

UNISYS

BTOS

C Compiler

**Programming
Reference Manual**

Relative to Release
Level 1.1

Priced Item

September 1988
Distribution Code SA
Printed in U S America
5029879

UNISYS

BTOS

C Compiler

**Programming
Reference Manual**

Copyright © 1988 Unisys Corporation
All Rights Reserved
Unisys is a trademark of Unisys Corporation

Relative to Release
Level 1.1

Priced Item

September 1988
Distribution Code SA
Printed in U S America
5029879

The names, places and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THE DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Comments or suggestions regarding this document should be submitted on a User Communication Form (UCF) with the CLASS specified as 3 (Applic.:Application Software), the Type specified as 3, and the product specified as the 7-digit form number of the manual.

About This Manual

This manual contains procedures and reference information on how to install and use the BTOS C Compiler, and then link the object code into an executable run file.

Who Should Use This Manual

This manual is designed for users who have a working knowledge of the C programming language; it is not intended to teach a user how to write a program in C.

The procedures are easier to perform if you are familiar with BTOS operations. However, the necessary information on how to install and operate the C Compiler, supplemented with references to your BTOS documentation, is included.

How to Use This Manual

If you are using the BTOS C Compiler for the first time, you should read section 1. It provides a brief overview of the product capabilities and features.

If you scan the table of contents and review the topics before you start, you may find this manual easier to use.

For definitions of terms used in this manual or related to this software, refer to the glossary.

To locate specific information, use the index.

How This Manual is Arranged

This manual contains seven sections, three appendixes, a glossary, and an index.

Conventions

The following conventions apply throughout this manual:

- The term **BTOS** refers to **BTOS II** in this manual.
- Information you enter at your keyboard appears in boldface.
- Executive commands appear in uppercase.
- When two keys are used together for an operation, their names are hyphenated. For example, **ACTION-GO** means you hold down **ACTION** and press **GO**.

Related Product Information

For information on the Operating System (BTOS), refer to the *BTOS II System Reference Manual*.

For more information about BTOS II system calls and structures, refer to the *BTOS II System Procedural Interface Reference Manual*.

For information on system status codes, refer to the *BTOS II System Status Codes Reference Manual*.

For information on Executive level commands, refer to the *BTOS II Standard Software Operations Guide*.

For information on the Editor, refer to the *BTOS II Editor Operations Guide*.

For information on BTOS Context Manager™ II, refer to the *BTOS Context/Window Manager Installation and Configuration Guide*.

In addition, the following manuals are related to the BTOS C Compiler:

- *BTOS II Language Development Programming Guide* (information and procedures for using the LINK, BIND, and LIBRARIAN commands)
- *BTOS II Debugger Programming Guide* (information and procedures for debugging programs)

Contents

About This Manual	v
Who Should Use This Manual	v
How to Use This Manual	v
How This Manual is Arranged	v
Conventions	vi
Related Product Information	vi
Section 1: Overview	1-1
Using the C Compiler	1-1
Features	1-2
Memory Requirements	1-2
Section 2: BTOS C Compiler Installation	2-1
Installing C Compiler Software	2-1
Installing C Compiler Software on a BTOS Workstation.....	2-1
Installing C Compiler Software on an XE520 Master.....	2-2
Configuring BTOS Context Manager II to Run This Application	2-3
Section 3: Using the C Compiler	3-1
Memory Utilization	3-1
Command Line Syntax	3-2
Frequently Used Command Line Options.....	3-2
Controlling Compilation Activity Options.....	3-3
Specifying Memory Model Options.....	3-3
Preprocessor Control Options.....	3-5
Disk Usage Options.....	3-5
Comment Control Option.....	3-5
Message Control Option.....	3-6
Advanced Options.....	3-6
8086 Support.....	3-6
8087 Support.....	3-7
Optimization Options.....	3-7
LINT and Warning Message Options.....	3-9
Debugging Options.....	3-13
Compatibility Options.....	3-13
Fast Calling Sequence Option.....	3-14
Segment Naming Options.....	3-16
Overlay Support Options.....	3-17
CCompiler.CFG	3-17
Linking a Program	3-19
LINT Source File Comments	3-20
/*ARGSUSED*/.....	3-21
/*LINTLIBRARY*/.....	3-21
/*NOSTRICT*/.....	3-21
/*NOTREACHED*/.....	3-22

<code>/*VARARGSn*/</code>	3-22
Compiler Operation	3-23
Temporary Files.....	3-23
Section 4: Runtime Environment	4-1
Program Execution	4-1
Memory Organization.....	4-3
Pointer Arithmetic.....	4-4
Section 5: Assembly Language Interface	5-1
External Variable Names	5-1
The C and PL/M Calling Sequences	5-2
Function Arguments.....	5-2
Calling Functions	5-4
Passing Return Values.....	5-5
Assembly Language File Structure	5-6
Defining Functions.....	5-7
Defining Data Constants.....	5-7
Global Data.....	5-8
Sample Assembly Language Modules	5-8
Small Model Version.....	5-9
Medium, Large, and Huge Model Version.....	5-9
Section 6: Library Reference	6-1
Library Overview	6-1
Runtime Support.....	6-1
I/O Operations.....	6-1
UNIX Compatible I/O.....	6-2
Standard I/O.....	6-2
BTOS I/O Services.....	6-3
Mathematical Functions	6-3
Header Files	6-4
ASSERT.H.....	6-4
CTxxx.H.....	6-4
CTYPE.H.....	6-5
ERRNO.H.....	6-5
FLOAT.H.....	6-5
I8086.H.....	6-5
LIMITS.H.....	6-6
MATH.H.....	6-6
SETJMP.H.....	6-6
SIGNAL.H.....	6-7
STDARG.H.....	6-7
STDDEF.H.....	6-7
STDLIB.H.....	6-7
STDIO.H.....	6-7
STRING.H.....	6-7
TIME.H.....	6-8
Library C Functions.....	6-8

abs	6-9
assert	6-10
atof, atoi, atol, strtod, strtol	6-11
bsearch	6-13
check8087, init8087	6-14
close	6-15
creat	6-16
ctime, localtime, asctime, gmtime	6-17
ecvt, fcvt, gcvt	6-19
exit, _exit	6-20
exp, log, log10, pow, sqrt	6-21
fclose, fflush	6-22
feof, ferror, clearerr	6-23
floor, ceil, fmod, fabs	6-24
fopen, freopen	6-25
fread, fwrite	6-27
frexp, ldexp, modf	6-28
fseek, ftell, rewind	6-29
getc, getchar, fgetc, getw	6-30
gets, fgets	6-31
index, rindex	6-32
inport, inportb	6-33
isalpha, isupper, islower, isdigit, isxdigit, isalnum, isspace, ispunct, isprint, isgraph, iscntrl, isascii	6-34
lsearch	6-36
lseek	6-37
malloc, calloc, free, cfree, realloc	6-38
memcpy, memset, memcmp, memchr	6-40
movmem	6-41
open	6-42
outport, outportb	6-43
peek, peekb	6-44
poke, pokeb	6-45
printf, fprintf, sprintf	6-46
putc, putchar, fputc, putw	6-51
puts, fputs	6-52
qsort, ssort	6-53
rand, srand	6-54
read	6-55
scanf, fscanf, sscanf	6-56
segread	6-60
setbuf, setvbuf	6-61
setjmp, longjmp	6-63
setmem	6-64
sin, cos, tan, asin, acos, atan, atan2	6-65
sinh, cosh, tanh	6-67
ssignal, gsignal	6-68
stime	6-69
strcat, strncat	6-70
strchr, strrchr, strpbrk	6-71
strcmp, strncmp	6-72

strcpy, strncpy	6-73
strlen	6-74
strspn, strcspn	6-75
strtok	6-76
swab	6-77
time	6-78
toupper, tolower, _toupper, _tolower, toascii	6-79
ungetc	6-80
unlink	6-81
vprintf, vfprintf, vsprintf	6-82
vscanf, vfscanf, vsscanf	6-83
write	6-84
Section 7: The C Programming Language	7-1
BTOS C Features	7-1
Translation Phases and Limits	7-3
Preprocessor Translations	7-3
Parser Translations	7-4
Optimizer Translations	7-4
Code Generator Translations	7-5
Compiler Limits	7-5
Preprocessing	7-5
Source File Inclusion	7-6
Define Macros	7-7
__FILE__	7-10
__LINE__	7-10
__DATE__	7-10
__TIME__	7-10
__SMALL__	7-11
__MEDIUM__	7-11
__LARGE__	7-11
__HUGE__	7-11
Conditional Compilation	7-11
Constant Expressions	7-12
Line Number Control	7-13
#pragma Directives	7-14
#error Directives	7-14
Null Directive	7-14
Comments	7-14
Lexical Conventions	7-15
Source Text Conventions	7-15
Identifiers	7-16
Keywords	7-17
Numerical Constants	7-18
Integer Constants	7-18
Character Constants	7-19
Escape Sequences	7-19
Floating Constants	7-20
Strings	7-21
Operators	7-21

Punctuation	7-22
Trigraphs	7-22
Types	7-23
Basic Types	7-23
Integral Types	7-23
Characters and Integers.....	7-23
Unsigned	7-24
Floating Types	7-24
Void Type	7-24
Enumerated Type	7-24
Composite Types.....	7-25
Pointers	7-25
Functions	7-25
Arrays	7-26
Structures and Unions	7-26
Bitfields	7-27
Type Modifiers.....	7-27
Declarations	7-28
Storage Class Specifiers.....	7-30
Auto.....	7-30
Extern	7-31
Register	7-31
Static.....	7-32
Typedef.....	7-32
Type Specifiers	7-32
Basic Arithmetic Type Specifiers.....	7-32
Structures and Unions	7-33
Enumerations.....	7-34
Void.....	7-35
Declarators.....	7-35
Pointer	7-36
Function	7-37
Array	7-38
Bitfields	7-39
Type names	7-40
Type Equivalence	7-41
Initialization	7-41
Scope of Identifiers.....	7-43
Linkage.....	7-44
Expressions	7-44
Lvalues	7-45
Primary Expressions	7-45
Postfix Operators.....	7-46
Postfix Increment and Decrement Operators	7-46
Function Calls	7-46
PL/M Functions.....	7-49
Interrupt Functions.....	7-50
Array Subscripts	7-51
Member Access Operations.....	7-51

Unary Operators	7-52
Prefix Increment and Decrement Operators	7-52
Address and Indirection	7-52
Unary Arithmetic Operators.....	7-53
Casts.....	7-54
Binary Operators.....	7-55
Normal Arithmetic Operators.....	7-56
Shift Operators	7-57
Relational Operators.....	7-57
Bitwise Boolean Operators.....	7-58
Logical Operators	7-58
Conditional Expressions	7-58
Simple Assignment	7-58
Compound Assignment.....	7-59
Comma Operator.....	7-60
Constant Expressions	7-60
Conversions	7-60
Integral Widening Conversions	7-61
Usual Arithmetic Conversions	7-61
Other Types	7-62
Statements	7-62
Labeled Statements	7-62
Blocks.....	7-63
Expression Statement.....	7-63
Null Statement	7-63
Selection Statements	7-63
Iteration Statements.....	7-64
Jump Statements	7-65
Inline Assembly Statements.....	7-66
Using Inline Assembly Language	7-67
Instruction Mnemonics.....	7-67
Inline Assembler References to Data and Functions.....	7-69
C Structure Members	7-71
Jump Instructions and Labels	7-72
Comments on Inline Assembler Statements.....	7-73
External Definitions	7-73
Function Definitions	7-73
Data Definitions.....	7-74
Portability Considerations	7-75
Appendix A: Diagnostic Messages	A-1
Fatal Messages	A-2
Error Messages	A-5
Warning Messages	A-19
Appendix B: Command Line Options Summary	B-1

Appendix C: Library Summary	C-1
Input-Output Functions	C-1
Standard I/O	C-1
Block I/O	C-3
File Management	C-3
String Handling	C-3
Memory Management	C-6
Miscellaneous Arithmetic	C-6
Searching and Sorting	C-7
Program Control	C-7
Date and Time Management	C-8
Hardware Functions	C-8
Mathematical Library	C-9
Glossary	Glossary-1
Index	Index-1

Figures

4-1	Small Model Segments	4-3
4-2	Medium Model Segments.....	4-4
4-3	Large Model Segments	4-5
4-4	Huge Model Segments	4-6

Tables

2-1	Application Information for Context Manager	2-3
3-1	Memory Models and Link Files	3-19

Overview

This section gives you an overview of the Unisys BTOS C Compiler, with a look at capabilities, operations, and features.

The BTOS C Compiler is capable of generating programs with effectively unlimited amounts of instruction code and data. A compilation switch that controls the amount of code and data allowed in a program is available. The compiler allows you to compile separate source files, that you can combine to produce an executable program with the BTOS Linker.

C, a robust and portable programming language, was developed for a UNIX operating system in the early 1970s by Dennis Ritchie at Bell Laboratories. The flexibility and efficient executing speed of C lend themselves to structured programming techniques.

Using the C Compiler

The C Compiler runs on a BTOS workstation (B26, B27, B28, B38, or B39) executing BTOS II 1.0 or higher. You can also install it on an XE520 master system and execute it on any BTOS cluster workstation.

You need to have the language development software on your system. Specifically you must have the following:

- BIND command
- ASSEMBLE command (for use of the inline assembler option)
- Temporary disk storage for intermediate results during each compile

The temporary disk storage should be approximately twice the size of the source file you compile when you generate object files directly, or three to four times its size when you generate assembly language output.

You install the software from the three installation diskettes (refer to section 2). Other files on the diskettes include the complete source code for the runtime library and the software installation submit files.

Features

The C Compiler provides you with the following features:

- Four memory models: small, medium, large, and huge
Huge models allow applications to exceed the normal limitation of 64 Kb for data (globals, statics, and stack area).
- Mixed-model programming
Mixed-model programming allows use of far data pointers while compiling modules in the medium model.
- Numerous compile-time switches
These are provided to control compilation and code generation.
- Built-in LINT facility
This can be used for examining a collection of source files for bugs and obscurities.
- BTOS II compatibility
- 8087 emulation support and/or use of a floating point coprocessor
- UNIX-like support of input/output redirection and pipes.

Memory Requirements

BTOS C requires a minimum memory size of 250 Kb to compile source programs of moderate size.

BTOS C Compiler Installation

You can use the procedures in this section to install your BTOS C Compiler software. After you install the software, you enter the CCOMPILER command at the Executive level to run the compiler.

Installing C Compiler Software

You install the C Compiler software from three 5-1/4-inch software diskettes, B25CE1-1, B25CE1-2, and B25CE1-3. The diskettes are write-protected.

Note: Because C chains to other run files, all C files reside on the [Sys]<Sys> and [Sys]<BtosC> directories. No directory specification is allowed on the Software Installation command form.

To install the C Compiler software on your system, you must have BTOS II 1.0 or higher installed. To use the BTOS C Compiler, you must have approximately 2200 sectors available.

Installing C Compiler Software on a BTOS Workstation

To install the C Compiler software on a BTOS workstation, use the following procedure:

- 1** Disable the cluster if the system is clustered (use the Executive DISABLE CLUSTER command or power down the other cluster units).
- 2** Insert the software diskette into the floppy drive [f0].
- 3** Enter **SOFTWARE INSTALLATION** at the Executive command line.
- 4** Press **GO**.

- 5 Follow the instructions displayed.

When the system finishes software installation, the highlighted message **INSTALLATION OF BTOS C COMPILER COMPLETE** appears, followed by an Executive command prompt.

- 6 Remove the software diskette.

If your workstation is clustered, you can resume cluster operations using the **RESUME CLUSTER** command.

The **CCOMPILER** command is now available at the Executive level; you can use the command to compile C programs.

Installing C Compiler Software on an XE520 Master

To install the C Compiler software on an XE520 master system, use the following procedure:

- 1 Boot the cluster workstation you want to use for software installation on the XE520.
- 2 Power down all other cluster workstations.
- 3 Insert the diskette into the floppy drive [f0] on your workstation.
- 4 Enter **XESOFTWARE INSTALLATION** at the Executive command line.
- 5 Press **GO**.
- 6 Follow the instructions displayed.

Configuring BTOS Context Manager II to Run This Application

BTOS Context Manager II is BTOS software that lets you run several applications simultaneously so that you can rapidly switch back and forth between them. When configuring (setting up) Context Manager, you provide information about the applications you wish to run.

Table 2-1 contains information you may need when configuring Context Manager for this application. For an explanation of how to use this information, refer to the *BTOS Context/Window Manager Installation and Configuration Guide*.

Table 2-1 Application Information for Context Manager

Run file name	exec.run
Run file version	6
Runs in Protected Mode	yes
Minimum memory required	250 KB
Maximum memory required	250 KB
Able to be swapped	yes
Needs Executive screen	no
Loads own font	no
Loads own keyboard translation table	no
Directly manipulates the video	no

Special Notes: Command case should be 'CM'.

Using the C Compiler

The C Compiler software package allows you to compile separate source files that can be combined to produce executable programs using the BTOS Linker. The compiler can generate files to assemble using the BTOS Assembler or produce standard object files.

This section provides you with procedures and information that you need to use the C Compiler. For more information about the Linker, refer to the *BTOS II Language Development Programming Guide*.

Memory Utilization

The compiler is capable of generating programs with effectively unlimited amounts of instruction code and data. A compilation switch controls the amount of code and data allowed in a program.

A total of four memory models are supported. Four sets of libraries are supplied, one set for each memory model. The memory models are:

	Maximum Code		Maximum Data
Small	64 Kb	+	64 Kb
Medium	1 Mb	+	64 Kb
Large	1 Mb	+	1 Mb (64 Kb of globals/statics and stack)
Huge	1 Mb	+	1 Mb (64 Kb of globals/statics per source file plus 64 Kb stack)

The smaller models use memory more efficiently and program execution may be somewhat faster. The larger models give the programmer complete flexibility for constructing large applications.

Command Line Syntax

The CCOMPILER command invokes each pass of the compiler with appropriate options. The **[options]** line consists of a series of options (each preceded by '-'), and the **Filename** line specifies one source file to compile. If a filename is provided with no extension, a .C extension is supplied. If some other extension is given, the file is treated as if it had a .C extension.

To compile your C program, use the following procedure:

- 1 At the Executive level, enter **CCompiler**. The following form appears:

```
Command: CCOMPILER
  [options]
  Filename
```

- 2 Enter the options you need in the **Options** field.

If you supply more than one option, separate each by a space. You can place options in any order, and you can include any number of them as long as there is room in the command line.

- 3 Enter the name of the C source file that you want to compile in the **filename** field.

- 4 Press **Go**.

The compiler creates an output file using the source filename with an **.obj** extension for object files or an **.asm** extension for assembly language files.

