns control to the operating system.

A second exit routine, _exit, quickly returns control to the operating system without performing any cleanup. This routine should be used only as a last resort because bypassing the cleanup will leave files open and will leave buffers of write data in memory.

4.3 Strings

A common data structure in C programs is the character string. The usual run-time representation for a string is an array of characters delimited by a 0 byte (\ 0). This representation is, in fact, the one used by the C compiler when a program contains a string constant (e.g., "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 holds a string of 19 (not 20) nonnull characters, delimited by a 0 byte.

Strings can often be assigned simply by shuffling pointers. If, however, characters must actually be moved, use the library routine. This function has two arguments. The first (a pointer to a string) points to the destination array; 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, as evidenced by the following example:

```
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 library function strcat 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 of it to hold the new characters. Strcat returns a pointer to the new result, delimited by a 0 byte.

A typical use of strcat is to create file names. In this case, a specific file type must be appended to a name that changes at run-time. 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'');
```

Strings must often be compared. This must be done, for example, if a list of strings is being sorted. The library function strcmp performs string comparison. This function 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), is equal to 0 if the two strings are equal, and is greater than 0 if the first string is greater than the second string.

Applications dealing with fixed length strings can use strncat, strncpy, and strncmp. These routines perform the same functions as their variable length counterparts (without the n); 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 transfers, word by word transfers, and string, block, and formatted transfers. All I/O modes may be freely intermixed.

4.4.1 The FILE type

The standard I/O header file stdio.h has 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 either 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). This character specifies that this is a binary (as opposed to an ASCII) stream and 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, the named file will be truncated to zero length.

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

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

stdin, stdout, and stderr are defined as macros in the header file stdio.h. 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. This argument is usually 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 (via internal calls to fclose in exit) when a program terminates.

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 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 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 the next character from the FILE or EOF. The definition of EOF is in the header file stdio.h. 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.

The routine getchar is equivalent to getc(stdin). getchar reads characters from the standard input FILE, normally the keyboard.

The routine ungetc(c, fp) returns c to the FILE fp. This routine is useful for looking ahead at the next input character 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 that contains the byte to be written, fp is a pointer to an output FILE. The first argument is returned unless a write error occurs, in which case EOF is returned.

The routine putchar(c) is equivalent to putc(c, stdout). putchar(c) writes characters to the standard output FILE, normally the video display.

Following is a simple example that uses the I/O routines discussed so far. The characters in the file mumble.c are copied to the display. A rude diagnostic is printed 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. The program must instead 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 returns the data and an error is posted to the FILE. This error may be detected by using the macro ferror(fp). End of file is posted only if a call to getw gets no data.

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

4.4.6 String I/O

A number of routines perform I/O on strings.

The most basic string read routine is fgets(b, n, fp). This routine reads a new line-delimited string from the FILE pointed to by fp and stores it into the array of characters (b). The new line-delimited character is transferred to the buffer, followed immediately by a 0 byte. 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.

C also has a routine called gets(b). This routine reads a new line-delimited string from the standard input stream and stores it in the array of characters (b). The new line is then deleted (this is different from fgets), and 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. A routine called puts(s) writes the string pointed to by s, followed by a newline, to the standard output.

In the following example the program reads a filename from the keyboard and 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 facilities are most often used on binary streams to move raw binary information to and from files. However, these library facilities 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 bytes 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

The C language provides routines 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 just interfaces to putc and getc). These routines are useful when dealing with scrambled 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. Section 7.8 describes the details of all their formatting options.

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 as determined by the format specification. The type of the argument must agree with that expected by the format specification. If the type does not agree (for example, if a long integer is placed in the argument list where a normal integer is expected), the result is undefined.

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

Following 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. The format string also contains some literal characters that get copied directly into the output.

4.4.9 Random Access

All of the examples given up to this point deal with sequential access FILE streams. However, the I/O library supports random access transfers as well. A seek pointer is associated with every FILE. This pointer starts off at the beginning of the file (except, of course, when a stream is opened for append, in which case 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 routine 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-position relative (how = 1), or end-of-file relative (how = 2). Fseek has no defined return value.

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.