Normally, compiler messages are written to the screen. To direct these compiler messages to a file other than **STDOUT**, specify redirection on the options line. For example:

```
>[splb]
>output.err
```

Frequently Used Command Line Options

The most frequently used compiler options are described in this subsection. The options are organized by topic. For a complete alphabetical list of the compiler options, refer to appendix B.

Controlling Compilation Activity Options

- S This option is required if you have inline assembly statements in your source file. When present, the named source file is compiled producing an assembly language output file. This .asm file is not assembled, but requires you to use the ASSEMBLE command to produce an object file for linking. This option is also useful when you wish to see the assembly language output for your C program.
- k If the compiler detects any errors it will exit with an error code of 1. For compilations invoked by a submit file, this error code will cause the submit file processing to stop. By default, the compiler always exits with an error code of 0, allowing submit file processing to continue.

Specifying Memory Model Options

BTOS C provides four memory models to choose from: small, medium, large, and huge. All of the source files for a program must be compiled with the same memory model option.

The memory model selected determines the size of pointers in memory and as a result determines the amount of memory a program can use. A program should use the smallest memory model that the program fits in, since the smaller memory models are much more efficient than larger ones, both in execution speed and in memory requirements.

Mixed model programming allows programmers to gain finer control over the manipulation of pointers. Because of the use of special keywords, however, mixed model programs are not directly portable to other environments, so this technique should be used with caution. Used properly, mixed model programming can provide high performance pointer manipulation even in the larger memory models.

These compiler options specify, for the large and huge memory models (and for far pointers in any memory model), which of two forms of pointer arithmetic is to be used. Files compiled with the same memory model, but different pointer arithmetic types, can be used in the same program. Refer to section 4 for more information about the runtime environment.

- mh** This option causes the compiler to produce huge memory model output code. The 20-bit pointer arithmetic is performed using subroutines.
- mhf** This option causes the compiler to produce huge memory model output code. The 16-bit pointer arithmetic is performed using inline instructions.
- ml** This option causes the compiler to produce large memory model output code. The 20-bit pointer arithmetic is performed using subroutines.
- mlf** This option causes the compiler to produce large memory model output code. The 16-bit pointer arithmetic is performed using inline instructions.
- mm** This option causes the compiler to produce medium memory model output code. Far pointers use full 20-bit pointer arithmetic if they appear in the files being compiled.
- mmf** This option causes the compiler to produce medium memory model output code. Far pointers use 16-bit pointer arithmetic if they appear in the files being compiled.
- ms** This option causes the compiler to produce small memory model output code. Far pointers use full 20-bit pointer arithmetic if they appear in the files being compiled.
- msf** This option causes the compiler to produce small memory model output code. Far pointers use 16-bit pointer arithmetic if they appear in the files being compiled.

Preprocessor Control Options

These options control the specification of preprocessor #define macros and the specification of search directories for #include files. These options may be used as many times as required.

- Didentifier Defines the named identifier to the string consisting of the single character '1'
- Diden=string Defines the named identifier iden to the string after the equal sign. The string cannot contain any spaces or tabs.
- Idirectory The indicated directory is searched for #include files in addition to the current directory.
- Uidentifier Undefines any previous definitions of the named identifier.

Disk Usage Options

These options change the directory used for temporary and output files.

- n1path Places any \$.CC files in the directory named by path.
- n2path Places any other temporary files in the directory named by path.
- nopath Places any .obj or .asm files in the directory named by path.

Comment Control Option

This option provides for the use of nested comments in the source file.

- C If present, nested comments are allowed. Comments cannot normally be nested.

Message Control Option

This option controls the output of warning messages.

-w If present, no warning messages are printed.

Advanced Options

The compiler normally generates code for an 8086/8088 with no 8087 coprocessor. Code generated in this mode is the most portable to the full range of microprocessors. This default mode does not make maximum use of the capabilities of the more advanced processors.

8086 Support

For the 8086 family of processors, word-sized data items stored at even addresses are more efficiently fetched or stored than word-sized items at odd addresses. By default, the compiler does not align data objects. The compiler can be directed to align word-sized items on even addresses. When not aligning, some care is needed to make sure that data references do not become confused.

If a structure is used in more than one source file of a program, all source files referencing that structure should be compiled with the same alignment setting. The libraries distributed with BTOS C can be used whether or not alignment is selected. Using alignment consumes slightly more storage, especially for structures containing both char and non-char members.

-a Align integer size items on a word boundary. Extra bytes are inserted in a structure to ensure alignment of fields. Automatic and global variables are aligned properly.

-l Generate 80186 "immediate" instructions for pushing and multiplying by constants. Generate ENTER and LEAVE instructions when appropriate.

8087 Support

By default, a BTOS C compiled program uses emulation routines to perform floating point arithmetic. These routines can take advantage of an 80x87 family Numeric Co-Processor (NCP) installed in the workstation when the program is run, even if the workstation the program was compiled on did not have an NCP.

For programs that use the math library, you must rebuild the library with the `-f` option if you wish to use inline floating point instructions.

- `-f` Floating point operations are generated using 80x87 inline instructions rather than calls to runtime emulation routines.
- If this option is selected, all files in a program must be compiled to use inline 80x87 instructions. The 80x87 instructions use the NCP to return floating point values from functions, rather than using 8086 registers in the default code generation.
- Programs compiled with this option should call `Check8087()` and check its return value, the `_8087` flag. If the flag is zero, the program should terminate.

The supplied C libraries have been compiled using the emulation routines. The emulation routines include the ability to exploit an 8087, 80287, or 80387 NCP if one is present. Calling the library function `Check8087()` from your program detects the presence or absence of a NCP and sets the `_8087` flag. The library floating point routines check this variable and use 80x87 instructions if the NCP is present.

Optimization Options

There are three separate optimization switches with BTOS C: `-O`, `-G`, and `-Z`. The first two switches, `-O` and `-G`, are always safe to apply. The `-O` option adds an extra pass to the compilation that eliminates redundant jump instructions and reorganizes loop and switch statements, causing a reduction in code size from a minimum of two to a maximum of fifteen percent. The loop reorganizations can

speed up tight inner loops by as much as ten percent, even though the space savings are not that great. The `-G` option controls the tradeoff decisions between consuming more memory with faster instructions.

The third optimization switch, the `-Z` option, causes the code generator to "remember" the contents of registers and use them if possible. This optimization is not altogether risk-free.

In the following example code, the variable `A` is loaded into register `AX`. If `A` were to be assigned a new value, the value of `AX` would be reset to indicate that its contents are no longer current. Unfortunately, the value of `A` is modified indirectly by assigning through a pointer and the compiler erroneously believes that `AX` still contains the correct value of `A`.

C Code	Optimized Assembler
<code>func()</code>	
<code>{</code>	
<code>int A, *P, B;</code>	
<code>A = 4;</code>	<code>MOV A, 4</code>
<code>...</code>	
<code>B = A;</code>	<code>MOV AX, A</code>
	<code>MOV B, AX</code>
<code>P = &A;</code>	<code>LEA BX, A</code>
	<code>MOV P, BX</code>
<code>*p = B + 5;</code>	<code>MOV DX, AX</code>
	<code>ADD DX, 5</code>
	<code>MOV [BX], DX</code>
<code>printf("%d\n", A);</code>	<code>PUSH AX</code>
<code>}</code>	

The call to `printf` is not correct. The compiler sees that `AX` contains the value of `A` and so pushes the contents of the register rather than the contents of the memory location. `Printf` will display a value of 4 rather than 9 because the indirect assignment through `P` has hidden the change to `A`. If the assignment had been written as `A = B + 5`, the compiler would recognize the change in value.

The contents of registers are "forgotten" whenever a function call is made or when a destination for a jump occurs, such as a label, a case statement, or the beginning

or end of a loop. Because of these limits and the small number of general purpose registers in the 8086 family of processors, most programs will optimize correctly.

- G Optimizes for speed over code size. This optimization mostly affects the instructions used to clean up the arguments after a function call.
- O When present, the jump optimizer pass (CC2.run) is executed to optimize the compiled C source file.
- Z Results in extra optimization to suppress redundant load operations. There are circumstances which can cause the optimized code to work incorrectly. The optimization is designed to suppress register loads when the value being loaded is already in a register. This can eliminate whole instructions and also convert instructions referring to memory locations to use registers instead.

LINT and Warning Message Options

Checking the declarations of functions and variables across multiple source files is done by the compiler. The BTOS C Compiler supplies an additional mode of operation (enabled with the `-L` and `-Lxxx` options) to perform those checks. This extra mode provides the full range of diagnostic checking found in LINT under UNIX.

If you had two source files (`afile.c` and `bfile.c`) that you wished to cross-check, you might use the following commands:

```
Command: CCOMPILER  
[options] -L afile.lnt  
Filename afile.c
```

```
Command: CCOMPILER  
[options] -L bfile.lnt  
Filename bfile.c
```

```
Command: CCOMPILER  
[options] -L  
Filename afile.lnt bfile.lnt [sys]<BtosC>clib.lnt
```

These commands compile both of the source files, then cross-check the call and declaration information, including any references to the Runtime Libraries. These references are defined in the compiler file `clib.lnt`.

If you have a library of commonly used routines that are debugged, so that including them in the LINT execution would be unnecessary, you can still check any code that uses the library. First, you need to prepare a LINT file that describes the library. Then you can use this LINT file in subsequent LINT executions to check programs that use the library.

LINT files are created using the `-L` option and a filename, which should include the `.LNT` extension. As each file is compiled, data about functions, calls, and global variables is appended to the LINT file, or replaced in the LINT file if it has already been placed there. The LINT files supplied with the compiler provide the definitions of the C library functions (`CLIB.LNT`) and the CTOS/BTOS interface procedures (`CTOS.LNT`). Similar LINT files can be built for user constructed libraries, which can then be used with future compilations.

The following options control the level of messages issued by the compiler. The compiler provides many warnings to aid the programmer in writing and debugging code. You can disable some or all of these warnings with these options.

- `-E` Allows more elastic type conversions in function arguments. Normally, types must match exactly in calls and function definitions. If this option is used, signed and unsigned integers of equal width are considered compatible, as are pointers to different types. Normally, these combinations are considered as distinct types and will cause error messages to be displayed.
- `-L` If present, the compiler performs a LINT compile for the named `.LNT` files. Note that this is the only time that more than one filename can be specified.

-
- Lfilename** Directs the compiler to perform a LINT compile for the named C source file. The call, function definition, and variable declaration information are written to the named filename. Only one **-L** option is allowed per command line. The named file should be written with the **.LNT** extension. This output file can be included in future LINT compiles when cross-checking files. Multiple **.LNT** files may be combined into one file by using the BTOS APPEND command.
- Q** If used with the **-L** option, definition information only is output to the LINT file.
- T** Check all casts for suspicious conversions. Normally, casts are not checked and no warnings are printed for them. Conversions caused automatically, such as when assigning between variables of different types, are always checked. In particular, converting a pointer to a different kind of pointer and converting a long to an integer type produce warnings.
- b** Suppress warnings about unreachable break statements.
- d** Suppress warnings about loss of significant digits in the conversion of longs to int.
- h** Bypass heuristic tests that attempt to report possible bugs, faulty style, or wasteful constructs.
- q** Suppress warnings about undefined external symbols in executing LINT.

- s Warn about any structure that is being passed by value rather than by address. This option allows programmers who wish to enforce obsolete structure usage to detect instances of inadvertently passing structures by value.
- x Report variables declared as external but never used.

The following options suppress all occurrences of the listed warning message.

- wamb Ambiguous operators need parentheses.
- wamp Superfluous & with function or array.
- wapt Non-portable pointer assignment.
- wasm Unknown assembler instruction.
- waus 'XXXXXXXX' is assigned a value which is never used.
- wcln Constant is long.
- wcpt Non-portable pointer comparison.
- wdef Possible use of 'XXXXXXXX' before definition.
- wdgn Degenerate constant expression.
- wdup Duplicate definition of 'XXXXXXXX'.
- weff Code has no effect.
- wfun Function 'XXXXXXXX' unused.
- wign 'XXXXXXXX' return value ignored.
- wpar Parameter 'XXXXXXXX' is never used.
- wpia Possibly incorrect assignment.
- wrch Unreachable code.
- wret Both return and return of a value used.
- wrpt Non-portable return type conversion.
- wrvl Function should return a value.
- wsig Conversion can lose significant digits.
- wstr 'XXXXXXXX' not part of structure.
- wstu Undefined structure 'XXXXXXXX'.
- wstv Structure passed by value.
- wsus Suspicious pointer conversion.
- wuse 'XXXXXXXX' declared but never used.
- wvoi Void functions cannot return a value.
- wzst Zero length structure.

The -w option (without any subsequent characters) suppresses all warning messages.

Debugging Options

These options are useful when debugging programs. Stack overflow checking requires added space and increases execution time in a program. But when the stack does overflow, it can be a difficult bug to discover. Generation of standard stack frames is useful when using the debugger to trace back through the stack of subroutine calls.

- N Generate stack overflow logic at the entry of each function. If an overflow is detected, the program exits with an error code of 400 (insufficient memory).
- Y Generate standard function entry and exit code. Normally, for maximum efficiency, the compiler minimizes the amount of information saved on entry to a function. This practice can prevent the BTOS debugger from displaying a complete stack trace. To correctly display the stack trace, use this flag.

Compatibility Options

These options are designed to enhance portability.

- A If present, any of the BTOS extension keywords are ignored and can be used as normal identifiers. These keywords include `near`, `far`, `asm`, `plm`, `interrupt`, `_es`, `_ds`, `_cs`, `_ss`, and `_ES`.
The `-A` option is designed to provide a maximally portable 'ANSI' environment where none of the BTOS C extensions are usable. This also means that application programs written using the BTOS C extended keywords as normal identifiers can be compiled by using this switch.
- K Treat all char declarations as if they were unsigned char type.

- i#** If present, this option specifies the number of significant characters in an identifier. All identifiers, whether variables, preprocessor macro names, or structure member names are treated as distinct only if the first # characters are unique. The number given can be any value from 1 to 32. The default number is 32.
- Other systems may ignore characters beyond the first 6 or 8. If you are porting to these other environments, this option be used to see if there are any name conflicts in long identifiers when they are truncated.
- r** If present, this option suppresses the use of register variables and all register keywords will be ignored.
- If you link with Assembly Language code that does not preserve the values of the SI and DI registers, the **-r** option allows you to call that code from BTOS C. Suppressing register variables does reduce the efficiency of generated code, in general, but can be necessary to use existing subroutines. Note that functions in a source file compiled with **-r** can call code in a source file compiled without **-r**, such as a routine in the Runtime Libraries. The opposite is not true, so functions in a file compiled with **-r** can only be called from a file also compiled with **-r**.

Fast Calling Sequence Option

This option, **-p**, specifies the PL/M calling sequence for a the functions in the source file. Functions explicitly declared to use a variable number of arguments, such as `printf`, are clearly exempted.

The advantage of the PL/M calling sequence is smaller and faster function calls. The major disadvantage of this calling sequence is greater sensitivity to error. With the normal calling sequence, function calls can omit any unused trailing arguments without causing adverse effects, whereas in the PL/M calling sequence all arguments must be specified.

PL/M handles function arguments by pushing the arguments in the reverse order of the C argument sequence. In C the calling code pops the arguments; in PL/M the called routine uses a special return instruction that returns and pops the arguments. If a function is called several times, this means only one set of pop code in PL/M while several such sequences are needed in C.

For programs that do not involve many calls to functions like `printf`, a significant space savings can be gained by using the PL/M calling sequence.

Pointers in C are returned via the same registers as integers (or long integers depending on the memory model), while PL/M uses different registers. Thus, the manner in which some C programs tend to move pointers to and from integers do not work in the PL/M sequence.

Programs using the PL/M calling sequence must be coded carefully. LINT can be helpful by identifying all the circumstances where the PL/M calling sequence might cause a problem. In particular, any function defined to return a pointer must be explicitly declared everywhere it is called.

The use of function prototypes can help as well.

Since the supplied libraries of BTOS C are built with the C calling sequence, and since recompiling the libraries would not help `printf` and `scanf`, any function defined and declared with a prototype containing an ellipsis (...) will be compiled using the C calling sequence. This technique allows mixing the calling sequences in a single program.

STDIO.H is written with the `printf` and `scanf` functions declared as accepting a variable number of arguments, so that including STDIO.H allows a program compiled using the PL/M calling sequence to still use `printf` and `scanf`. To build a program using the PL/M calling sequence, the runtime libraries must be rebuilt using the `-p` option.

Register variables are not used in declared plm functions, but for functions compiled with the `-p` option they are used. Remember that BTOS functions must be declared as plm functions, even if you are using the `-p` option.

`-p` Generate all subroutine calls and all functions using the PL/M-86 calling sequence.

Segment Naming Options

For some complex applications, the ability to organize pieces of code or data into specific segments is desirable. For this reason, BTOS C provides flexible segment naming options. Each C output file defines three segments. The code is placed in the first segment, the initialized static and global data is placed in the second segment, and the uninitialized static and global data is placed in the third. In the huge model, the second and third segments are merged into a single data segment.

Groups are generated depending on the memory models and the segment naming options, which are listed and described below. The code segment of a source file is not given a group association unless the `-zP` option is used. The data segments for all memory models are placed in `DGROUP`.

- `-g` Make all segments "public". This is most useful when using the huge memory model and it is desired that segments from different modules be combined and accessed with one selector.

- `-j` Causes all data in the module to be placed into the code segment. This option should not be used with the small and medium memory modules.

- `-zAname` Changes the name of the code segment class to 'name'. By default, the code segment is assigned to class 'CODE'.

- `-zBname` Changes the name of the data segments class to 'name'. By default, the data segment is assigned to class 'STACK'.

- `-zCname` Changes the name of the code segment to 'name'. By default, the code segment is named 'CODE', except for the medium and large models, where the name is 'C_filename', where filename is the source filename.

- `-zDname` Changes the name of the uninitialized data segment to 'name'. The uninitialized data segment is named 'BSS'.

-
- zGname** Changes the name of the data group to 'name'. By default, the data group is named 'DGROUP'.
 - zPname** Causes output files to be generated with a code group, for the code segment named 'name'.
 - zRname** Changes the name of the initialized data segment to 'name'. By default, the initialized data segment is named 'DATA'.
 - zSname** Changes the name of the initialized data group to 'name'. By default, the initialized data group is named 'DGROUP'.
 - zTname** Changes the name of the initialized data class to 'name'. By default, the initialized data class is named 'STACK'.

Overlay Support Options

The BTOS Linker and Overlay Manager require all modules that are linked to use overlays to conform to rigid conventions. If your application will use overlays, specify *all* of the following options:

- t** Causes all functions to be PUBLIC FAR procedures, including any static functions.
- r** No register variables.
- Y** Force generation of standard function entrance code.
- mlf** Large memory model; this is the only memory model supported by BTOS C for overlays.
- S** Assembly code output. Use the BTOS Assembler to produce object files.