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

The following example program opens a file on the disk and then lets the user display, by number, 8-byte fixed length records.

```
#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

Data often needs to be sorted. However, good sorting routines are tricky and difficult to debug. Because of this, the standard library contains two sort functions that 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 routine qsort(b, n, size, f) 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 routine shellsort(b, n, size, f) has exactly the same calling sequence but uses Shell's sorting method. For most purposes, qsort is preferable.

The routine quicksort 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.

In the following 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 Allocating Dynamic Memory

When building linked data structures or dealing with arrays whose size can be determined only at run-time, allocate blocks of memory dynamically. The standard functions malloc, calloc, and free implement a general-purpose memory allocation system that is used by user programs and 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 creates a small (and probably unusable) block on the list of free memory. The block contains garbage; it is not initialized in any way. If the memory left in the free space pool is insufficient, a NULL pointer is 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 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 routines permit files to be renamed and deleted, and exceptions to be caught (including the control C key). These routines also permit other low-level operations.

All of these routines, along with their calling sequences, are described in Chapter 8. For additional details see the *Series III System Programmer's Reference Manual.*

4.8 Odds and Ends

Some routines in the library perform conversions between character strings and binary values, generate random numbers, and perform other required actions. Chapter 7 describes all of these routines.

CHAPTER 5
RUN-TIME ISSUES

The aim of this chapter is to assist those users who must interface code generated by iC-86 with code generated by other Intel translators, such as PL/M-86 or ASM86. Described in detail are the calling sequences used by C functions, the conventions regarding the use of machine registers, the manner in which segment registers are set up, and other, low-level issues. Experienced users, already familiar with Intel translators, may choose to skip this chapter.

The run-time environment used by the SMALL model of segmentation is quite different from that used by the LARGE model of segmentation. Mixtures of the two models may work (and may, in fact, be necessary) in some circumstances. However, models should be mixed only by the most experienced of users.

5.1 Small Model of Segmentation

5.1.1 Segment Names and Attributes

In the SMALL model of segmentation, a program has two segments, each 64K bytes (maximum) in size. One segment, mapped by the CS segment register and spanned by the group CGROUP, contains all of the machine code generated by iC-86. The other segment, 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.

iC-86 places all instructions in a segment called CODE. This segment 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. This segment 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. This segment has a class name of DATA. The 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 of segmentation. iC-86 segment names, class names, groups, and attributes are completely compatible with PL/M-86 segment names, class names, groups, and attributes. The rules stated in the ASM86 Language Reference Manual (order number 121703) 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

The C calling sequence is different from the calling sequence used by PL/M-86 (and other Intel translators). 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 frequently have a variable number of arguments. In fact, the two formatted I/O routines in the standard library (printf

and scanf) 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. Furthermore, 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.

The calling sequence below is used. The function arguments are pushed, right to left, as described in detail below. The function is then called, either directly or indirectly, with a NEAR CALL instruction. An ADD instruction after the call removes the arguments from the stack.

Function arguments are pushed as follows.

char	widened to int, then pushed
int	pushed
long	high order word pushed, then low order word pushed
float	widened to double, then pushed in 8087 order
double	pushed in 8087 order
struct	pushed in memory order
union	pushed in memory order
pointer	offset pushed

For example, the following function call:

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

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

generates the code below:

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

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

Because C functions may use registers SI and DI for register variables, the C prolog saves SI and DI. It also saves and resets BP. Arguments are at offsets B and up from BP; locals are at offsets -2 and down. The SP points at the local variable with the lowest address. The C epilog resets SP from BP and then restores DI and SI before returning. Segment Registers are unchanged; all other registers have unknown contents.

Functions return values as follows. Functions returning struct or union objects actually return a pointer to the struct or union; the code generated for the function call block moves the result to its destination.

char	in AL
int	in AX
long	in DX:AX
float	on 8087 stack
double	on 8087 stack
struct	pointer in AX
union	pointer in AX
pointer	in AX

For more detailed information, use the ASM86 option in iC-86 to examine the generated code.

```
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

C programs use the stack provided by the SYSTEMSTACK module in SMALL.LIB. The C run-time start-off routine contains a zero length stack segment that has a symbol at the end of it. This symbol is relocated to the 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 run-time start-off routine and/or the loader initializes the segment registers CS, DS, ES and SS. It also sets up SP. Interrupts disabled before touching any of these registers, are unconditionally enabled when the initialization is completed. The same startup routine handles both absolute images and LTL images. As noted earlier, a C function will preserve SI, DI, BP, SP and the segment registers; other registers may be clobbered.

5.1.5 Command Line Processing

The command line that invoked the C program is obtained by calling DQGETARGUMENT repeatedly until an argument string delimited by a carriage return is encountered. This command argument is 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.

As explained in Section 1.2.4, all alphabetic characters in the arguments are mapped to lower case.

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 simply by placing a label into the segment in SQMAIN) from the size of the data segment (obtained by a call to DQGETSIZE). 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 DQGETSIZE does not return a useful result if the program has been bound as an absolute image (that is, if the program has been processed by LOC86). Consequently, the storage allocator malfunctions if it is used by an absolute program - which means that absolute programs may not use the standard I/O package without providing their own versions of malloc and free. This is so because 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 conventions. 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 (e.g., the interface routines of RMX86), the linkage must be written in ASM86. In the following simple example, assume 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          ; Reserve locals, if needed

            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 of Segmentation

5.2.1 Segment Names and Attributes

Object modules generated by iC-86 in the LARGE model of segmentation always contain two segments. One segment 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; the data segment always has class name DATA. These same naming conventions are used by PL/M-86 in the LARGE model of segmentation.

The segment name_CODE includes code, linkage vectors, literals, switch tables, and read only data. If the ROM control is specified, it also includes strings. The name_DATA segment includes 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 in the LARGE model.

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.

Because 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]

LO:     les     si,p1
        inc     word ptr p1
        mov     al,es:[si]
        cbw
        mov     c,ax
        or      ax,ax
        je      L2

        cmp     c,'A'
        jl      L1
        cmp     c,'Z'
        jg      L1
        mov     c,'!'

L1:     les     si,p2
        inc     word ptr p2
        mov     ax,c
        movb    es:[si],al
        jmp     LO

L2:     les     si,p2
        mov     es:byte ptr [si],0

        pop     bp
        pop     di
        pop     si
        ret

f_      endp
```

5.2.3 Run-Time Start-Off

The run-time start-off routine works exactly the same way in the LARGE model of segmentation 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 as long as these Intel supplied routines function correctly.

5.2.5 Interfacing the LARGE Model

The LARGE model interface to routines supplied by Intel 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.

In the following 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 opoint, 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
SPECIAL CONSIDERATIONS

This section documents some peculiarities of iC-86 and it's run-time environment and warns the new user of the more common problems, especially when C programs are ported from other machines to iC-86.

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 routine fopen takes an extra format specifier in the mode field (`''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. These kinds of problems happen most frequently when one is moving a program from an operating system that does not distinguish 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 first write to a stream, enough space may not be available to claim the I/O buffer. To make programming easier, the standard error stream preallocates its buffers. Caution should be used when writing diagnostics to the standard output or to some other stream.

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

6.3 Fields

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

iC-86 allows fields of char, unsigned char, short, unsigned short, int, and unsigned int. When referenced, fields in signed types are sign extended to integers when referenced. On the other hand, 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.

6.4 Absolute Memory Addressing

All of the iAPX 86 memory is segmented and this fact must be borne in mind when referencing absolute locations. In the SMALL case, all references to absolute locations are relative to the beginning of the group (DGROUP for DATA and CGROUP for CODE). In the LARGE case, both the offset and segment values must be specified with a 32-bit value which has segment base in the most significant 16 bits and the offset in the least signficant 16 bits.

Example:

```
/* LARGE MODEL EXAMPLE */

char c_var, *char_ptr, (*func_ptr)();

char_ptr = 0x12000345;

*char_ptr = 'A';        /* Store A in address 12345H */

func_ptr = 0x1234H;