CCompiler.CFG

To specify many command line options for a single compile, a special file named CCOMPILER.CFG can be used. This file is a simple text file which may have several compile options per line, with as many lines as desired.

The compiler reads this file first and, once all of these options have been processed, reads the options given on the command line. An option like `-w` or `-f`, with no associated string, acts as a toggle. If the option is given once, it is turned on. If the option appears more than once, it is toggled once for each appearance. So, if an option appears in the `CCOMPILER.CFG` file, it can be suppressed by including it again on the command line. Define macros supplied in the `CCOMPILER.CFG` file can be undefined with the `-U` option.

Comments can be placed in the `CCOMPILER.CFG` file by placing a semicolon (`;`) on any line. Any text appearing after the semicolon up to the end of the line is considered a comment. Completely blank lines are allowed and any amount of white space may appear before or after an option string.

The `CCOMPILER.CFG` file can be located in the current directory or in the `[sys]<sys>` directory. By appropriately creating `CCOMPILER.CFG` files in each directory, options specific to a particular component of a large system under development can be repeatedly used.

Examples:

```
-I [sys]<BtosC>      ; Preprocessor options
-Y    -r            ; Debugging options
-O                ; Use the optimizer
-l -a -f          ; 80186, with 8087 instructions
```

The `-I` option is used to tell the compiler where the standard include files (like `stdio.h`) are located. The `-r` option causes standard function entry and exit code to be generated for debugging purposes.

The `-O` option invokes the optimizer on all compiles. The `-l` and `-a` options are used for a 80186. Code executes more quickly on an 80186 if word sized variables are aligned at even memory addresses. The `-f` option causes 8087 inline floating point instructions to be generated.

Linking a Program

To produce a complete, executable program, the object files produced by the compiler and assembler must be combined with the C runtime library using the BTOS Linker and the BIND command.

C programs are often divided into multiple source files. Functions in one source file may refer to data and functions in another file. Dividing source code like this is particularly valuable in large programs. Even small programs contained entirely in a single source file may refer to functions in the C library.

The function `main` is the starting point of any C program and it expects two arguments (`argc`, `argv`). A small amount of code must be executed to initialize these arguments before calling `main`. Linking is used to merge in that initialization code, which is contained in the object files `C0x.obj`, where `x` specifies the appropriate memory model. `C0s.obj` must be listed first on the object modules line of the BIND command. The corresponding library files must be specified in the [Libraries] line in order to incorporate the library routines for the appropriate memory model (refer to table 3-1).

Table 3-1 Memory Models and Link Files

Model	C0x.obj in Object Module Field	[Libraries]
Small	COS.OBJ	CLIBS.LIB, MATHS.LIB
Medium	COM.OBJ	CLIBM.LIB, MATHM.LIB
Large	COL.OBJ	CLIBL.LIB, MATHL.LIB
Huge	COH.OBJ	CLIBH.LIB, MATHH.LIB

To link a program using the BTOS Linker, use the following command:

Command: BIND

Object modules	C0x.obj object_files
Run file	run_file
[Map file]	
[Publics?]	
[Line Numbers?]	
[Stack size]	stack_size (default is 8096)
[Max array, data, code]	
[Min array, data, code]	
[Protected capability]	
[Version]	
[Libraries]	libraries
[DS allocation?]	
[Symbol file]	

This command executes the Linker to combine the named object modules and produce the run_file.

A stack size must be given if your program makes extensive use of the stack or the heap in the small and medium memory models. The default stack size is 8096 bytes. In the small and medium models, calls to the malloc family of functions use part of the stack space for the memory heap. You should include any space needed for these calls in the stack size figure you use. For the large and huge memory models, the stack size value is used exclusively for stack space. malloc in the large and huge models uses short-lived memory.

LINT Source File Comments

Command line arguments are the usual means of controlling the operation of the compiler. Certain information about the source file cannot be conveyed by setting a single command line flag, so a convention has been adopted of defining special LINT comments which are interpreted by the compiler. These comments must be in uppercase letters. No characters are allowed between the start of the comment (/*) and the keyword of the comment. Any characters can be included in the LINT comment after

the keyword as long as the keyword is separated from the additional text by white space.

/*ARGSUSED*/

This comment is placed before any function definition where some of the function parameters declared are not used. This suppresses the compiler warnings about unused function parameters.

This is most often used when there are stub test routines included for unfinished parts of a program. It can be useful to force the calls to the unfinished functions to pass the correct parameters while not actually placing any code in them.

/*LINTLIBRARY*/

This comment is placed at the head of a file of declarations of library functions. LINT normally issues warnings about any functions in a program that are not called. This is because such functions are wasting space in the program. Library functions are not included by the linker, and therefore no space is wasted by unused library functions. LINT must be informed about which functions are in a library so that unnecessary warnings are suppressed.

Once a library is built, rather than compile all the functions in it and produce a large LINT file with many calls that need to be checked every time LINT is called, a special file of just the function declarations (not the code in the functions) is prepared. For assembly language subroutines this is necessary, since the compiler cannot automatically determine what the function parameters are. These source files are then compiled and compact LINT files are produced with just the function interfaces included. This is precisely the method used to create CLIB.LNT.

/*NOSTRICT*/

This comment is placed just ahead of a statement and suppresses the strict type checking for that statement alone. It has the same effect as the `-h` command line option but only on one statement.

/*NOTREACHED*/

This comment is placed in a function to notify the compiler that the current point in the code is never reached in executing the function. This affects the unreachable code warning message.

This comment is usually placed after a call to a function like `exit`, which never returns, or after a looping construct, which never falls through (the compiler tries to recognize such loops but cannot recognize all of them).

The compiler claims there is a return with no value at the end of a function where the end is reachable. If the function returns a value explicitly elsewhere or is declared to return some non-integer type, a warning message is given at the end of the function. The `NOTREACHED` comment is placed just ahead of the end of the function to suppress this message. The comment also serves to document functions which do not return, or unending loops.

/*VARARGSn*/

C allows functions to accept a variable number of parameter arguments. `Printf` and `scanf` are the prime examples. The ellipsis (...) notation is used to declare `printf` and `scanf`. The `VARARGS` comment notifies LINT that the following function accepts a variable number of arguments, even though it does not use the ellipsis notation. If a number immediately follows with no intervening white space, the number gives the minimum number of fixed arguments that must be present in the call. If no number is given then zero is assumed.

Functions such as `printf` that accept a variable number of arguments normally have a few required arguments at the beginning of the parameter list. LINT checks the required arguments in each call for type compatibility and ignores any excess. Without the `VARARGS` comment, LINT prints error diagnostics for each call which does not have the exact number and type of arguments declared in the function.

Compiler Operation

BTOS C compiles a source file by running the compiler passes, CC0.RUN, CC1.RUN, CC2.RUN, and CC3.RUN, in that order. The options given to each pass are the same as the options specified in the CCOMPILER command.

The CC2.RUN pass is the optional optimization pass invoked by the `-O` command line option.

The CC4.RUN pass does LINT cross-checking and is executed when the `-L` option, with no trailing filename, is used.

Except for CC4.RUN, each pass of the compiler accepts only one source file name, and wildcards are not allowed. The CC4.RUN pass accepts multiple LINT filenames, and wildcards are allowed.

All passes of the compiler normally return zero exit codes, permitting submit files to continue. If the `-k` option is used, any pass that detects an error will exit with an error code of 1, thereby stopping the submit file processing.

Temporary Files

The individual passes produce the following temporary files:

Pass	Temporary Files
CC0.RUN	srcfil.\$CC
CC1.RUN	srcfil.\$CD srcfil.\$CF srcfil.\$CG srcfil.\$CI srcfil.\$CS
CC2.RUN	srcfil.\$CX

These temporary files are created in the current directory, unless directed otherwise with a `-n1path` or `-n2path` option. The temporary files are normally deleted when the pass that reads them is finished with them.

Runtime Environment

Program Execution

The standard start-up code supplied with the BTOS C Compiler performs the following steps whenever a C program starts executing:

- The DS register is set to point to the Data segment.
- The SS register is set to point to the Stack segment.
- The SP register is set to point to the top of the Stack.
- The program parameters are copied into the program data. These parameters are then pointed to by the argv array and argc is set. Argv[0] is set to the command name.
- The files stdin, stdout, and stderr are opened. Any redirection and piping options are recognized. By default, the standard I/O files are opened so that stdin is set to the file [kbd], stdout is set to the file [vid], and stderr is also set to [vid]. If, in any of the command parameters, a parameter beginning with a left or right angle-bracket ('<' or '>') is encountered, that parameter is treated as a file redirection. The parameter is not placed in the argv array.
- A redirection parameter beginning with a '<' character provides an alternate filename for the stdin file. The parameter string (excluding the leading '<' character) is treated as a filename and opened for input. A redirection parameter beginning with a '>' character provides an alternate filename for the stdout file. The parameter (excluding the leading '>' character) is treated as a filename and opened for output. The file is created if it does not exist, and is truncated to zero length if it does exist. If the parameter begins with two '>' characters, both are ignored to find the filename, and the output of the program is appended to the file. In this second case, the file is also created if it does not exist.
- The function main is called.

After main returns, if piping was not specified, exit is called with an error code of zero. This is not in accordance with the latest draft of the ANSI C Standard, which specifies that exit should be called with the return value from main.

If piping was specified, the next program is activated with the specified arguments.

For example, the command (assuming XYZ is a C program):

```
Command  XYZ
params   file1 file2
```

Main is called with a value of 3 for argc, and argv[0] is set to **XYZ**, argv[1] is **file1** and argv[2] is **file2**. If main returns, then exit is called with a zero argument.

If the RUN command is used to start a C program, argv[0] is set to the **Command** entry, and the following argv[] entries are set to the **Parameter** entries. The argv array does not preserve the multiline structure of the original command entries. All subparameters are strung together as if they were all entered as one long parameter.

Emulation of UNIX pipes is performed by passing the stdout output of a program to the stdin input of the next program when the | symbol is processed.

For example, the command (assuming XYZ and ABC are C programs):

```
Command  XYZ
params   p1 p2 | ABC p3 p4
```

XYZ is run with parameters p1 and p2, and its stdout output is placed into [SYS]<\$>pipe.file. After XYZ main returns, ABC is run with parameters p3 and p4, and its stdin comes from [SYS]<\$>pipe.file. Note that ABC.run must be in the current directory or in [SYS]<SYS>; otherwise, the directory must be specified in the params line (for example, | [D1]<TEST>ABC). The space after the | is optional, but the space before it must be present to recognize piping. The number of piping specifications is limited only by the size of the Variable Length Parameter Block allocated by the Executive.

Memory Organization

Figures 4-1 through 4-4 illustrate the memory layout for a BTOS C program in the various memory models. A segment register name with an arrow pointing into a figure represents the location where the segment register is assigned at program start-up. In the medium, large, and huge models, the CS register changes value as functions are called in different source files. In the huge model, DS also changes value as functions are called in different source files.

The heap is an area of storage used to dynamically allocate memory as a program runs. The most common interface to the heap are the functions malloc and free, and other allocation functions. Each block of memory allocated can be a different size. There are no restrictions on the order in which objects are allocated and released.

In the small and medium models, the stack is located at a higher address in memory than the heap, sharing the data segment with the static and global data. The heap grows higher in memory, while the stack grows lower.

Figure 4-1 **Small Model Segments**

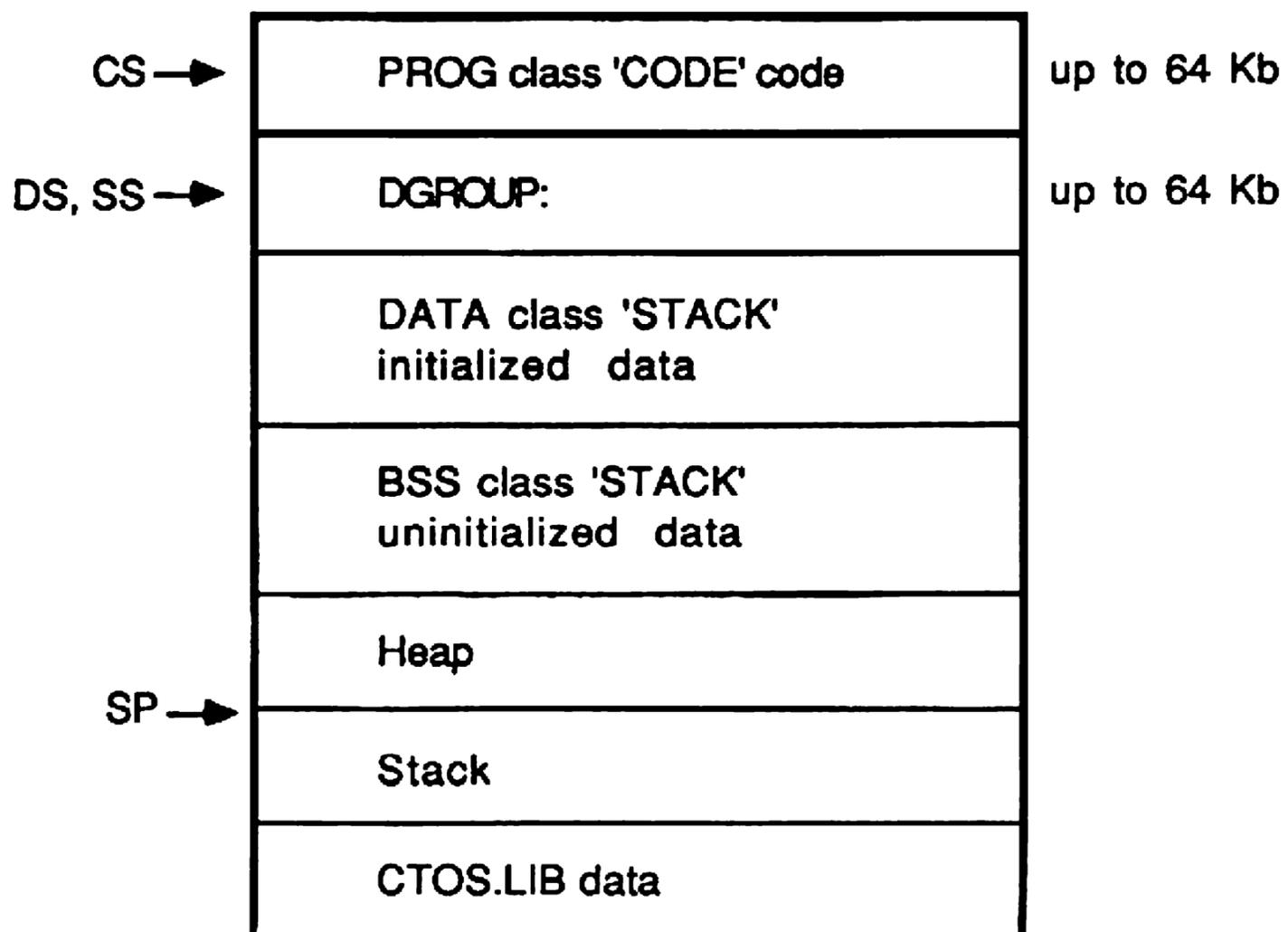
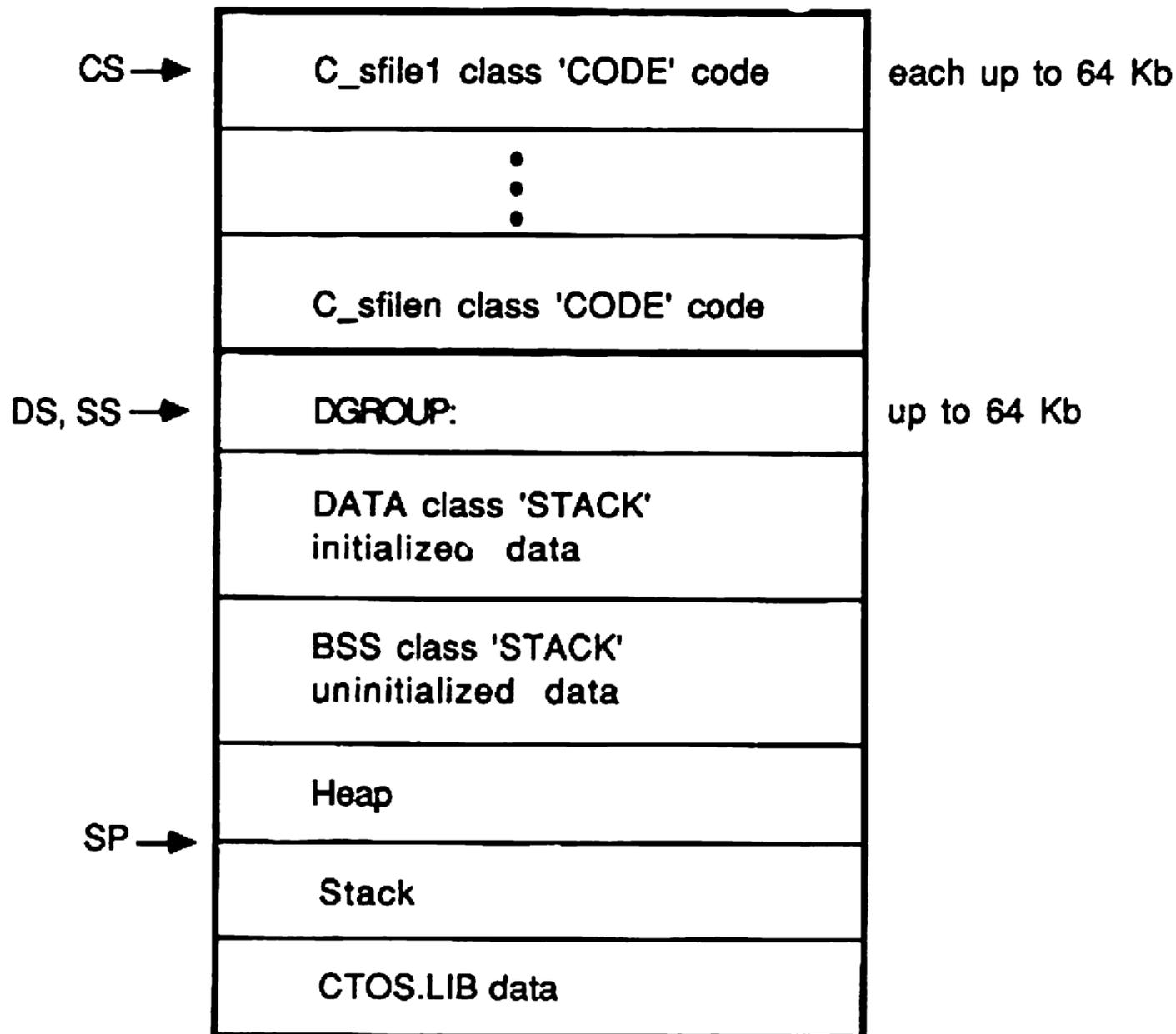