c_var = (*func_ptr) (2);  /* call function at 01234H */
```

6.5 PSCOPE/I²ICE™ Operation

All symbols in the source program have an underscore (_) appended to them in the debug environment. For example, program symbol VAR is referenced as VAR_ under PSCOPE. Also, lowercase characters in symbols are converted to uppercase. Using names that differ only in case may lead to confusion under PSCOPE.

iC-86 programs have a separate startoff routine that collects the command line parameters etc. and then calls MAIN_. After loading, the debug cursor points to this startoff routine. Execution of program to the breakpoint MAIN_ will bring the debug cursor to the beginning of the user program main. Therefore, under PSCOPE or I²ICE, the first command line should be:

*GO TIL .MAIN_

6.6 External Identifiers

All program identifiers are mapped into uppercase when iC-86 emits the object file. Therefore, although the iC-86 compiler treats uppercase and lowercase letters as distinct characters in the program, this case sensitivity does not extend to external identifiers.

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. The descriptions give the calling sequence (the type of the return value and the types of each of the arguments) for each routine and give an explanation of the function of each routine.

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 determined easily.

The macros have a non-zero value if the condition tested is true, and value 0 if the condition is not true.

The **isascii** macro is defined on all integers. All other macros are defined only on the special value EOF and the 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 uppercase and lowercase letters and the underscore character (_).

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

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

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

The **iscntrl** macro tests if *c* is a rubout character (7FH) or a control character (less than 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 lowercase 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 a character that is neither a control character nor an alphanumeric character.

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

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

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

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

7.2 String Manipulation

The string manipulation routines work on 0-byte terminated strings stored in arrays of characters. These routines 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 to the end of the string pointed to by *s1* a copy of the string pointed to by *s2*. 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*); 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 comparision. 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 into the string pointed to by *s1* the string pointed to by *s2*. 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 in the string *s1* a pointer to the first occurrence of the character *c*. A NULL pointer is returned if the character is not in the string.

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

The **rindex** routine routine returns in the string *s1* a pointer to the last occurrence of the character *c*. A NULL pointer is returned if the character is not 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 to it the device and/or file specified by the name argument. The *name* argument is a string. Any device and/or file name defined by the operating system is acceptable. The *mode* string must be **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**, the file will be created. If the mode is **w** and the file does exist, the file is truncated to zero length (the old contents are destroyed). The *mode* string may also contain the character **b** to specifiy 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 the **fopen** routine except that it takes a third argument, *fp*. This FILE object is closed and the named file is 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*. The routine first finishes up any I/O operations associated with the FILE, then releases any I/O buffers and detaches the connection. A 0 is returned if all goes well; a -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. This routine returns a 0 if all goes well; -1 on any kind of error. **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. The routine 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 often used to prevent I/O buffers from being allocated in the dynamic storage pool in programs that require 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. feof returns non-zero when an input stream reaches end of file and 0 otherwise.

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, and a non-zero value is returned by **ferror**.

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

The **clearerr** macro clears the _FERR flag in the FILE *fp*. This macro 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*. This macro might be used, for example, to obtain the connection number to 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 any type of 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 goes 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 not normally called by users. User programs call these routines through the following 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, for example, in reading numbers when an extra character must be read 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*. The routine 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*, *fo*); int *i*; FILE **fp*.

The **putw** routine writes the next 16-bit word i to the output FILE *fp*. The routine returns i if the write is successful; EOF on any kind of error. Since EOF is a legal word value, 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*. **fgets** stops reading on the 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 is terminated by an 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. This routine has no n parameter to specify the length of the buffer. Delimiting newlines are not stored in the buffer.

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

The **fputs** routine writes onto the output FILE *fp* the 0-byte terminated string in the buffer *b*. The routine returns EOF on failure.

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

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

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. **printf** writes characters to the standard output FILE, **fprintf** writes characters to the FILE *fp*, and **sprintf** stores characters in the string *sp*.

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

Format specifications are introduced by a percent sign (%). The items below may follow a percent sign:

1. A minus sign (−) specifying that the data in the output field should be left adjusted instead of right adjusted (the default).
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 zero (0), the field is 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 asterisk (*), the next int from the *list* is used as the field width.
3. A period (.) that only separates the two decimal digit strings.
4. A string of decimal digits that specifies the precision of an **e**, **f** or **g** conversion item, or that specifies the maximum number of characters to be output by an **s** conversion item. If the maximum number is an *, the next **int** from the *list* isused as the maximum width.
5. An l specifying 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 as follows:

%	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 thefraction string ffffff is given by the precision(default 6).
f	The next **float** or **double** from the *list* is output in the format [−]*d.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 or either the **e** or **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 as arguments **printf** style format lists.
s	The next item from this *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.

If you require floating point output, see section 3.1. Floating point output may print several strings, in addition to the usual numbers. The string { Float } indicates that the real floating point output routine is not included in the link, as described in section 3.1 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. **scanf** 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 the 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 is to be stored. White-space characters (space, tab, and 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. Ensure type matching between the arguments and the conversion specifications for correct results.

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

The % character may be followed by characters that indicate the width of the input field and the conversion type. A field is delimited by white space (space, tab, and newline) or by the given field width, if any. Newlines are white space; thus, the input can handle more than one line. The following modifiers may precede the conversion type, but only in the order given:

1. An optional asterisk (*), 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 specifies 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 as follows:

   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

A long integer that contains the *seek pointer* is associated with every FILE. Initially, this pointer is offset 0 bytes from the start of the file. This pointer specifies the next byte to 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 the 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*, OL, O). This routine is provided only for programming convenience.