Figure 4-2 Medium Model Segments



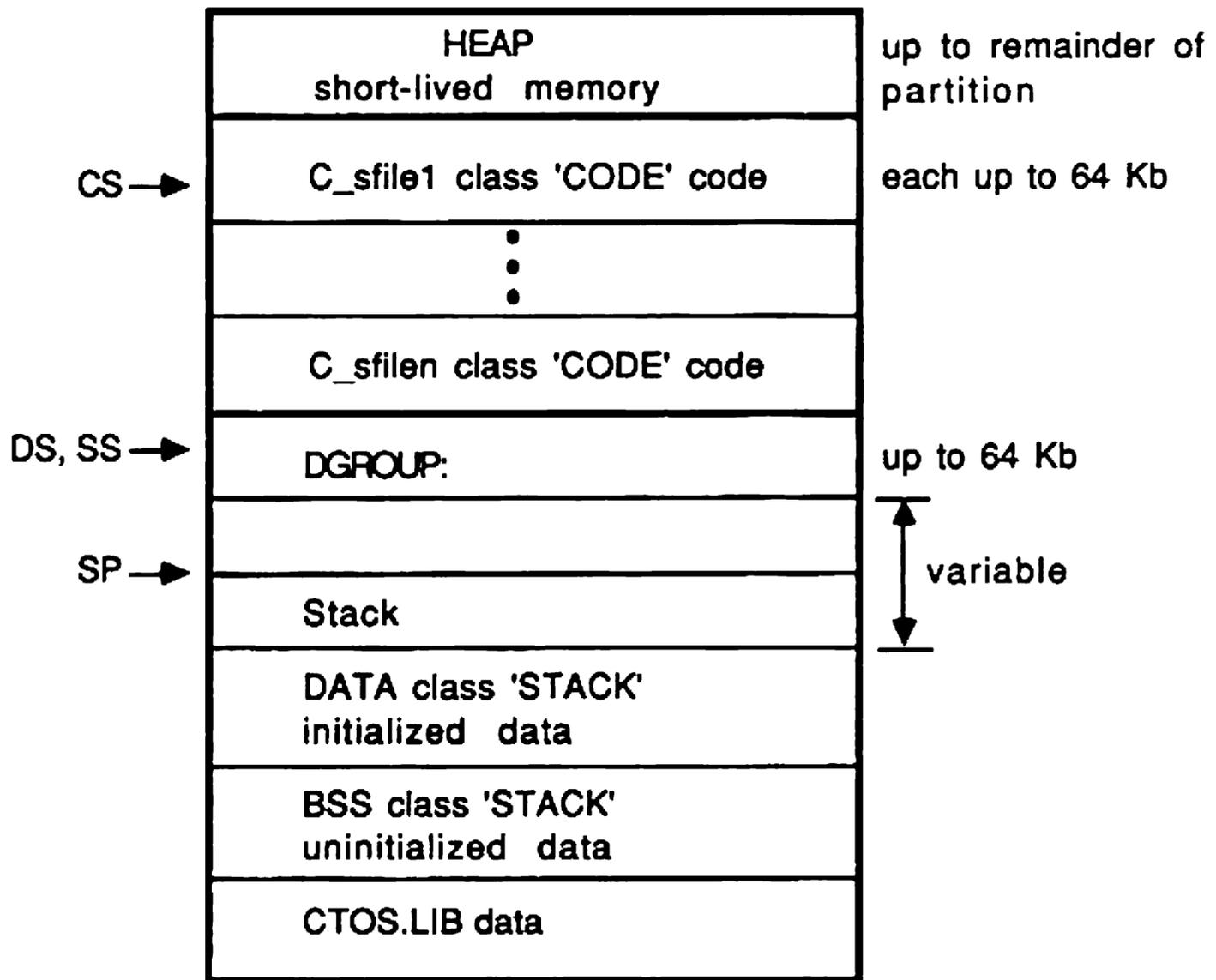
In the large and huge models, the stack is located independently of the heap, and grows downward towards the start of the data segments. Note that the first data segment still combines CTOS.LIB data with the stack. In these models the heap can extend to fill all of short-lived memory.

Pointer Arithmetic

In the small and medium models, data pointers are two bytes long, and arithmetic involving them is very much like simple integer arithmetic. For the large and huge memory models and for far pointers in the other models, there are added complexities.

The Intel 8086 family of microprocessors supports a Segmented Memory architecture. This means that programs may not simply treat all of the workstation memory address space as a single array of characters.

Figure 4-3 Large Model Segments



A memory address used by a program in real mode is constructed as a four-byte quantity. The high-order two bytes are the segment part and the low-order two bytes are the offset part. All 32 bits of these bytes are used to find the physical memory location pointed to by the address.

The segment part is shifted by 4 bits from the offset part to produce a 20-bit result. The resulting segment and offset values are added to determine the real address.

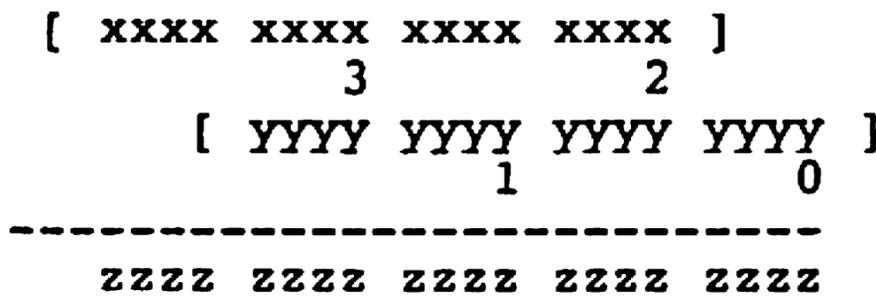
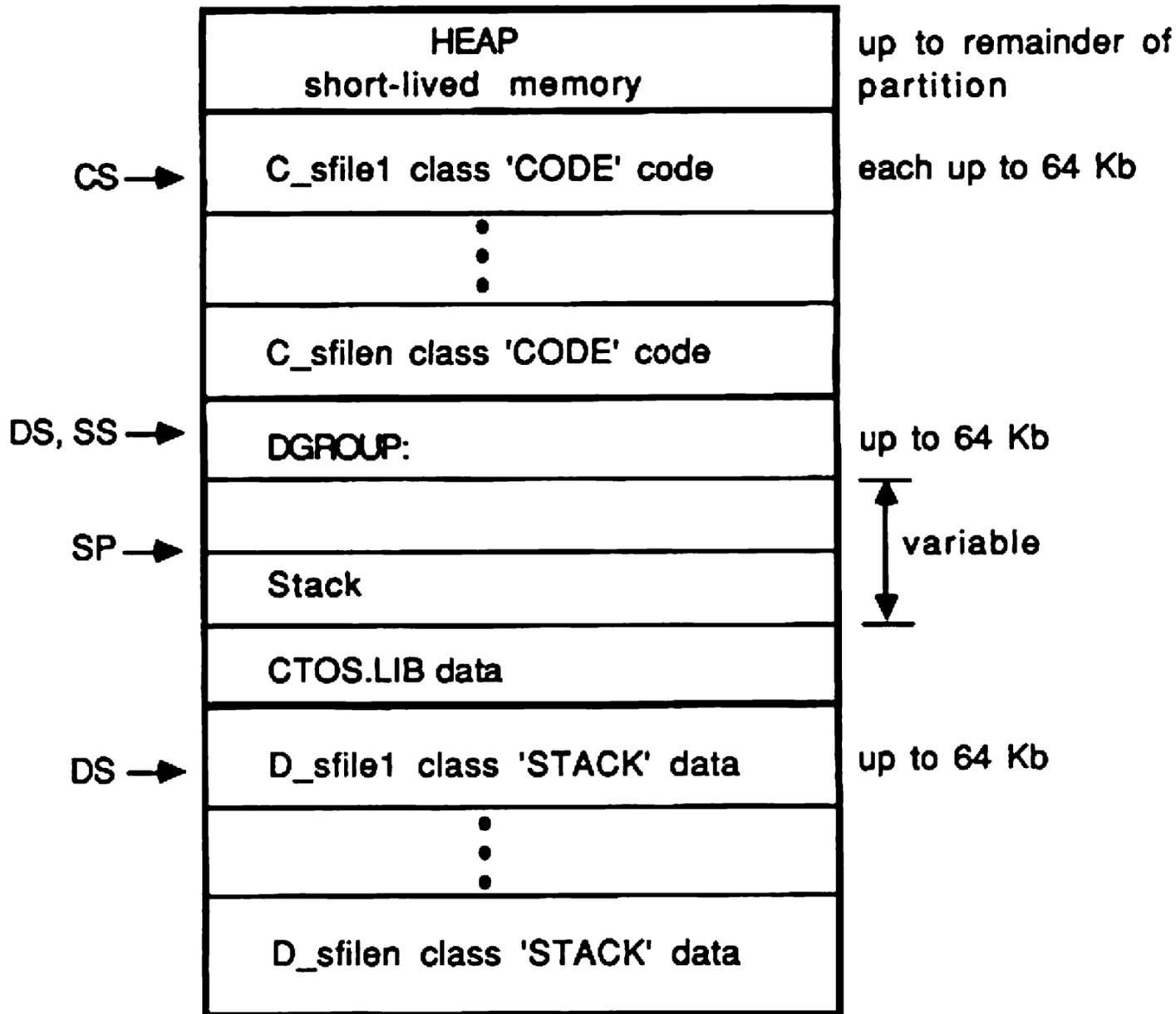


Figure 4-4 Huge Model Segments



There are restrictions on hardware addressing using this scheme. If an instruction uses an index register and offset to address; (for example an array element) the offset and index register value is added. If the result exceeds 64 Kb, no carry bit is retained and the actual resulting location is quite different from that intended.

The 8086 instruction set does not include suitable instructions for easily computing segment/offset combinations where the offset exceeds 64 Kb. However, any carry from the offset must be propagated, not to the low order bit of the segment, but to bit 12. In addition, because of the wrap-around problems of referencing offsets in structures when the index register value is high, one must severely restrict such references or else generate extra code.

For these reasons, there are two pointer arithmetic options for each of the four memory models. The slow version uses subroutine calls for each pointer arithmetic operation. The routines guarantee that the resulting offset is less than 16. Also, when adding a long integer to such a pointer, the operation simulates a large linear array. Individual structures are still limited to 64 Kb because structure offsets must be supplied as part of the workstation instructions for efficiency.

Comparing two pointers in this mode always produces the true relation of the pointers in memory. The difference of two pointers is a signed long integer which reflects the difference as if the two pointers pointed into a single large array. This occurs even if the two pointers were pointing into unrelated places.

The fast pointer arithmetic option uses inline instructions and assumes that the segment part of a pointer is never changed when adding or subtracting integers to or from the pointer. Also, comparing or taking the difference of two pointers assumes that the pointers have the same segment part. These limitations mean that an array must be less than 64 Kb.

In fast pointer arithmetic, two pointers can only be meaningfully compared or subtracted if they point into the same array. Equality and inequality comparisons are always valid.

The fast arithmetic option produces object files in the large model that are only about 20 to 25 percent larger and slower than corresponding small model object files. Slow arithmetic is not much larger, but is considerably slower. Pointer intensive loops can be up to five times slower and overall programs are as much as twice as slow as small model programs doing the same work. Modules compiled with the two different forms of arithmetic may be combined in a single program.

Note: For programs to execute successfully in protected mode, the fast arithmetic option must be specified.

Assembly Language Interface

This section describes the assembly language interface used by C programs, as well as a number of the constructs used in the code generated by the compiler. Programmers intending to write assembly language subroutines called from C or which call C functions, as well as programmers using the inline assembly feature, may use this information to aid in writing their code.

Since the compiler can generate assembly output it may be desirable to hand-optimize fragments of code which are executed many times during a program. You should use the inline assembly feature to substitute assembly statements for C code wherever needed, and not actually modify the assembler output. This way subsequent changes to the source file do not require repeated modification of the assembly output.

External Variable Names

Global symbols, whether a function name or data name, ignore distinctions between uppercase and lowercase letters.

Since C global variable names may have the same spelling as an Assembler reserved word, there are some names you may not be able to use if you specify the `-S` command line option to generate assembly source. This is the case with the ABS library function.

Variable names longer than eight characters in the C source are allowed and are not truncated. If you used the `-i` compiler option to specify the length of an identifier, the name is truncated to whatever length you specified.

The C and PL/M Calling Sequences

The most common method of calling a function used in C programs for the 8086 is the C calling sequence. Refer to section 3, Fast Calling Sequence Option, for a discussion of C versus PL/M calling sequences.

Function Arguments

Arguments are passed to a function on the stack. With the C calling sequence, the arguments are pushed onto the stack in a right-to-left order. For example:

```
int    i,j;
long   k;
.
.
.
i = 5;
j = 7;
k = 0x1407aa;
funca(i, j, k)
```

would load the stack as follows at the entry point to funca using the small memory model:

```
SP + 08:    0014
SP + 06:    07aa          k
SP + 04:    0007          j
SP + 02:    0005          i
SP:         return address
```

For the medium, large, and huge memory models, the stack appears as follows:

```
SP + 10:    0014
SP + 08:    07aa          k
SP + 06:    0007          j
SP + 04:    0005          i
SP + 02:    return segment
SP:         return address
```

An assembly function cannot determine the number of arguments actually passed by the caller. The called

function should not pop any arguments when returning. The calling function does that.

With the PL/M calling sequence, the arguments are pushed onto the stack in a left-to-right order, the reverse of the C sequence. For example:

```
int    ij;
long   k;
.
.
.
i = 5;
j = 7;
k = 0x1407aa;
funca(i, j, k)
```

would load the stack as follows at the entry point to funca using the small memory model:

SP + 08:	0005	i
SP + 06:	0007	j
SP + 04:	0014	
SP + 02:	07aa	k
SP:	return address	

For the medium, large, and huge memory models, the stack appears as follows:

SP + 10:	0005	i
SP + 08:	0007	j
SP + 06:	0014	
SP + 04:	07aa	k
SP + 02:	return	
	segment	
SP:	return address	

An assembly function knows the number of arguments because the number must be the same at all calls. The function uses a RET instruction with an operand of 8. The calling function needs no code to clean up the stack after the call.

Calling Functions

To call a C function from assembly language using the small memory model, the following code is used:

```
EXTRN    FUNCC:NEAR
.
.
.
CALL     FUNCC
```

To call a C function using the medium, large, or huge memory models, the following code is used:

```
EXTRN    FUNCC:FAR
.
.
.
CALL     FUNCC
```

Obviously, only one EXTRN statement for each function being called is needed in the assembly module. Also, the EXTRN statement must be placed outside any segments given in the file.

When calling a C function, arguments should be pushed in right-to-left order. After the called function returns, the caller should pop the number of words pushed. For more than one or two arguments, the best method for popping arguments is to add a constant to SP.

For example:

```
MOV      AX,10          ; push ...
PUSH     AX             ; argument 10
LEA     AX,B           ; push ...
PUSH     AX             ; argument b
LEA     AX,A           ; push ...
PUSH     AX             ; argument a
CALL    MOVMEM         ; movmem(a, b, 10)
ADD     SP,6           ; pop the arguments
```

would call the `movmem` function to copy 10 bytes from the global array `a` to the global array `b`. Using the PL/M calling sequence this same call becomes:

```
LEA      AX,A           ; push ...
PUSH     AX             ; argument a
LEA      AX,B           ; push
PUSH     AX             ; argument b
MOV      AX,10          ; push
PUSH     AX             ; argument 10
CALL     MOVMEM         ; movmem(a, b, 10)
```

Passing Return Values

Integer, unsigned, and enumeration values are returned in `AX`. In the small memory model, pointers to functions are returned in `AX`. In the small and medium memory models, pointers to data are returned in `AX`.

Using the PL/M calling sequence, two-byte pointers are returned in `BX`. This fact forces you to be very careful when declaring C functions using the PL/M calling sequence. You must make sure that all functions returning pointers are explicitly declared as `extern` wherever such functions are used.

Long and unsigned long values are returned in `AX` and `DX`, with the low order bits in `AX`. The high order bits are in `DX`. In the medium memory model, pointers to functions are returned similarly. In the large and huge memory models, all pointers are returned in `AX` and `DX`.

Using the PL/M calling sequence, four-byte pointers are returned in `ES:BX`.

Double values are returned in `AX`, `BX`, `CX`, and `DX`, where `AX` contains the most significant bits of the double, and `DX` the least significant.

Structure values are returned by placing the value in a static data location and placing the address of that location in `BX`. The calling function must copy that value to wherever it is needed. In the large and huge memory models, these values use `ES:BX` to address the static area.

Assembly Language File Structure

Assembly language modules may be included in a C program if they conform to certain conventions used by the compiler. All such assembler modules should begin with the following:

```
NAME          filename
```

An assembly language module which defines code to be used with a C program compiled for the small memory models must begin with these statements:

```
PROG          SEGMENT      BYTE PUBLIC 'CODE'
              ASSUME       CS:PROG
```

For a C program compiled for the medium, large, or huge memory model, the following statements must be used:

```
C_filename   SEGMENT BYTE 'CODE'
              ASSUME   CS:C_filename
```

Filename, by convention, is the name of the source file.

If you redefine the names of the segments generated by the compiler, you may need different values if you have set up your own segmentation scheme.

If you define data elements as follows:

```
DGROUP      GROUP      DATA
DATA        SEGMENT    WORD PUBLIC 'STACK'
```

then you must place the following statement at the beginning of the code segment:

```
ASSUME      DS:DGROUP
```

You may need to use different values if you have changed the segment, group, or class names generated by the compiler.

Defining Functions

To define a function called from a C module using the small memory model, the following declarations are needed at the beginning of the function:

```
FUNCA          PUBLIC      FUNCA
FUNCA          PROC        NEAR
.
.
.
FUNCA                      ENDP
```

For the medium, large, or huge memory models, a slightly different sequence is used:

```
FUNCA          PUBLIC      FUNCA
FUNCA          PROC        FAR
.
.
.
FUNCA                      ENDP
```

An assembly function must preserve the values of the BP, SI, and DI registers. Note that when register variables have been suppressed in the program, SI and DI need not be preserved. The segment registers CS, DS, and SS must always be preserved. ES may be used as a scratch register in all memory models.