7.10 Sorting

The standard library provides two general sorting routines (given below) that implement only the framework of the sort. The user program must provide a routine to perform key comparision.

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 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 Allocating Dynamic Memory

The standard library provides a general-purpose dynamic memory allocation system. This system is used both by user programs and the I/O routines (in the standard libaray) 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. This routine clears memory to binary zeros and returns a pointer. The routine 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 that has been allocated by **malloc** or **calloc** and returns the block to the free memory pool. Passing to free random pointers or pointers to blocks of memory not allocated by **malloc** or **calloc** is dangerous.

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

The **malloc** routine allocates and returns a pointer to a block of memory at least *n* bytes long. The memory is not cleared. The routine 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 to a **double**, an **int**, or a **long**, respectively, a number stored as an ASCII character string. 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 this routine is called, it returns a new random number in the range 0 - $2^{15} - 1$. The generator has a period of 2^{32}. The **srand** routine can be called to seed (reset) the random number generator. Often, a timing device (an Intel 8253, for example) can be used as the source of such 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 control transfers. 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 run-time 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 run-time stack from the *env*, then makes the call to setjmp, which again sets up the environment return. However this time, the **setjmp** routine returns *value*.

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

7.13 Trigonometric Functions

#include <math.h>
double **acos**(*arg*);
double *arg*;

double **asin**(*arg*);
double *arg*;

double **atan**(*arg*)
double *arg*;

double **atan2**(*num*, *den*);
double *num*, *den*;

double **sin**(*radian*);
double *radian*;

double **cos**(*radian*);
double *radian*;

double **tan**(*radian*);
double *radian*;

The trigonometric functions are **sin**, **cos**, and **tan**. The argument *radian* should be in radian measure.

The inverse trigonometric functions are **asin**, **acos**, and **atan**. The argument of **asin** or **acos** should be in the range [−1., 1.] while the argument of **atan** is any real number. The result is in the range [−pi/2, pi/2] for **asin**, in the range [0, pi] for **acos**, and in the range (−pi/2, pi/2) for **atan**.

The **atan2** function returns **atan** of the quotient of its arguments, *num/den*, with the result in the range [−pi, pi]. The sine of the result will have the same sign as *num*, and the cosine of the result will have the same sign as *den*.

Out of range arguments set **errno** to **EDOM** and return 0. **tan** returns a very large number where it is singular and sets **errno** to **ERANGE**.

7.14 Complex Absolute Value Functions

#include <math.h>
double **cabs**(*z*);
struct { double *r*, *i*; } *z*;

double **hypot**(*r*, *i*);
double *r*, *i*;

The **cabs** function computes the absolute value (or modulus) of its complex argument *z*. The absolute value of a complex number is the length of the hypotenuse of a right triangle with sides given by the real part *r* and the imaginary part *i*. The result is the square root of the sum of the squares of the parts.

hypot computes the same value, but with *r* and *i* passed as separate parameters.

The functions return a very large number and set **errno** to **ERANGE** when the correct result would overflow.

7.15 Hyperbolic Functions

#include <math.h>
double **cosh**(*z*);
double *z*;

double **sinh**(*z*);
double *z*;

double **tanh**(*z*);
double *z*;

The functions **sinh**, **cosh**, and **tanh** compute the hyperbolic sine, hyperbolic cosine, and hyperbolic tangent, respectively. The argument *z* is in radians.

Both **sinh** and **cosh** set **errno** to **ERANGE** and return a huge value with the same sign as the actual result when overflow occurs.

7.16 Logarithmic and Exponential Functions

#include <math.h>
double **exp**(*z*);
double *z*;

double **log**(*z*);
double *z*;

double **log10**(*z*);
double *z*;

double **pow**(*z*, *x*);
double *z*, *x*;

double **sqrt**(*z*);
double *z*;

The **exp** function returns the exponential of *z*, or e^z.

The **log** function returns the natural (base e) logarithm of *z*. **log10** returns the common (base 10) logarithm of *z*.

pow returns *z* raised to the power *x*, or z^x.

The **sqrt** function returns the square root of *z*.

exp and **pow** indicate overflow by an **errno** of **ERANGE** and a huge returned value. A domain error in **log** (*z* is less than or equal to 0), in **pow** (*x* is negative and not an integer, or both *z* and *x* are 0), or in **sqrt** (*z* is negative) sets **errno** to **EDOM** and returns 0.

7.17 Bessel Functions of the First Kind

#include <math.h>
double **j0**(*z*);
double *z*;

double **j1**(*z*);
double *z*;

double **jn**(*n*, *z*);
int *n*;
double *z*;

j0, **j1**, and **jn** take an argument *z* and compute the Bessel function of the first kind for order 0, order 1, and order *n*.

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 transformations upon their parameters to make it easier to call the system from C programs. In particular, the routines 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. This file includes 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.

Following are brief descriptions of each routine. Experienced Series III programmers will find this information sufficient. Less experienced programmers should refer to the Intel publications listed in the Preface of this manual 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 long (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 model of segmentation, because the new segment may not be addressable. However, this routine is used (almost directly) as a dynamic memory allocator by programs in the LARGE model of segmentation.

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 **dqgetsize**(*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** model of segmentation 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. This function returns a pointer to the old exception handling function, or returns **NULL** if no handler has been established yet.

This function has the same calling sequence in both models of segmentation. In both cases, the actual exception handler is 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, whichever 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 (***dqgetexception$handler**(*excep$p*))();
int **excep$p*;

This function returns a pointer to the current exception handling function, or returns **NULL** if no handler has been established yet. This function 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 that describes the error code passed in code, and stores the message in the buffer *buf* as a UDI string.

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. This function returns a pointer to the old handler, or **NULL** if no handler has been established yet. The handler function is called with no arguments.

As with **dq$trap$exception**, this routine is the same in both models of segmentation; 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 **dqgettime**(*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 in the supplied *gt* structure (which is defined in udi.h) as UDI format strings.

8.5 Get System Identification

void **dqgetsystem$id**(*id, excep$p*);
char *id*[21]; int **excep$p*;

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

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 UDI 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 process 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 randomly.

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 the number of bytes 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$BACK**) to seek backward by *offset* bytes, 2 (**DQ$SET**) to set the seek pointer to *offset*, 3 (**DQ$FORWARD**) to seek forward by *offset* bytes, or 4 (**DQ$ENDBACK**) to seek backward 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 **dqgetconnection$status**(*conn, gs$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 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 **dqgetargument**(*buf*, *excep$p*);
char *buf*[81]; int **excep$p*;

This routine gets the next argument from the command tail and stores it in the supplied buffer as a UDI format string. This routine 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 run-time start-off 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 **dqgetargument** to a user specified area in memory. This routine is useful for parsing imbedded `'$'` control lines and other related tasks.

The first time 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 **dqgetargument**.

8.18 File Information

void **dq$file$info**(*conn*, *mode*, *file$info$p*, *excep$p*);
connection *conn*; int *mode*; struct *file$info* **file$info$p*;
int *excep$p*;

This routine fills in the supplied *file$info* structure with file information obtained from the connection conn. The file owner is identified if mode is 1 and is not identified if the mode is 0.

The *file$info* structure definition is in the **udi.h** header file and is shown below:

```
struct file$info {
    char    fi_owner[15];
    long    fi_length$of$file;
    char    fi_type;
    char    fi_owner$access;
    char    fi_world$access;
    long    fi_create$time;
    long    fi_last$mod$time;
    char    fi_reserved[20];
};
```

The fi_owner field is a C string identifying the system name of the file owner. The fi_type field indicates the file usage: 0 for data; 1 for directory. The fi_owner$access and fi_world$access fields describe the access rights of the file owner and others. The fi_create$time and fi_last$mod$time fields indicate the times of creation and last modification of the file.

APPENDIX A
KEYWORDS

iC-86 uses the following identifiers as keywords. These identifiers 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 iC-86. '%s' will be replaced by a string. '%d' will be replaced 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 scaler 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 fatal error messages may be printed by iC-86. '%s' will be replaced by a string; '%d' will be replaced by a decimal number.

```
%s: cannot create, argv[d]
%s: cannot open, ifn
%s: cannot reopen, argv[5]
%s: unknown option, argv [d]
8087 item
cannot allocate tree nodes
grabnval
out of space (glookup)
out of tree nodes
aerr
afupdate
asmgen op=%d r=%d mode=%d regm=%d
asmgen
bad bit
bad func %d, op
bad macro %d, c
bad rev
bad temp. file opcode %d, op
base
```