Defining Data Constants

Initializing data constants is done in the usual way for numeric constants. For pointers to data in the small and medium memory models, use the following method to define a word containing the address of xxx:

```
DW          DGROUP:xxx
```

To define a pointer to a function in the small memory model, use the following:

```
DW          PROG:xxx
```

To define a pointer to a function in the medium or large memory model, or any far pointer to a function, use the following:

```
DD          xxx
```

To define a pointer to data in the large or huge memory models, or any far pointer to data, use the following:

```
DD          DGROUP:xxx
```

Global Data

To define a global variable, a PUBLIC statement must be included in the DATA segment of an assembly module. For example, to define an integer variable **A** and initialize it to zero, use the following example:

```
          PUBLIC  A
A          DW     0
```

To define an external data variable, an EXTRN statement must be included in the DATA segment of an assembly module, except for the huge model where the EXTRN statement must be placed outside all segments. For example, to define an external integer variable **A**, use the following example:

```
EXTRN     A:WORD
```

To define an external character variable **B**, use the following example:

```
EXTRN     B:BYTE
```

Sample Assembly Language Modules

The following sample Assembly Language source modules implement the ABS function in the specified memory models. The medium model version is the same as the large and huge model versions because no global data or pointer references are included in the function.

Small Model Version

```

PROG      SEGMENT  BYTE PUBLIC 'CODE'
          ASSUME   CS:PROG

          PUBLIC   ABS

ABS       PROC     NEAR
          PUSH    BP
          MOV     BP,SP
          MOV     AX,[BP+4]
          OR     AX,AX           ; set condition codes
          JNL    ADONE
          NEG    AX

ADONE:
          POP    BP
          RET

ABS       ENDP
PROG     ENDS
          END

```

Medium, Large, and Huge Model Version

```

C_ABS    SEGMENT  BYTE 'CODE'
          ASSUME   CS:C_ABS

          PUBLIC   ABS

ABS       PROC     FAR
          PUSH    BP
          MOV     BP,SP
          MOV     AX,[BP+6]
          OR     AX,AX           ; set condition codes
          JNL    ADONE
          NEG    AX

ADONE:
          POP    BP
          RET

ABS       ENDP
C_ABS    ENDS
          END

```


Library Reference

The C runtime library contains functions for performing input/output, numerical calculation, and file management. In addition to the functions in the library, a set of header files (.h extension) are provided to be included in source programs. These header files define many constants needed by application programs, and declare structures and functions which may also be needed.

The first part of this section gives a broad overview of each major component of the library. The second section of this reference discusses each of the header files. The third section provides an alphabetical list of the functions in the runtime library.

Library Overview

Runtime Support

Several standard C operations, particularly the floating point arithmetic operations, are implemented as subroutines. The compiler generates the necessary code to call the subroutines and cause them to be linked. These functions do not conform to normal C calling conventions so C code may not call them explicitly. Documentation of these functions is not included because their interface may be changed without notice and the routines are not callable from your C source program.

The operators implemented as subroutines are: long shift, long multiply, long divide and remainder, double addition, subtraction, multiplication and division, conversion routines between float and double and between long and double, slow pointer arithmetic routines, and structure copy and parameter passing routines.

I/O Operations

The BTOS C Compiler library supports many ways of performing input and output operations. BTOS Services are callable from a C module directly, providing all of the BTOS file access methods directly. The BTOS C Compiler also provides UNIX compatible I/O services for accessing sequential and random access files.

The I/O functions usable with a file are determined by which function was used to open the file. An existing file may be opened using the appropriate open operation, or a new file may be opened by using the appropriate create operation.

Only one of the methods described below should be used to access a single file at one time. Calls from one method cannot be mixed with calls from another. Files opened with BTOS I/O calls may not be used with the Standard I/O method, for example. Detailed documentation for each function mentioned below is given in the individual library entry pages. Refer to those pages for specific information.

If any standard I/O or UNIX compatible I/O function returns an error condition, the code for the BTOS error that caused the error condition is stored into the global variable `errno`.

UNIX Compatible I/O

An existing file is opened using `open`, and a new file is created using `creat`. Once opened, data may be read using `read` or written using `write`. Random access can be gained by using `lseek`. A file may be closed using `close`.

Disk file layouts for text files under BTOS are compatible with those used by UNIX. No special translations are needed by these routines under BTOS.

Standard I/O

An existing or new file is opened using `fopen`. `Fopen` returns a file pointer used by the Standard I/O package to control I/O to a file.

Data may be read in Standard I/O using `getc`, `getchar`, `fgetc`, `getw`, `scanf`, `fscanf`, `fread` and others. Data may be written using `putc`, `putchar`, `fputc`, `printf`, `fprintf` and others. Random access may be gained by using `fseek` or `rewind`. A file may be closed using `fclose`.

The Standard I/O package is strongly oriented toward character streams or free format sequential streams. Buffering is used to make the operations reasonably efficient even for programs which use `getc` or `putc`.

Standard I/O also is more likely to be implemented on non-UNIX systems, making programs using only Standard I/O the most portable to a new system.

BTOS I/O Services

These services are provided by means of directly calling the procedures documented in the BTOS Reference Manual. Header files (CTxxx.H) are provided for the different services. You should use these header files, since the services are accessed as external PL/M procedures and BTOS C generates an incompatible calling mechanism as a default.

Mathematical Functions

A collection of UNIX compatible math functions are provided for exponential, trigonometric, and hyperbolic functions. The functions are listed below:

Function

acos	arc cosine
asin	arc sine
atan	arc tangent
atan2	full circle arc tangent
ceil	ceiling
cos	cosine
cosh	hyperbolic cosine
exp	exponential
fabs	absolute value
floor	floor
fmod	remainder
frexp	return fraction and exponent
ldexp	combine fraction and exponent
log	natural logarithm
log10	logarithm base 10
modf	split integer and fractional part
pow	power
sin	sine
sinh	hyperbolic sine
sqrt	square root
tan	tangent
tanh	hyperbolic tangent

Header Files

This subsection describes the header files available with the C compiler.

ASSERT.H

This header file contains the definition of the assert debugging macro.

CTxxx.H

The following files are the interface header files for BTOS:

ctclust.h	cluster management routines
ctcomm.h	communications routines
ctcont.h	contingency routines
ctdam.h	DAM routines
ctexch.h	exchange routines
ctfile.h	file handling routines
ctinter.h	interrupt service routines
ctkeybd.h	keyboard handling routines
ctmem.h	memory management routines
ctmsg.h	message passing routines
ctos.h	all interfaces in one file
ctparm.h	parameter handling routines
ctpart.h	partition management routines
ctproc.h	process control routines
ctqueue.h	queue management routines
ctrsam.h	RSAM routines
ctserv.h	system service routines
ctspool.h	spooler management routines
ctstam.h	standard access management routines
ctstream.h	Sequential Access Method routines
cttask.h	task management routines
cttimer.h	timer management routines
ctvideo.h	video management routines
ctvirt.h	virtual code segment routines

CTYPE.H

This file is used for the character classification macros such as `isalpha`. These macros provide a convenient means for determining, for example, if a character is an uppercase letter.

ERRNO.H

This file defines constant mnemonics for the error codes returned by the math functions.

FLOAT.H

This file defines the characteristics of floating types and provides values that describe the BTOS C implementation of floating point arithmetic.

I8086.H

The `segread` function fills in the structure `SREGS`. The following fields correspond to the 8086 registers named.

```
struct      XREG      {
             short     ax;
             short     bx;
             short     cx;
             short     dx;
             short     si;
             short     di;
             };

struct      HREG      {
             unsigned   char     al;
             unsigned   char     ah;
             unsigned   char     bl;
             unsigned   char     bh;
             unsigned   char     cl;
             unsigned   char     ch;
             unsigned   char     dl;
             unsigned   char     dh;
             };
```

```

union      REGS      {
            struct    XREG      x;
            struct    HREG      h;
            };

struct     SREGS     {
            short     es;
            short     cs;
            short     ss;
            short     ds;
            };

```

LIMITS.H

This file provides some useful information about compile time limitations of the BTOS implementation. This also contains various values for ranges of integral quantities.

MATH.H

This file declares a number of mathematical functions that return double values.

The functions perform trigonometric, hyperbolic, exponential, and logarithmic calculations.

The macro `HUGEVAL` evaluates to an expression whose value is the maximum double-precision floating point number representable on an 8087.

SETJMP.H

The functions `setjmp` and `longjmp` need this file to define a type used by the functions. The functions are useful for error handling. They allow a program to bypass the normal flow of call and return. A function nested several call-levels deep may return in a single stroke to the top level function. This is dangerous and should be avoided except in limited situations, since the return does not automatically clean up open files or heap memory allocated by any intervening code.

This file defines a type `jmp_buf` as an array used by the `longjmp` and `setjmp` functions.

SIGNAL.H

This file is used by the **signal** and **gsignal** functions. In UNIX this file is more important, but here the file is used to define two constants, **SIG_IGN** and **SIG_DFL**, which are needed by these two functions.

STDARG.H

This file defines the macros used to read the list of arguments in a function declared to accept a variable number of arguments.

STDDEF.H

This file defines several data types and commonly used macros.

STDLIB.H

This file defines several commonly used functions.

This file also defines a type **size_t**, which is the type of the **sizeof** operator.

STDIO.H

This file defines mnemonics, types, and macros needed for the standard I/O package.

This file also defines the type **FILE** used throughout the Standard I/O system and the variables **stdin**, **stdout**, and **stderr**.

The macro **NULL** is defined as a suitably sized 0 for the current memory model. The macro **EOF** is defined to be the standard error and end of file return for Standard I/O functions (and has the value **-1**).

The macro **SYS_OPEN** is defined to be the maximum number of **FILEs** that can be open simultaneously.

STRING.H

This file defines the various string handling functions.

TIME.H

Time.h defines a structure filled in by the time conversion routines localtime and gmtime. UNIX provides a comprehensive set of time and date conversion routines. See the function pages ctime, time, and stime for the treatment of time and date management in this library.

This file defines a type time_t, which is the type of the time value used by the time and stime functions. Note that time_t is defined as a structure, while UNIX defines it as a long integer.

This file also defines the structure tm used by the various ctime functions to hold time information.

Library C Functions

All of the following pages in this section are organized as follows:

The name of the function is the heading of the subsection, followed by a C declaration of the function and global variables described by that entry. Arguments passed to the functions must match the type declared.

The information under the Include Files heading lists each of the header files which are needed if the functions are used.

Return values specify the range of possible values returned by each function, in particular any values indicating error.

The information under Portability gives an indication of whether the functions can be found on UNIX systems, as a guide when portable programs are desired.

abs

```
int      abs(int i);
```

Abs returns the absolute value of the integer argument **i**.

Include Files

```
#include <stdlib.h>
```

Return Value

An integer in the range of 0 to 32767 is returned except for an argument of -32768, which is returned as -32768.

Portability

Available on UNIX systems. This function is defined in the ANSI Standard.

assert

```
void    assert(int test);
```

Assert is a macro that tests a condition and expands to an if statement which, if the test fails, prints a message and terminates the program.

The message is:

```
Assertion failed: file xxx, line nnn
```

The filename and line number are the source file name and line number where the assert macro appears.

If the macro `NDEBUG` is defined before `assert.h` is included, the assert macro becomes null.

Include Files

```
#include    <assert.h>
```

Return Value

This function does not return a value.

Portability

This macro is available on some UNIX systems.

atof, atoi, atol, strtod, strtol

```
double   atof(char *nptr);
int      atoi(char *nptr);
long     atol(char *nptr);
long     strtol(char *nptr, char **endptr, int base);
double   strtod(char *nptr, char **endptr);
```

These functions convert an ASCII string pointed to by `nptr` to the specified return value type.

`Atof` and `strtod` recognize:

- an optional string of tabs and spaces
- an optional sign
- a string of digits and an optional decimal point
- an optional `e` or `E` followed by an optional signed integer

`Atoi`, `atol`, and `strtol` recognize:

- an optional string of tabs and spaces
- an optional sign
- a string of digits

The first unrecognized character ends the conversion.

There are no provisions for overflow.

The third parameter to `strtol` specifies the base for the string of digits. The second parameter to `strtod` and `strtol` is a pointer to an object into which a pointer to the converted string is stored, provided the second parameter is not a null pointer.

Include Files

```
#include <stdlib.h>
```

Return Value

Each function returns the appropriate value of the string. If there are no characters at the beginning of the string that match a number, each function returns zero.

Portability

Available on UNIX systems. All these functions are defined in the ANSI Standard.

Note: For related information, refer to the `scanf` function.

bsearch

```
void      *bsearch( void *key,
                   void *base,
                   int nelem,
                   int width,
                   int (*fcmp)( ));
```

Bsearch is a binary search algorithm designed to search an arbitrary table of information. The address of the table to be searched is passed in base. The table has nelem entries and each entry is width bytes long. Bsearch makes repeated calls to the function whose address is passed in fcmp to do the actual comparisons.

The entries in the table must be sorted into ascending order before bsearch is called.

Fcmp is passed two arguments. The first argument is key and the second argument is the address of some entry in the table being searched. Fcmp must return an integer greater than, equal to, or less than zero according to whether the key is greater to, equal to, or less than the entry in the table. Fcmp is free to interpret key and the table entries anyway it likes.

Include Files

```
#include <stdlib.h>
```

Return Value

Bsearch returns the address of the entry in the table which matches the key. If no match is found, bsearch returns 0.

Portability

Available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the lsearch, qsort, and ssort functions.

check8087, init8087

```
int      check8087( );  
void     init8087( );
```

Check8087 detects the presence on the workstation of an 80x87 NCP. If an 80x87 is detected, the global flag `_8087` is set to 1 and `check8087` returns a non-zero value. If an NCP is not detected, the `_8087` flag is set to 0 and zero is returned.

Init8087 initializes the 80x87 NCP and sets the `_8087` flag to 1.

Include Files

There are no include files required for these functions.

Portability

These functions are unique to the 8086 family and BTOS C.

close

```
int      close(int handle);
```

Handle is a file handle obtained from a `creat` or `open` call. `Close` closes the file handle indicated by `handle`.

`Close` fails if `handle` is not a valid open file handle.

Include Files

```
#include <errno.h>
```

Return Value

Upon successful completion, `close` returns zero. Otherwise, a value of `-1` is returned.

Portability

`Close` is available on UNIX systems.

Note: For related information, refer to the `creat` and `open` functions.

creat

```
int      creat(char *filename, int mode);
```

Creat creates a new file or prepares to rewrite an existing file named by the string pointed to by filename.

If the file exists, the length is truncated to zero and the file attributes are left unchanged.

The creat call accepts a UNIX-style access mode word, which is ignored.

Upon successful creation, the file pointer is set to the beginning of the file. The file is opened for both reading and writing.

Include Files

```
#include <errno.h>
```

Return Value

Upon successful completion, the new file handle is returned, a non-negative integer. Otherwise, a -1 is returned.

Portability

Creat is available on UNIX systems.

Note: For related information, refer to the close, lseek, open, read, and write functions.

ctime, localtime, asctime, gmtime

```
char          *ctime(long *clock);
struct tm     *localtime(long *clock);
struct tm     *gmtime(long *clock);
char          *asctime(struct tm *tm);
```

Ctime converts a time pointed to by clock, such as that returned by the function time, into ASCII and returns a pointer to a 26-character string in the following form. All the fields have constant width.

```
Mon Nov 21 11:31:54 1983\n\0
```

Localtime returns a pointer to a structure containing the broken-down time. Localtime uses the BTOS time of day services to determine the values in the structure.

Asctime converts a broken-down time to ASCII and returns a pointer to a 26-character string.

Gmtime always returns a null pointer. It is provided for compatibility.

The structure declaration from the include file is:

```
struct tm {
    int tm_sec;
    int tm_min;
    int tm_hour;
    int tm_mday;
    int tm_mon;
    int tm_year;
    int tm_wday;
    int tm_yday;
    int tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month (0-11), weekday (Sunday = 0), year -1900, day of year (0-365), and a flag that is non-zero if daylight savings time is in effect.

Include Files

```
#include <time.h>
```

Return Value

Ctime and asctime return the ASCII string date and time. Localtime returns the broken-down time structure. This structure is a static which is overwritten with each call.

Portability

All functions are available on UNIX systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to the stime and time functions.

ecvt, fcvt, gcvt

char *ecvt(double value, int ndigit, int *decpt, int *sign);

char *fcvt(double value, int ndigit, int *decpt, int *sign);

char *gcvt(double value, int ndigit, char *buf);

Ecvt converts the value to a null-terminated string of ndigit ASCII digits and returns a pointer to the string. The position of the decimal point relative to the beginning of the string is stored indirectly through decpt (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by sign is non-zero, otherwise it is zero. The low-order digit is rounded.

Fcvt is identical to ecvt, except that the correct digit has been rounded for F-format output of the number of digits specified by ndigit.

Gcvt converts the value to a null-terminated ASCII string in buf and returns a pointer to buf. It attempts to produce ndigit significant digits in F-format if possible, otherwise E-format, ready for printing. Trailing zeros may be suppressed.

Include Files

There are no include files required for these functions.

Return Value

The return values point to static data whose content is overwritten by each call to ecvt or fcvt. Gcvt returns the string pointed to by buf.

Portability

These functions are available on UNIX.

Note: For related information, refer to the printf function.

exit, _exit

```
void    exit(int status);
```

```
void    _exit(int status);
```

Exit terminates the current program and returns control to BTOS. All files are closed and buffered output waiting to be output is written before exiting.

_exit terminates without closing any files or flushing any output.

In either case, status is returned as the exit status of the program.

Include Files

```
#include    <stdlib.h>
```

Return Value

Exit does not return a value.

Portability

Available on UNIX. The exit function is defined in the ANSI Standard.

exp, log, log10, pow, sqrt

```
double    exp(double x);
double    log(double x);
double    log10(double x);
double    pow(double x, double y);
double    sqrt(double x);
```

Log returns the natural logarithm of x.

Log10 returns the base 10 logarithm of x.

Pow returns $x ** y$.

Sqrt returns the positive square root of x.

Include Files

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

Return Value

Exp and pow return a huge value when the correct value would overflow. A large argument can result in errno being set to ERANGE.

Log returns a huge negative value and sets errno to EDOM when x is less than or equal to zero.

Pow returns a huge negative value and sets errno to EDOM when x is less than zero and y is not a whole number.

Sqrt returns 0 and sets errno to EDOM when x is negative.

Portability

Available on UNIX. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the sinh and trig functions.

fclose, fflush

```
int      fclose(FILE *stream);
```

```
int      fflush(FILE *stream);
```

Fclose causes any buffers for the named stream to be written and the files to be closed. Buffers allocated by malloc are freed.

Fclose is performed automatically upon calling exit.

Fflush causes any buffered data being output to a named stream to be written. The stream remains open.

Include Files

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

Return Value

These functions return 0 upon success, and EOF if any errors were detected.

Portability

Available on UNIX Systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to the close, fopen, and setbuf functions.

feof, ferror, clearerr

```
int      feof(FILE *stream);  
int      ferror(FILE *stream);  
void     clearerr(FILE *stream);
```

Feof returns non-zero if an end of file was detected on the last input operation on the named stream.

Ferror performs stream status inquiries. Ferror returns non-zero if an error was detected on the named stream.

Clearerr resets the error indication on the named stream.

These are implemented as macros.

The end of file indicator is reset with each input operation.

Portability

Available on UNIX. These macros are defined in the ANSI Standard, although the Standard requires that they exist as functions in addition to being defined as macros.

Note: For related information, refer to the open and fopen functions.

floor, ceil, fmod, fabs

double floor(double x);

double ceil(double x);

double fmod(double x, double y);

double fabs(double x);

Floor returns the largest integer (as a double) not greater than x .

Ceil returns the smallest integer (as a double) not less than x .

Fmod returns the number f such that $x = iy + f$, for some integer i , and $0 \leq f < y$.

Fabs returns the absolute value of x .

Include Files

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

Portability

Available on UNIX Systems. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the abs function.

fopen, freopen

FILE *fopen(char *filename, char *type);

FILE *freopen(char *filename, char *type, FILE *stream);

Fopen opens the file named by filename and associates a stream with it. Fopen returns a pointer to be used to identify the stream in subsequent operations.

Freopen substitutes the named file in place of the open stream. The original stream is closed, regardless of whether the open succeeds.

Freopen is useful for changing the file attached to stdin, stdout, or stderr.

The type string used in each of these calls is one of the following values:

- “r” open for reading only
- “w” create for writing
- “a” append; open for writing at end of file, or create for writing if the file does not exist
- “r+” open an existing file for update (reading and writing)
- “w+” create a new file for update
- “a+” open for append; open (or create if the file does not exist) for update at the end of the file

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening fseek or rewind, and input may not be directly followed by output without an intervening fseek, rewind, or an input which encounters end of file.

Include Files

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

Return Value

On successful completion, each function returns the newly opened stream. Freopen returns the argument stream. In the event of error each function returns NULL.

Portability

These functions are available on UNIX Systems and are also defined in the ANSI Standard.

Note: For related information, refer to the open and fclose functions.

fread, fwrite

```
int    fread(void *ptr, int size, int nitems, FILE *stream);
```

```
int    fwrite(void *ptr, int size, int nitems, FILE *stream);
```

Fread reads, into a block pointed to by ptr, nitems of data of the type ptr pointed to from the named input stream.

Fwrite appends nitems of the type pointed to by ptr beginning at ptr to the named output stream.

Ptr in the declarations is a pointer to any object. Size is the size of the object ptr points to. The expression sizeof *ptr produces the proper value.

Include Files

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

Return Value

On successful completion, each function returns the number of items (not bytes) actually read or written. Fread returns a short count (possibly zero) on end of file or error. Fwrite returns a short count on error.

Portability

These functions are available on all UNIX systems. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the read, write, fopen, getc, putc, gets, puts, printf, and scanf functions.

frexp, ldexp, modf

double frexp(double value, int *eptr);

double ldexp(double value, int exp);

double modf(double value, double *iptr);

Frexp returns the mantissa of a double value as a double quantity, x , of magnitude less than 1 and stores an integer n such that $\text{value} = x * 2 ** n$. The number n is stored in the integer pointed to by $eptr$.

Ldexp returns the quantity $\text{value} * 2 ** \text{exp}$.

Modf returns the fractional part of value and stores the integer part in the double pointed to by $iptr$.

Include Files

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

Portability

These functions are available on all UNIX systems. These functions are also defined in the ANSI Standard.

fseek, ftell, rewind

```
int      fseek(FILE *stream, long offset, int whence);
```

```
long     ftell(FILE *stream);
```

```
int      rewind(FILE *stream);
```

Fseek sets the file pointer for the next input or output operation on the stream. The new position is at the signed distance offset bytes from the

- beginning
- current position
- end of the file

respectively, as whence has the value 0, 1, or 2.

Fseek discards any character pushed back using ungetc.

After fseek or rewind, the next operation on an update file may be either input or output.

Ftell returns the current file pointer. The offset is measured in bytes from the beginning of the file.

Rewind(stream) is equivalent to fseek(stream, 0L, 0).

Include Files

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

Return Value

Fseek and rewind return non-zero for improper seeks, otherwise zero. Ftell returns the current file position, or end of file on an error.

Portability

These functions are available on all UNIX systems. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the lseek and fopen functions.

getc, getchar, fgetc, getw

```
int    getc(FILE *stream);
int    getchar(void);
int    fgetc(FILE *stream);
int    getw(FILE *stream);
```

Getc returns the next character on the named input stream.

Getchar() is a macro defined to be getc(stdin).

Fgetc behaves exactly like getc, except that it is a true function while getc is a macro.

Getw returns the next integer in the named input stream. Getw assumes no special alignment in the file.

Include Files

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

Return Value

Getc, getchar and fgetc return the next input character upon success. On end of file or error, they return EOF.

Getw returns the next integer on the input stream. On end of file or error, getw returns EOF. Because EOF is a legitimate value for getw to return, feof or ferror should be used to detect end of file or error.

Portability

All functions are available on UNIX systems. These macros and functions are also defined in the ANSI Standard.

Note: For related information, refer to the ferror, fopen, fread, gets, putc, and scanf functions.

gets, fgets

```
char    *gets(char *s);
```

```
char    *fgets(char *s, int n, FILE *stream);
```

Gets reads a string into *s* from the standard input stream *stdin*. The string is terminated by a newline character, which is replaced in *s* by a null character.

Fgets reads *n*-1 characters, or up to a newline character (which is retained), whichever comes first, from the stream into the string *s*. The last character read into *s* is followed by a null character. Fgets returns its first argument.

Include Files

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

Return Value

Each function, on success, returns the string argument *s*.

Each returns NULL on end of file or error.

Portability

Available on UNIX systems. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the *ferror*, *fopen*, *fread*, *getc*, *puts*, and *scanf* functions.

index, rindex

int index(char *s, char *t);

int rindex(char *s, char *t);

Index returns the index of the leftmost occurrence of the string `t` in `s` (not counting the terminating null character). The first character of `s` is numbered 0, so that subscripting produces the correct result.

For example, a call of:

```
index("four score and seven", " s");
```

returns 4.

Rindex returns the index of the rightmost occurrence of `t` in `s`.

Include Files

There are no Include Files required for this function.

Return Value

Both functions return a non-negative index if a character was found, and -1 if no character was found.

Portability

These functions are not available under UNIX System III or V.

Note: For related information, refer to the string function.

inport, inportb

```
int      inport(int port);
```

```
int      inportb(int port);
```

Inport reads the value of a word port and returns the value read.

Inportb read the value of a byte port and returns the value read.

Include Files

```
#include <i8086.h>
```

Portability

These functions are unique to the 8086 family of microprocessors.

**isalpha, isupper, islower, isdigit,
isxdigit, isalnum, isspace, ispunct,
isprint, isgraph, iscntrl, isascii**

```
int    isalpha(int c);
int    isupper(int c);
int    islower(int c);
int    isdigit(int c);
int    isxdigit(int c);
int    isalnum(int c);
int    isspace(int c);
int    ispunct(int c);
int    isprint(int c);
int    isgraph(int c);
int    iscntrl(int c);
int    isascii(int c);
```

These macros classify ASCII-coded integer values by table lookup. Each is a predicate returning non-zero for true and zero for false. `isascii` is defined on all integer values; the rest are defined only where `isascii` is true and on the single non-ASCII value EOF.

Include Files

```
#include <ctype.h>
```

Return Value

`isalpha` Non-zero if `c` is a letter.
`isupper` Non-zero if `c` is an uppercase letter.
`islower` Non-zero if `c` is a lowercase letter.

<code>isdigit</code>	Non-zero if <code>c</code> is a digit.
<code>isxdigit</code>	Non-zero if <code>c</code> is a hexadecimal digit [0-9], [A-F] or [a-f].
<code>isalnum</code>	Non-zero if <code>c</code> is an alphanumeric.
<code>isspace</code>	Non-zero if <code>c</code> is a space, tab, carriage-return, newline, vertical tab, or form-feed.
<code>ispunct</code>	Non-zero if <code>c</code> is a punctuation character (neither control nor alphanumeric).
<code>isprint</code>	Non-zero if <code>c</code> is a printing character, code 0x20 (space) through 0x76 (tilde).
<code>isgraph</code>	Non-zero if <code>c</code> is a printing character, like <code>isprint</code> , except that space is excluded.
<code>isctrl</code>	Non-zero if <code>c</code> is a delete character (0x7f) or ordinary control character (0x00 to 0x3f).
<code>isascii</code>	Non-zero if <code>c</code> is an ASCII character, code in the range from 0x00 to 0x7f.

Portability

All these macros are available on UNIX workstations. These macros are defined in the ANSI Standard, although the Standard requires that these be available as functions as well as macros.

lsearch

```
char *lsearch(void *key,  
              void *base,  
              int nelemp,  
              int width,  
              int (*fcmp)( ));
```

Lsearch is a linear search algorithm that searches a table for a specific key, and if not found inserts it at the end of the table. The address of the table is given in base. Nelemp points to a word containing the number of entries in the table. Width contains the number of bytes in each entry. Key points to the item to be searched for. Lsearch calls the function pointed to by fcmp repeatedly until the item is found or the end of the table is reached.

Fcmp is called with two arguments. The first is key, the address of the item being searched for. The second is the address of an entry in the table. Fcmp must return zero if the two items are equal, and non-zero if they are not equal.

If the search item is not found in the table, it is copied into the end of the table and the word pointed to by nelemp is incremented. The table must have enough room to add any new entries. If there is not enough room, unpredictable results may occur.

Include Files

There are no Include Files required for this function.

Return Value

Lsearch returns the address of the entry matching the search key. If the item was not in the table, then lsearch returns the address of the new entry.

Portability

This function is available on UNIX systems.

lseek

long lseek(int handle, long offset, int whence);

Handle is a file handle obtained from a `creat` or `open` call. `Lseek` sets the file pointer associated with handle as follows:

If `whence` is 0, the pointer is set to offset bytes.

If `whence` is 1, the pointer is set to its current location plus offset.

If `whence` is 2, the pointer is set to the size of the file plus offset.

Return Value

Upon successful completion, a non-negative integer indicating the file pointer value is returned. Otherwise, a value of `-1` is returned.

Portability

Available on UNIX systems.

Note: For related information, refer to the `creat` and `open` functions.

malloc, calloc, free, cfree, realloc

```
void      *malloc(unsigned size);
void      *calloc(unsigned nelem, unsigned elsize);
void      free(void *ptr);
void      cfree(void *ptr);
void      *realloc(void *ptr, unsigned size)
```

These functions provide access to the C memory heap. The heap is available for use for creating variable sized blocks of memory. Many data structures such as trees and lists naturally employ heap memory allocation.

Malloc returns a pointer to a memory block of length size. If not enough memory is available to allocate the block, malloc returns NULL (0). The maximum value of size is 65526.

Calloc allocates a block like malloc, except the block is of size nelem times elsize. The block is cleared to zero.

Free deallocates a previously allocated block. Ptr must contain the address of the first byte of the block.

Cfree is an alternative name for free. They each perform the same work and may be used interchangeably.

Realloc changes the size of a block previously allocated. Ptr is the address of the block. Size is the new size in bytes.

Blocks may be allocated and freed in any order.

Include Files

```
#include <stdlib.h>
```

Return Value

Malloc, realloc, and calloc return a null pointer (0) if there is not enough space available to allocate the needed block. When realloc returns 0, the block pointed to by ptr is preserved.

Portability

Calloc, malloc, realloc, and free are available on UNIX systems. Calloc, malloc, realloc, and free are also defined in the ANSI Standard.

memcpy, memset, memcmp, memchr

```
void      *memcpy(void *dst, void *src, unsigned n);
```

```
void      *memset(void *s, char c, unsigned n);
```

```
int        memcmp(void *s1, void *s2, unsigned n);
```

```
void      *memchr(void *s, char c, unsigned n);
```

These functions are portable memory functions.

Memcpy copies *n* bytes from the *src* to the *dst* array.

Memset sets all of the bytes of *s* to the char *c*. The size of the *s* array is given by *n*.

Memcmp compares two strings, given by *s1* and *s2*, for a length of *n* bytes.

Memchr searches the first *n* bytes of array *s* for *c*.

In all of these functions arrays are *n* bytes in length, even if they contain null bytes.

Include Files

```
#include    <string.h>
```

Return Value

Memcpy returns the value of *dst*.

Memset returns the value of *s*. Memcmp returns -1 , 0 , or 1 depending on whether the *s1* string is less than, equal to, or greater than the *s2* string. Exactly *n* bytes are compared. Memchr returns a pointer to the first occurrence of *c* in *s*, or 0 if *c* does not occur in the *s* array.

Portability

Available on UNIX System V systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to `movmem` and `setmem`.

movmem

```
void    movmem(void *src, void *dest, unsigned len);
```

Movmem moves a block of len bytes from src to dest using the 8086 string move instruction. If the source and destination strings overlap, the copy direction is chosen so that the data is always copied correctly.

Include Files

```
#include    <i8086.h>
```

Portability

This function is unique to BTOS C.

Note: For related information, refer to the memcpy and string functions.

open

```
int      open(char *filename, int oflag);
```

This function opens a file for reading or writing.

Filename points to a string naming a file. The function opens a handle for the named file according to the value of oflag.

For open, the oflag values may be:

- 0 Read access only
- 1 Write access only
- 2 Both read and write access

Upon successful completion a non-negative integer, the file handle, is returned.

The file pointer used to mark the current position in the file is set to the beginning of the file.

The maximum number of simultaneously open files is 20.

Return Value

On successful completion, this function returns a non-negative integer.

On error, open returns -1.

Portability

Open is available on UNIX systems.

Note: For related information, refer to the close, creat, lseek, read, and write functions.

outport, outportb

```
void    outport(int port, int value);  
void    outportb(int port, char value);
```

Outport writes value to the word port.

Outportb writes value to the byte port.

Include Files

```
#include <i8086.h>
```

Portability

These functions are unique to the 8086 family.

peek, peekb

```
int    peek(int segment, int offset);
```

```
char   peekb(int segment, int offset);
```

Peek returns the integer stored at the memory location addressed by segment and offset. Segment is treated as a paragraph address, while offset is a byte offset from the segment.

Peekb returns the byte stored at the memory location addressed by segment and offset.

Include Files

```
#include <i8086.h>
```

Portability

These functions are unique to the 8086 family.

Note: For related information, refer to the poke function.

poke, pokeb

```
void    poke(int segment, int offset, int value);
```

```
void    pokeb(int segment, int offset, char value);
```

Poke deposits the integer value at the memory location addressed by segment and offset. Segment is a paragraph address, while offset is a byte offset from that address.

Pokeb is the same as poke, except that a byte is deposited instead of an integer.

Include Files

```
#include <i8086.h>
```

Portability

These functions are unique to the 8086 family.

Note: For related information, refer to the peek function.

printf, fprintf, sprintf

```
int    printf(char *format, ...);
int    fprintf(FILE *stream, char *format, ...);
int    sprintf(char *s, char *format, ...);
```

These functions format output.

Printf places its output on the standard output stream `stdout`. Fprintf places its output on the named stream. Sprintf places output, followed by the null character (`\0`), in consecutive bytes starting at the address `s`. With `sprintf` it is the user's responsibility to ensure there is enough space in `s` to hold the formatted output. Characters generated by `printf` and `fprintf` are printed as if `putc` had been called.

Each of these functions converts, formats, and prints its args under control of the format string. The format is a character string containing two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more args. The results are unpredictable if there are insufficient args for the format. If the format is exhausted while args remain, the excess args are ignored.

Each conversion specification is begun by the character `%`. After the `%`, the following options appear in sequence.

An optional list of flag characters appears, in any order:

- Forces the result of the conversion to left-justified within the field.
 - + The result of a signed conversion always begins with a sign (`+` or `-`).
 - space If the first character of a signed conversion is positive, a space is used instead of a `+` character. A negative result will show as a `-` character.
- The `+` flag takes precedence over blank if both flags are present.

The # flag specifies that the arg is to be converted using an alternate form. For c, d, s, or u conversions, the flag has no effect. For octal (o) conversion, it increases the precision to force the first digit of the result to be zero. For x (X) conversion, a non-zero arg has 0x (0X) preceding it. For e, E, f, g, and G conversions, the result always contains a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeroes are not removed from the result (as they are normally).

An optional width specifier, width is given either by a decimal digit string or by the character asterisk (*). An asterisk indicates that the width should be obtained by using the next arg in the call (treating it as an integer). In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

An optional precision specifier. The precision, if present, is preceded by a decimal point to separate it from any preceding width specifier. The precision specifier is either a decimal digit string or an asterisk. The asterisk, as in the width specifier, indicates that the precision should be determined by using the next arg in the call (treating it as an integer).

Note: If asterisks are used for width or precision specifiers, the width arg must appear first, then the precision arg if any, and finally the arg for the data to be converted.

An optional character l follows specifying that a following d, o, u, x, or X conversion applies to a long integer arg instead of an integer.

The conversion character itself then appears. The conversion characters and their meanings are:

d,o,u,x,X The integer (or long integer if `l` preceded the conversion character) `arg` is converted to signed decimal, unsigned octal, unsigned decimal, or hexadecimal (`x` or `X`), respectively. For hexadecimal conversions the letters `abcdef` are used if the conversion character was lowercase `x`, and `ABCDEF` if the conversion was uppercase `X`. The precision specifies the minimum number of digits to appear; if the value being converted needs fewer digits, the output is padded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of zero is a null string (unless the conversion is `o`, `x`, or `X` AND the `#` flag is present).

A leading zero given with the width of the format spec (for example a `"%04d"` format spec) forces `printf` to display the number with zero fill instead of blank fill.

f The float or double `arg` is converted to decimal notation in the style `"[-]ddd.ddd"&!1` where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, six digits are output; if the precision is explicitly 0, no decimal point appears.

e,E The float or double `arg` is converted in the style `"[-]d.ddde{+-}ddd"`, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. If the exponent is less than 100 it will contain two digits; otherwise, three digits.

- g,G** The float or double arg is printed in style f or e (or in style E in the case of a G format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style e is used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeroes are removed from the result; a decimal point appears only if it is followed by a digit.
- c** The character arg is printed.
- s** The arg is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision is reached. If the precision is missing, it is taken to be infinite, so all characters up to the first null character are printed.
- p,P** The arg is taken to be a pointer. The pointer is assumed to be a near pointer for the small and medium memory models, and a far pointer for the large and huge memory models. A far pointer is printed as `XXXX:YYYY`. A near pointer is printed as `YYYY` (offset only). `XXXX` and `YYYY` are printed in hex format: `04x` if `p` is specified, or `04X` if `P` is specified. Both `p` and `P` may be prefixed with `N` or `F` to indicate that the arg is a near or far pointer, respectively, regardless of the memory model.
- %** `cPrint a %`; no argument is converted.

Include Files

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

Return Value

Each function returns the number of bytes output. `Sprintf` does not include the null byte in the count.

In the event of error, these functions return end of file.

Portability

These functions are available on UNIX systems. The ANSI Standard provides a definition for these functions, but with slightly greater functionality.

Note: For related information, refer to the `ecvt`, `putc`, and `scanf` functions.

putc, putchar, fputc, putw