```
botch in popistack
bss
call no star
call not ofs
cannot assemble %d, opcode
code: data
collect
d def (glookup)
d def (llookup)
decrefc passed non label
genfield, mode%d, mode
goal
grablval
increfc passed non label
jump
large item
ldes
lptr
macro body too long
macro expansion overflow
missing output file
modxfun
ndp item
no ':'
no lofs
no match op=%d, tp->t_op
no patp
no rofs
node will not fix
not an ofs
options
out of space (glookup)
out of space (llookup)
out of space
out of tree space
output write error
reach, disp=%d, disp
restlocals
sdi
sdibump
sellv
switch overflow
switch underflow
temp. file write error
too many args
too many args in macro
too many cases
too many constants
too many directories in include list
too many members
too many stores
undef
write error on ouput object file
x seg #1
x seg #2
xeq
```

The following warning messages may be printed by iC-86. '%s' will be replaced by a string.

```
divide by zero
%s in macro argument
empty switch
macro %s redefined, ident
macros nested %d deep, loop likely
missing '='
multiple #else's
nested comment
nested comments
preprocessor assertion failure
possible missing initializer
sizeof(function) set to 1
sizeof(void) set to 0
switch of non integer
symbol '%s' truncated to 39 characters
trailing \", \# in initialization list
zero modulus
%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 CHARACTER SET

ASCII Character	HEX	ASCII Character	HEX
NUL	00	@	40
SOH	01	A	41
STX	02	B	42
ETX	03	C	43
EOT	04	D	44
ENQ	05	E	45
ACK	06	F	46
BEL	07	G	47
BS	08	H	48
HT	09	I	49
LF	0A	J	4A
VT	0B	K	4B
FF	0C	L	4C
CR	0D	M	4D
SO	0E	N	4E
SI	0F	O	4F
DLE	10	P	50
DC1	11	Q	51
DC2	12	R	52
DC3	13	S	53
DC4	14	T	54
NAK	15	U	55
SYN	16	V	56
ETB	17	W	57
CAN	18	X	58
EM	19	Y	59
SUB	1A	Z	5A
ESC	1B	[	5B
FS	1C	\	5C
GS	1D	]	5D
RS	1E	^(↑)	5E
US	1F	—	5F
space	20	`	60
!	21	a	61
"	22	b	62
#	23	c	63
$	24	d	64
%	25	e	65
&	26	f	66
'	27	g	67
(	28	h	68
)	29	i	69
*	2A	j	6A
+	2B	k	6B
,	2C	l	6C
-	2D	m	6D
.	2E	n	6E
/	2F	o	6F
0	30	p	70
1	31	q	71
2	32	r	72
3	33	s	73
4	34	t	74
5	35	u	75
6	36	v	76
7	37	w	77
8	38	x	78
9	39	y	79
:	3A	z	7A
;	3B	{	7B
<	3C	|	7C
=	3D	}	7D

APPENDIX D
USING iRMX™ SYSTEM CALLS IN iC-86

All of the iRMX system calls provided in the Nucleus, BIOS, EIOS, Human Interface, and Application Loader can be used in LARGE MODEL iC-86 programs. In order to make use of these calls, once the program using these calls is compiled, it has to be linked with the appropriate C Interface Library and then the appropriate iRMX Interface Library routines.