```
int      putc(char c, FILE *stream);
int      putchar(char c);
int      fputc(char c, FILE *stream);
int      putw(unsigned w, FILE *stream);
```

Putc and fputc append the character *c* to the named output stream. Putc is a macro, while fputc is a true function.

Putchar(*c*) is a macro defined to be putc (*c*, stdout).

Putw appends the unsigned integer *w* to the output stream. Putw neither expects nor causes special alignment in the file.

The streams stdout and stderr are unbuffered, while all other output files are by default buffered. Setbuf may be used to change the buffering style being used. Unbuffered means that characters written to a stream are immediately output to the file or device, while buffered means that the characters are accumulated and written as a block.

Include Files

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

Return Value

Putc, fputc, and putchar return the character *c* on success. Putw returns the integer *w*. On error all the functions return EOF. Since EOF is a legitimate integer, ferror should be used to detect errors with putw.

Portability

All functions are available on UNIX systems. All these functions, except putw, are defined in the ANSI Standard.

Note: For related information, refer to the ferror, fopen, fwrite, getc, printf, and puts functions.

puts, fputs

```
int      puts(char *s);
```

```
int      fputs(char *s, FILE *stream);
```

Puts copies the null-terminated string *s* to the standard output stream `stdout` and appends a newline character.

Fputs copies the null-terminated string *s* to the named output stream, and does not append a newline character.

Include Files

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

Return Value

Upon successful completion, each function returns 0. Otherwise, a value of EOF is returned in the event of an error.

Portability

Both functions are available on UNIX systems. These functions are also defined in the ANSI Standard.

Note: For related information, refer to the `ferror`, `fopen`, `fwrite`, `gets`, `printf`, and `putc` functions.

qsort, ssort

```
void    qsort(void *base,
             int nelem,
             int width,
             int (*fcmp)( ));
```

```
void    ssort(void *base,
             int nelem,
             int width,
             int (*fcmp)( ));
```

Qsort is an implementation of the quicker-sort algorithm. Ssort is an implementation of the shell-sort algorithm. Base is a pointer to the table to be sorted. Nelem is the number of entries in the table. Width is the size in bytes of each entry in the table. Qsort and ssort sort the entries into order by repeatedly calling the function pointed to by fcmp.

Fcmp accepts two arguments, each the address of an entry in the table. Fcmp returns a number greater than zero if the first argument should appear after the second in the final sequence. Fcmp returns a number less than zero if the first argument should appear before the second in the final sequence. Fcmp returns zero if the two arguments are equal. If the table entries are already sorted, qsort requires approximately (20*nelem) bytes of stack space. Ssort always uses only 20 bytes of stack space.

Include Files

```
#include    <stdlib.h>
```

Return Value

These functions do not return a value.

Portability

Qsort is available on UNIX systems and is defined in the ANSI Standard. Ssort is not portable.

Note: For related information, refer to the bsearch and lsearch functions.

rand, srand

```
void    srand(unsigned seed);  
int     rand(void);
```

Rand uses a multiplicative congruential random number generator with period 2^{32} to return successive pseudo-random numbers in the range from 0 to $2^{15} - 1$.

The generator is reinitialized by calling `srand` with an argument value of 1. It can be set to a random starting point by calling `srand` with whatever you choose as argument.

Include Files

```
#include <stdlib.h>
```

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

read

```
int      read(int handle, void *buf, int nbyte);
```

Read attempts to read `nbyte` bytes from the file associated with `handle` into the buffer pointed to by `buf`.

`Handle` is a file handle obtained from a `creat` or `open` call.

On disk files, the read begins at the current file pointer. On completion of the read, the file pointer is incremented by the number of bytes read.

On devices the bytes are read directly from the device.

Upon successful completion, the functions return the number of bytes read and placed in the buffer.

A value of zero is returned when end of file is reached.

Include Files

```
#include <errno.h>
```

Return Value

Upon successful completion a positive integer is returned, indicating the number of bytes placed in the buffer.

On end of file, `read` returns zero.

On error, `read` returns `-1`.

Portability

`Read` is available on UNIX systems.

Note: For related information, refer to the `creat` and `open` functions.

scanf, fscanf, sscanf

int scanf(char *format, ...);

int fscanf(FILE *stream, char *format, ...);

int sscanf(char *s, char *format, ...);

Scanf reads from the standard input stream stdin.

Fscanf reads from the named input stream.

Sscanf reads from the character string s.

Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects as arguments a control string format, described below, and a set of pointer arguments indicating where the converted input should be stored.

The control string usually contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

- Blanks, tabs, or newlines, which cause input to be read up to the next non-white-space character.

- An ordinary character (not %), which must match the next character of the input stream.

- Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless argument suppression was indicated by *. An input field is defined as a string of non-space characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must usually be of a restricted type. The following conversion characters are legal:

%

A single % is expected in the input. No assignment is done.

d

A decimal integer is expected. The corresponding argument should be a pointer to an integer.

o

An octal integer is expected. The corresponding argument should be an integer pointer.

x

A hexadecimal integer is expected. The corresponding argument should be an integer pointer.

i

An integer is expected. If it begins with 0x or 0X it is assumed to be hexadecimal. If it begins with 0 it is assumed to be octal. Otherwise, it is assumed to be decimal. The corresponding argument should be a pointer to an integer.

s

A character string is expected. The corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating \0, which is added automatically. The input field is terminated by a space or a newline.

c

A character is expected. The corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case. To read the next non-space character, use %1s. If a field width is given, the corresponding argument should refer to a character array, the indicated number of characters is read.

e,f

A floating point number is expected. The corresponding argument should be a pointer to a float. The input format for floating point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an E or an e, followed by an optionally signed integer.

[

Indicates a string that is not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket. The characters between the brackets define a set of possible characters making up the string. If the first character is a circumflex (^), then the search set of characters is inverted to include all ASCII characters EXCEPT those between the circumflex and the right bracket. The input is scanned until a character not in the search set is found. The corresponding argument should point to a character array.

The conversion characters d, o, x, and i may be capitalized and/or preceded by the letter l or L to indicate that a pointer to long rather than an int is in the argument list. Similarly, the conversion characters e and f may be capitalized and/or preceded by the letter l or L to indicate a pointer to double rather than to float is in the argument list.

Scanf conversion terminates at end of file, at the end of the control string, or when an input character conflicts with the control string. In the latter case, the offending character is left unread in the input stream.

Trailing white space is left unread (including a newline) unless explicitly matched in the control string.

A pointer to unsigned character, integer, or long can be used in any conversion where a pointer to a character, integer, or long is allowed.

The success of literal matches and suppressed assignments is not directly determinable.

Sscanf does not change the source string s.

Include Files

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

Return Value

Upon successful completion these functions return the number of successfully matched and assigned input items. In the event of a conflict between the format string and the input, a lesser count is returned, which may be zero if the conflict occurs early enough. If the input ends before the first conflict or conversion, EOF is returned.

Portability

These functions, except for the *i* (general integer) conversion, are available on UNIX systems. These functions, with slightly greater functionality, are defined in the ANSI Standard.

Note: For related information, refer to the *atof*, *getc*, and *printf* functions.

segread

```
void    segread(struct SREGS *segtbl);
```

This function reads segment registers and places the current values into the segtbl structure. For the small memory model, the value of the CS register does not change during execution. For the medium, large, and huge memory models, the CS register varies from one function to another. Each source file is given a different CS register value.

For all memory models except huge, DS and SS are the same and remain unchanged for the complete execution of a program. For the huge model, each source file is given a different DS register value. For all models, ES may vary during program execution.

Include Files

```
#include    <i8086.h>
```

Portability

This function is unique to BTOS C.

setbuf, setvbuf

void setbuf(FILE *stream, char *buf);

int setvbuf(FILE *stream, char *buf, int type, unsigned size);

Setbuf and setvbuf are used after a stream is opened but before any reading or writing is done on the stream. They cause the buffer buf to be used instead of an automatically allocated buffer.

In setvbuf, the type parameter is one of the following:

- `_IOFBF` The file is fully buffered. When a buffer is empty, the next input operation attempts to fill the entire buffer. On output the buffer is completely filled before any data is written to the file.
- `_IOLBF` The file is line buffered. When a buffer is empty, the next input operation still attempts to fill the entire buffer. On output, however, the buffer is flushed whenever a newline character is written to the file.
- `_IONBF` The file is unbuffered. The buf and size parameters are ignored. Each input operation reads directly from the file and each output operation immediately writes the data to the file.

In setbuf, if buf is the constant pointer NULL, I/O is unbuffered, otherwise it is fully buffered. In setvbuf, if buf is the constant pointer NULL a buffer is allocated using malloc using the size parameter as the amount allocated. In setbuf the buffer must be BUFSIZ (specified in stdio.h) bytes long. In setvbuf, the size parameter specifies the buffer size, and must be greater than zero.

A common cause for error is to allocate the buffer as an automatic variable and then failing to close the file before returning from the function where the buffer was declared.

Setbuf produces unpredictable results if it is called for a stream, except immediately after opening the stream or any call to fseek. Calling setbuf after a stream has been unbuffered is legal and does not cause problems.

Include Files

#include <stdio.h>

Return Value

The `setvbuf` function returns non-zero if an invalid value is given for type or size, or when `buf` is NULL if there is not enough space to allocate a buffer.

Portability

Setbuf is available on UNIX systems. Both functions are defined in the ANSI Standard.

Note: For related information, refer to the `fopen`, `fseek`, and `malloc` functions.

setjmp, longjmp

```
int      setjmp(jmp_buf env);
void     longjmp(jmp_buf env, int val);
```

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

Setjmp saves the current stack environment in env for later use by longjmp. It preserves the current stack pointer, register variable, and automatic variable frame pointer as well as the return instruction pointer. It returns zero when it is initially called.

Longjmp restores the environment in env and then returns in such a way that it appears that setjmp returned with the value val. Longjmp cannot return the value 0; if passed 0 in val, longjmp returns 1.

The routine that called setjmp, and set up env, cannot have returned in the interim before calling longjmp. If this happens the results are unpredictable.

Automatic variables and function arguments have values as of the time longjmp was called. Any register variable is restored to the value at the time of the call to setjmp.

Include Files

```
#include <setjmp.h>
```

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to the `ssignal` function.

setmem

```
void    setmem(void *addr, int len, char value);
```

Setmem assigns the byte value to each character in the string pointed to by addr. Len gives the number of characters to assign. This function uses the 8086 stosb instruction and is extremely fast.

Include Files

```
#include    <i8086.h>
```

Portability

This function is unique to the 8086 family.

Note: For related information, refer to the movmem and string functions.

sin, cos, tan, asin, acos, atan, atan2

```
double    sin(double x);
double    cos(double x);
double    tan(double x);
double    asin(double x);
double    acos(double x);
double    atan(double x);
double    atan2(double y, double x);
```

These calls perform trigonometric functions.

Sin, cos, and tan return the corresponding trigonometric functions. Angles are specified in radians.

Asin, acos, and atan return the arc sine, arc cosine, and arc tangent, respectively, of the input value. Arguments to asin and acos must be in the range -1 to 1 . Arguments outside that range cause asin or acos to return zero and set errno to EDOM.

Atan2 returns the arc tangent of y/x and produces correct results even when the resulting angle is near $\pi/2$ or $-\pi/2$ (x near zero).

Include Files

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

Return Value

Sin and cos return a value in the range -1 to 1 .

Tangent returns any value for valid angles. For angles close to $\pi/2$ or $-\pi/2$, tangent returns zero and sets errno to ERANGE.

Asin returns a value in the range $-\pi/2$ to $\pi/2$.

Acos returns a value in the range 0 to π .

Atan returns a value in the range $-\pi/2$ to $\pi/2$.

Atan2 returns a value in the range $-\pi$ to π .

Portability

These functions are available on UNIX systems. These functions are defined in the ANSI Standard.

sinh, cosh, tanh

double sinh(double x);

double cosh(double x);

double tanh(double x);

These functions compute the designated hyperbolic functions for real arguments.

Include Files

```
#include    <math.h>
```

Return Value

Sinh and cosh return a huge value of appropriate sign when the correct value would overflow.

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

ssignal, gsignal

```
int (*ssignal(int sig, int (*action)( )))( );
```

```
int gsignal(int sig);
```

Ssignal and gsignal implement a software signalling facility. Ssignal is used to establish an action routine for servicing a signal. Gsignal is used to raise the signal and execute the action routine.

Software signals are associated with integers in the range from 1 to 15.

The first argument to ssignal is a number identifying the type of signal for which an action is established. The second argument defines the action; it is either the name of a (user defined) action function or one of the manifest constants SIG_DFL (default) or SIG_IGN (ignore). Ssignal returns the action previously established, or if the signal number is illegal, ssignal returns SIG_DFL.

Gsignal raises the signal identified by its argument, sig:

If an action function has been established for sig, then that action is reset to SIG_DFL and the action function is entered with argument sig. Gsignal returns the value returned to it by the action function.

If the action for sig is SIG_IGN, gsignal returns the value 1 and takes no other action.

If the action for sig is SIG_DFL, gsignal returns the value 0 and takes no other action.

If sig has an illegal value or no action was ever specified for sig, gsignal returns the value 0 and takes no other action.

Include Files

```
#include <signal.h>
```

Portability

These functions are available on UNIX systems.

stime

```
int      stime(time_t *tp);
```

Stime sets the system time and date. Tp points to the value of time as measured in BTOS SimpleDate format.

Include Files

```
#include <time.h>
```

Return Value

A value of 0 is returned.

Portability

This function is available on UNIX systems.

Note: For related information, refer to the time function.

strcat, strncat

char *strcat(char *dest, char *src);

char *strncat(char *dest, char *src, int maxlen);

Strcat appends a copy of src to the end of dest. The length of the resulting string is `strlen(dest) + strlen(src)`.

Strncat copies at most maxlen characters of src to the end of dest, and then appends a null-byte terminator. The maximum length of the resulting string is `strlen(dest) + maxlen`.

Include Files

```
#include <string.h>
```

Return Value

Both functions return the first argument.

Portability

These functions are available on UNIX systems, and are defined in the ANSI Standard.

Note: For related information, refer to the `memcpy`, `movmem`, `setmem`, `strchr`, `strcmp`, `strcpy`, `strlen`, `strspn`, and `strtok` functions.

strchr, strrchr, strpbrk

```
char    *strchr(char *s, char c);
```

```
char    *strrchr(char *s, char c);
```

```
char    *strpbrk(char *s1, char *s2);
```

Strchr and strrchr scan a string for a specific character. The null character terminating a string is considered to be part of the string, so that for example:

```
strchr(s, 0)
```

returns a pointer to the terminating null byte of the string.

Strpbrk scans a string for one of several different characters.

Include Files

```
#include <string.h>
```

Return Value

Strchr returns a pointer to the first occurrence of the character *c*, or NULL if *c* does not occur in *s*. Strrchr returns a pointer to the last occurrence of the character *c*, or NULL if *c* does not occur in *s*. Strpbrk returns a pointer to the first occurrence of any of the characters in *s2*, or NULL if none of the *s2* characters occurs in *s1*.

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to the memcpy, movmem, setmem, strcat, strcmp, strcpy, strlen, strspn, and strtok functions.

strcmp, strncmp

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

```
int      strncmp(char *s1, char *s2, int maxlen);
```

strcmp lexicographically compares its arguments up to the terminating null-bytes.

strncmp makes the same comparison but looks at no more than maxlen characters.

Include Files

```
#include <string.h>
```

Return Value

These functions return -1 if s1 is less than s2, 0 if s1 equals s2, and 1 if s1 is greater than s2.

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to memcpy, movmem, setmem, strcat, strchr, strcpy, strlen, strspn, strtok.

strcpy, strncpy

```
char    *strcpy(char *dest, char *src);
```

```
char    *strncpy(char *dest, char *src, int maxlen);
```

Strcpy copies string `src` to `dest`, stopping after the null character has been moved.

Strncpy copies exactly `maxlen` characters, truncating or null-padding `dest`; the target may not be null-terminated if the length of `src` is `maxlen` or more.

Include Files

```
#include <string.h>
```

Return Value

Both functions return `dest`.

Portability

Available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the `memcpy`, `movmem`, `setmem`, `strcat`, `strchr`, `strcmp`, `strlen`, `strspn`, and `strtok` functions.

strlen

```
int      strlen(char *s);
```

Strlen returns the number of non-null characters in s.

Include Files

```
#include <string.h>
```

Portability

Available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the memcpy, movmem, setmem, strcat, strchr, strcmp, strcpy, strspn, and strtok functions.

strspn, strcspn

```
int      strspn(char *s1, char *s2);
```

```
int      strcspn(char *s1, char *s2);
```

Strspn returns the length of the initial segment of string *s1*, which consists entirely of characters from string *s2*.

Strcspn returns the length of the initial segment of string *s1*, which consists entirely of characters not from string *s2*.

Include Files

```
#include <string.h>
```

Portability

Available on UNIX systems. These functions are defined in the ANSI Standard.

Note: For related information, refer to the `memcpy`, `movmem`, `setmem`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strlen`, and `strtok` functions.

strtok

```
char      *strtok(char *s1, char *s2);
```

Strtok considers the string `s1` to consist of a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `s2`.

The first call (with pointer `s1` specified) returns a pointer to the first character of the first token, and writes a NULL character into `s1` immediately following the returned token.

Subsequent calls with zero for the first argument works through the string `s1` in this way until no tokens remain. The separator string `s2` may be different from call to call. When no tokens remain in `s1` a NULL is returned.

Include Files

```
#include    <string.h>
```

Portability

This function is available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the `memcpy`, `movmem`, `setmem`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strlen`, and `strspn` functions.

swab

```
void      swab(char *from, char *to, int nbytes);
```

Swab copies bytes from the from string to the to string for nbytes length. Adjacent even and odd byte positions are swapped. This is useful for carrying data from one workstation to another. Nbytes should be even.

Include Files

There are no include files required with this function.

Portability

This function is available on UNIX systems.

time

```
time_t    time(time_t *tloc);
```

Time returns the value of time in BTOS SimpleDate format.

If tloc is non-zero, the return value is also stored in the location to which tloc points.

Include Files

```
#include    <time.h>
```

Portability

Available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the stime function.

toupper, tolower, _toupper, _tolower, toascii