D.1 iRMX System Calls in iC-86

The following iC-86 module has a procedure that can be used to write out a buffer using BIOS calls. This procedure illustrates features such as the correspondence between C data types and iRMX data types, and how to address structures embedded within iRMX 86 objects. Following that are the commands needed to compile and link the program:

```
#include ‹ stdio.h ›

/* BIOS_write : Procedure to write out a string to a device using BIOS calls.

   Interface variables :

      device_conn : token of the connection to the device to which to write.
      mbox_token : token of a mail box, created by the main program, and used in
                   asynchronous call to bios.
      string_ptr  : pointer to the array of bytes that are to be written out.
      char_count  : number of characters that are to be written out.

*/

bios_write (device_conn,string_ptr,char_count,mbox_token)

unsigned int dvice_conn, mbox_token, char_count;
char *string_ptr;

{

      unsigned int exception_code;
      unsigned int dummy;

      typedef struct iors_structure {
                                       unsigned int status;
                                       unsigned int unit_status;
                                       unsigned int actual;
                                     } iors_struc;

      union   {
                  struct { unsigned int offset, base; } ptr_overlay;
                  iors_struc *iors_pointer;
              }   union_struct;

#define iors (*(union_struct.iors_pointer))
```

```
    rqawrite (device_conn,string_ptr,char_count,mbox_token,&
            exception_code);
    if (exception_code !=0)
      {
          printf(''Exception %x on write. \n'',exception_code);
          exit(2);
      }
    union_struct.ptr_overlay.offset = 0;
    union_struct.ptr_overlay.base = receivemessage(mbox_token,
                                &0xffff,dummy,&exception_code);
    if (iors.status !=0)
      {
          printf(''Status = %x on write. \n'',iors.status);
          exit(2);
      }
    rqdeletesegment(union_struct.ptr_overlay.base,&exception_code);
}   /* end bios_write */
```

To compile the above program named DEMONSTRATE.C, use

```
cc86 DEMONSTRATE.C large
```

Once the object file is produced, you can get an executable file by using

```
link86 DEMONSTRATE.OBJ, /clibrary/rcifl.lib, /clibrary/lclib.lib, &
       /clibrary/icifl.lib, /clibrary/lqmain.obj, &
       /rmxlibrary/large.lib, /rmxlibrary/87null.lib, &
       /rmxlibrary/ipifl.lib, /rmxlibrary/rpifl.lib &
   to DEMONSTRATE bind map &
       mempool(+20000,500000) segsize(stack(+512))
```

Notes

This is an example link command that assumes for readability that all the libraries related to iC-86 are in the directory /clibrary, and that all the libraries related to RMX 86 are in the directory /rmxlibrary.

The library rcifl.lib is the iC-86 interface to RMX 86 Nucleus calls. Since the example used Nucleus calls, this library has been linked in. Similarly, the RMX library rpifl.lib has been linked in the sample program; it is the RMX library for programs that make calls to the Nucleus. The libraries icifl.lib and ipifl.lib were included to allow BIOS calls in the example.

Following are the libraries you need to use to make calls to the different layers of RMX 86:

Layer Name	iC-86 Library	RMX 86 Library
Nucleus	**rcifl.lib**	rpifl.lib
BIOS	**icifl.lib**	ipifl.lib
EIOS	**ecifl.lib**	epifl.lib
Human Interface	**hcifl.lib**	hpifl.lib
Application Loader	**lcifl.lib**	lpifl.lib

As you can see in the example, the way to address structures embedded in RMX 86 objects is to overlay the pointer to the embedded structure with a pointer structure. The union definition does just that , and by putting the token of the object into the base field, you create a pointer to the structure in the object.

The correspondence between RMX 86 data types and iC-86 data types is as follows:

RMX 86	iC-86
Token	Unsigned int
Pointer	char *
Word	Unsigned int
Byte	char
Selector	Unsigned int
Offset	Unsigned int
RMX (or UDI type string)	c type null terminated string

1. Wherever the RMX calls require a string as one of the parameters, the user supplies a c_type null terminated string. Similarly calls that return strings return c_type strings.
2. Users are to be warned that they might not be able to use the Human Interface parsing calls without modifying the sq/Lq main.A86. This is because main.A86 initially gets all the arguments, and thus the HI parse buffer is closed. Users who want to use HI calls for parsing should modify their sq/Lq main.A86.
3. All of the iRMX calls follow the interface specified in their respective manuals except for the two calls rq8create$ file and rq8delete$file. For these two calls, the RMX manuals say that the subpath$ptr parameters is not used in physical files. But the c_library interface requires a null string here, even though these parameters are not used.
4. For those calls that return strings, the returned string should not be trusted if an exception occurs on the call.

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

Printed in U.S.A.