```
int    toupper(int c);
int    tolower(int c);
int    _toupper(int c);
int    _tolower(int c);
int    toascii(int c);
```

Toupper and tolower convert integers in the range EOF to 255. Toupper leaves the character unchanged, except for lowercase letters which are converted to uppercase. Ttolower similarly converts uppercase letters to lowercase and leaves all others unchanged.

_toupper and _tolower do the corresponding conversions, except that _toupper only works for lowercase letters and _tolower only works for uppercase letters. These are implemented as macros and are therefore much faster than the functions tolower and toupper.

Toascii yields its argument with all bits cleared except the lower seven bits, giving a value in the range of 0 to 127. It is intended for compatibility with other systems.

Include Files

```
#include <ctype.h>
```

Portability

All functions are available on UNIX systems. These functions, except toascii, are defined in the ANSI Standard.

Note: For related information, refer to isalpha.

ungetc

```
int      ungetc(char c, FILE *stream);
```

Ungetc pushes the character *c* back onto the named input stream. This character returns on the next call to `getc` or `fread` for that stream. Ungetc returns *c*.

One character may be pushed back in all situations. A second call to `ungetc` without a call to `getc` forces the previous character to be forgotten.

`fseek` erases all memory of a pushed back character.

Include Files

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

Return Value

Ungetc always returns the character pushed back.

Portability

This function is available on UNIX systems. This function is defined in the ANSI Standard.

Note: For related information, refer to the `fseek` and `getc` functions.

unlink

```
int      unlink(char *filename);
```

Unlink deletes a file specified by the filename. Any BTOS drive, directory, and filename may be used as a filename.

Return Value

On successful completion, a zero is returned. On error a -1 is returned.

Portability

This function is available on UNIX systems.

vprintf, vfprintf, vsprintf

```
int      vprintf(char *format, va_list argp);  
int      vfprintf(FILE *stream, char *format, va_list  
         argp);  
int      vsprintf(char *s, char *format, va_list argp);
```

These functions are alternate entry points for the printf functions. They behave exactly like the corresponding printf functions, except that instead of providing the arguments to be formatted explicitly in the command line, they are supplied in an array pointed to by argp.

The argp parameter is the va_list array filled in by a call to va_start.

Include Files

```
include   <stdio.h>  
include   <stdarg.h>
```

Return Value

The return value is the same as for the corresponding printf function.

Portability

These functions are available on UNIX. These functions are defined in the ANSI Standard.

vscanf, vfscanf, vsscanf

```
int      vscanf(char *format, va_list ap);
```

```
int      vfscanf(FILE *stream, char *format, va_list ap);
```

```
int      vsscanf(char *s, char *format, va_list ap);
```

These functions are alternate entry points for the scanf functions. They differ from the normal scanf functions in that instead of supplying the list of arguments explicitly in the call, a pointer to an array of arguments is supplied.

The ap parameter is actually the array set up by a call to va_start. The ap parameter then points to an array of scanf parameter pointers. The pointers in this array must correspond to the format specifiers in the format string.

Include Files

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

```
#include <stdarg.h>
```

Return Value

The return value is the same as for the corresponding scanf function.

Portability

These functions are available on UNIX.

write

```
int      write(int handle, void *buf, int nbyte);
```

This function writes a buffer of data to the file or device named by the given handle.

Handle is a file handle obtained from a `creat` or `open` call.

This function attempts to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with `handle`. If the number of bytes actually written is less than the number requested, the condition should be considered an error and probably indicates a full disk.

For disk or diskette files, writing always proceeds from the current file pointer (see `lseek`). For devices, bytes are directly sent to the device.

Include Files

There are no include files required for this function.

Return Value

The number of bytes written is returned by `write`. In case of error, `write` returns `-1`.

Portability

`Write` is available on UNIX systems.

Note: For related information, refer to the `creat`, `lseek`, and `open` functions.

The C Programming Language

This section contains a description of the C programming language and emphasizes the language capabilities and advantages.

BTOS C Features

BTOS C includes these features:

- Comments can optionally be nested.
- The keywords `interrupt`, `plm`, `_cs`, `_ds`, `_es`, `_ES`, `_ss`, `near`, and `far` have been added for special hardware support.
- Inline assembly language code, using the keyword `asm`, can be included in C source modules.
- The preprocessor supports `#pragma`, `#error`, and `#` directives.
- The preprocessor supports the `__DATE__` and `__TIME__` macros.
- The preprocessor no longer expands macro arguments inside strings and character constants.
- The preprocessor uses the `#` symbol in front of the argument name in a macro expansion to support 'string-izing' macro arguments.
- The preprocessor uses the `##` symbol between two other tokens to support token-concatenating in macro expansion.
- Comments are replaced with a single space character after macro expansion.
- Nested macros mentioned in a macro definition string are expanded only when the macro itself is expanded. This mostly affects the interaction of `#undef` with nested macros.
- The `#include` and `#line` directives can have macros on them. If a macro is mentioned on the directive line it is expanded before the directive is performed. This allows using macros to define source file names.
- A progression of signed and unsigned types that an integer constant could be, depending on the value and radix of the constant.

- Floating point constants can contain F or L suffixes to specify float or long double constants.
- A keyword, signed, implies the integer object is signed. This is important for char objects. A compile-time switch (-K) does not affect char objects explicitly declared as signed char.
- There is a new floating type, long double. This type is treated exactly like double.
- A type modifier, const, declares constant data objects. You can use this to define ROMable data objects.
- A type modifier, volatile, declares objects that can be modified in unseen ways. These objects include those that can be modified by an interrupt processing routine, and which may not be overly optimized.
- A function prototype is a function declarator that, unlike the normal empty pair of parentheses, contains a list of function parameter types. These types are used in two ways. First, to check the validity of the parameters actually given in a call. Second, to adjust the type of function arguments to match the parameter types in the prototype.
- Extern declarations given inside a function obey proper block scope. The declarations are not recognized beyond the scope of the block in which they are defined.
- The default conversion rules for mixed type expressions have been modified slightly. This affects the type conversions whenever unsigned char is present, or when unsigned int is combined with signed long.
- A series of addressing type modifiers, near, far, _cs, _ds, _es, and _ss can be given with pointer declarations. These modifiers declare the specific addressing the pointer uses in the 8086 architecture.
- Definitions have been added for the following pseudo-variables:

<code>_AX</code>	<code>_BX</code>	<code>_CX</code>	<code>_DX</code>
<code>_SI</code>	<code>_DI</code>	<code>_BP</code>	<code>_SP</code>
<code>_AL</code>	<code>_AH</code>	<code>_BL</code>	<code>_BH</code>
<code>_CL</code>	<code>_CH</code>	<code>_DL</code>	<code>_DH</code>
<code>_CS</code>	<code>_DS</code>	<code>_ES</code>	<code>_SS</code>

You can use these as unsigned int (or unsigned char for the byte registers) variables. You should be careful when you use these names, since most of these registers are not saved across function calls. If a function uses the `_ES` variable, the ES register is saved before each function call and restored on return. Also, except for `_SI`, `_DI`, `_BP`, `_SP`, `_CS`, `_DS`, and `_SS`, the registers are treated as available scratch registers even in the same expression where they are used. You should make sure that you use them in simple statements as much as possible. If you have any doubts about how an expression is generated, have the compiler produce an assembly listing produced by a given expression. That should reveal any problems.

In general, these pseudo-variables are most useful for setting register parameters to non-C routines, like assembly code routines.

- A function type modifier allows interrupt functions that return a value. This change makes the syntax for specifying non-standard functions a little more uniform.
- A series of function modifiers, `near`, `far`, `plm`, and `interrupt` give greater flexibility in creating functions in mixed model and mixed language environments.
- The huge memory model has been implemented.
- All passes of the BTOS C Compiler run in protected mode under BTOS II on a B28, B38, or B39.
- Emulation of UNIX pipes has been implemented.

Translation Phases and Limits

Preprocessor Translations

This pass reads a source file, processing each source line independently. Preprocessor statements are acted upon as they are encountered. Include files are expanded inline. The only factor limiting the nesting level of include files is the maximum number of files which can be open simultaneously. Since this number is 20, the maximum include file nesting depth is 14.

Define macros can be expanded to up to 4096 bytes in length, and there is no particular limit on the number of macro arguments.

Conditional compilation skips lines by replacing the input line by a line containing nothing but white space. All conditional compilation statements must be complete in the source or include file in which they are begun.

Comments are removed.

The output file is a text string fully readable by any text editor. Lines are placed in the output beginning with a '#' character to indicate the correct line number and source filename.

Parser Translations

The parser (CC1.Run) performs all of the C syntax checking of the compiler. Declarations are recorded and subsequently written to an intermediate file. A minimum of memory allocation is performed at this stage. Automatic storage and function arguments are deferred so that intelligent allocation of register variables can be made.

Executable code is written as a series of expression trees, punctuated by jump and label statements. All iteration and conditional statements are transformed into simple tests and jumps.

Constant subexpressions are evaluated and replaced by a single constant. Both integer and floating point values are calculated. Floating point values are computed using some 8087 emulating arithmetic.

A few limited special cases in expressions are reduced in complexity (such as adding or subtracting a constant zero). Conditional statements with a constant test are reduced to either a no-op or an explicit jump.

Optimizer Translations

The optimizer (CC2.Run) makes a few simple reductions in the code. Jumps to jumps are removed. Unreachable code is eliminated. The condition and increment parts of a for loop are moved to the bottom of the loop. The switch case table is moved from the bottom of the switch to the top (eliminating a jump). Two or more identical code sequences are reduced to a single copy.

These optimizations are guaranteed to preserve the exact execution sequence of the program, with the exception that execution is somewhat faster and the code is almost always smaller.

Code Generator Translations

The parser and/or optimizer output a series of expressions. The Code Generator (CC3.Run) reads these expressions and generates code, a single expression at a time. Before code is actually written to the output, enough code is held in order to generate the shortest size jump instructions for the given output file. As much as possible, the assembly language output is designed to match the object code output.

Compiler Limits

Other than the following limits, the compiler makes use of all available memory to hold intermediate tables.

The maximum number of nested include files is 14.

The maximum size of a single macro definition is 4096 characters. The maximum number of nested macros is 100.

The maximum size of a single expression is limited to 1000 expression tree nodes. Each identifier and operator consumes a single node. In addition, each function call argument consumes one extra node. Implicit or explicit conversions consume one node each. Structure member references consume two extra nodes for arrow ($->$) and three for dot ($.$).

The maximum number of cases in a switch is 256.

Preprocessing

The preprocessor (CC0.Run) scans the source text character by character. Lines with an initial '#' character are directives to be processed as encountered. White space can appear before or after the '#' character.

Preprocessor directives normally are limited to a single line. Directives can be continued onto multiple lines by ending all but the last line of a continuation with a backslash (\) character. In general, any source line of a C program can be continued in this way. Thus, string literals can use this technique to continue the literal onto multiple source lines. The backslash and the following newline are removed, effectively splicing the adjacent lines together into a single long source line.

Source File Inclusion

The complete contents of a text file can be included in a source file for compilation by means of the `#include` preprocessor directive. The compiler treats the named file as if it appeared in its entirety in place of the `#include` directive. This directive is of the form:

```
#include    "filename"
```

or

```
#include    <filename>
```

In general, the form with quotation marks is used to indicate application-specific include files, while the angle bracket form is reserved for system-supplied include files. This convention is suggested, but not enforced.

The filename given does not have to be a complete pathname. If one is given, regardless of the filename delimiter used in the directive, the exact pathname is used. If the file is not found, an error message is displayed. If no pathname is given, or an incomplete one is given, the file is searched for in a number of different directories. For either delimiter form, the first location checked is the current default directory. Then, each of the directories given on the command line with the `-I` option is searched. If the named directories do not exist, no error is reported. If the include file cannot be found anywhere in the list of directories searched, an error is displayed.

The pathname, including the delimiters, can be constructed using macro-expansions. However, if a string is enclosed in quotation marks it is not examined for embedded macros. Also, token and string concatenation cannot be used in macros within an include directive.

Trigraphs are replaced inside the pathname, but escape characters are not translated.

Define Macros

The define macros are as follows:

```
#define identifier token-string newline
```

or

```
#define identifier (argument-list) token-string  
newline
```

```
#undef identifier
```

The identifier named in a `#define` directive is defined as either a define string or a macro. If the character immediately following the identifier is a left parenthesis (no white space is allowed) the definition is of a macro. Otherwise, the definition is of a simple define string macro. The definition applies from the line after the directive to the end of the source file even across include files, or until an `#undef` directive is encountered.

The following identifiers may not appear in a `#define` or `#undef` directive: `defined`, `__FILE__`, `__LINE__`, `__DATE__`, or `__TIME__`. These are names possessing special significance to the preprocessor.

The `#undef` directive causes the current definition, if any, of the identifier to be forgotten. In the case of a define string, the identifier is replaced in all subsequent lines by the token and the string given in the directive.

In the case of a macro, the argument-string is a list of identifiers separated by commas. When the macro is actually used, each of the arguments is substituted within the macro. A macro invocation supplies a list of token-strings, separated by commas and enclosed in parentheses. The entire sequence from the macro identifier to the closing parenthesis is replaced with the token-string in the macro definition.

The number of macro arguments given must be correct. Commas can be given in a macro invocation argument only if they are enclosed in parentheses, double quotation marks, or single quotation marks.

When the `#define` directive is processed no macros and define strings are expanded. Then, when the macro or define string is activated, a scan is performed on the token and the string, recursively expanding any more defined names as they are encountered. The following example should clarify this:

```
#define iszero(a)    ((a) == '0')
#define isalnum(a)  (iszero(a) | islower(a))
#define islower(c)  ((c) >= 'a' && (c) <= 'z')
```

```
isalnum(*p)
```

expands to:

```
(((*p) == '0') | ((*p) >= 'a' && (*p) <= 'z'))
```

if then we have the following:

```
#undef islower
```

```
isalnum(*p)
```

expands to:

```
(((*p) == '0') | islower(*p))
```

if then we have the following:

```
#undef iszero
```

```
isalnum(*p)
```

would expand to:

```
(iszero(*p) | islower(*p))
```

Identifiers inside comments, strings, or character constants are never expanded, even if they are the same as defined macros.

Identifiers inside strings or character constants never expand during preprocessing. Since a common feature programmers need is the ability to enclose a macro argument inside quotation marks and display the string at runtime, a macro argument identifier preceded by a `#` is converted to a string by the preprocessor.

Thus, the following macro definition can be used:

```
#define      DEBUG(a)          printf(#a " = %d\n", a)
          DEBUG(x + y);
```

would expand to:

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

Note that the argument is replaced with the overt spelling of the argument, before any macros in the argument are expanded. Strings of white space are replaced with a single space character, including comments.

Two tokens can be concatenated in a macro definition. Two tokens are separated by ## plus optional white space. The preprocessor removes the white space and the ##, effectively combining the separate tokens. Typically, this is used to construct identifiers. It is not guaranteed to be portable to concatenate two things that do not result in a single token distinguishable from any adjacent tokens. For example:

```
#define      VAR(i, j)      (i ## j)
          VAR(x, 6);
```

would expand to:

```
x6;          /* Guaranteed ok */
VAR(x +, 6);
```

would expand to:

```
x +6;          /* Not portable */
```

The second invocation in this example may not port to other systems because the sequence in which a compiler is required to process input allows compilers to verify tokens in their input at an early stage and do the concatenating process later on as a special case. If the two concatenated tokens do not form a single simple token, the compiler may not produce the same results as the BTOS C implementation.

The definition of a replacement string is trimmed of all leading and trailing spaces, so care must be taken when using macros beginning or ending with operators. The compiler can get confused if the macro identifier is used with more operators immediately around it. In general, enclosing the macro expansion string in parentheses eliminates any problems.

__FILE__

This macro is automatically defined to be the current source file being processed. This macro is changed whenever a new `#include` directive is processed, when the include file is completed, or a `#line` directive is processed.

This macro appears as a string in the processed text. Backslashes expand to double backslashes, preventing characters in a string from being improperly interpreted.

__LINE__

This macro is automatically defined to be the number of the current source file line being processed. The first line of a source file is defined to be 1.

__DATE__

This macro is automatically defined to be the date the preprocessor began processing the current compiled source file. Thus, each inclusion of the macro in a source file is guaranteed to contain the same value, even near midnight.

The date appears as “Mmm dd yyyy”, where Mmm is the month (from Jan to Dec), and dd is the day (from 1 to 31) including a leading blank for days less than ten, and yyyy is the year.

__TIME__

This macro is automatically defined to be the time the preprocessor began processing the current compiled source file. Thus, each inclusion of the macro in a source file is guaranteed to contain the same value.

The format of the replacement string of this macro is “hh:mm:ss”, where hh is the hour (from 00 to 23) using a 24-hour clock, mm is the minutes (from 00 to 59) and ss is the seconds (from 00 to 59).

__SMALL__

This macro is defined by the small memory model selection options. The value is 1.

__MEDIUM__

This macro is defined by the medium memory model selection options. The value is 1.

__LARGE__

This macro is defined by the large memory model selection options. The value is 1.

__HUGE__

This macro is defined by the huge memory model selection options. The value is 1.

Conditional Compilation

Six directives are defined that can be used to provide conditional compilation of source text. One or more blocks of text lines are delimited by these directives, an `#ifdef`, `#ifndef`, or `#if` directive to begin the sequence, then a series of text lines to be compiled if the initial condition is true. An `#elif` or `#else` statement may occur next. If the first condition is `#elif`, a subsequent `#elif` condition is tested and if true, the following set of text lines is compiled. If an `#else` directive is encountered and all previous condition directives in the sequence were false, then the set of text lines following the `#else` are compiled. The entire sequence is terminated by an `#endif` statement.

Other preprocessor directives can be nested within a conditional compilation, including more conditional compilation directives. The `#else`, `#elif`, and `#endif` directives associated with a leading `#ifdef`, `#ifndef`, or `#if` directive must appear in the same text file. They cannot be spread across several include files.

When skipping text in a conditional sequence, the lines are examined only to keep track of nesting of conditional compilation directives and filename and line number information.

Constant Expressions

In the following conditional directives, the term “constant expression” applies to any expression involving only integer constants and excluding the sizeof operator and any operators which involve side effects, such as increment, decrement, or the assignment operators. Macros and define strings are expanded by the preprocessor. Any undefined identifiers in the expression after expansion is complete are replaced with a constant 0.

Only constant expressions in the preprocessor can incorporate the following subexpression:

```
defined ( identifier )
```

or

```
defined identifier
```

This operation has the value 1 if the identifier is a defined macro or string, 0 otherwise. Thus, the following pairs of directives are equivalent:

```
#if defined ( XYZ )
```

and

```
#ifdef XYZ
```

```
#if !defined ( XYZ )
```

and

```
#ifndef XYZ
```

```
#ifdef Directive
```

```
# ifdef identifier
```

This directive causes the following text lines to be compiled if the identifier is a defined string or macro. This is considered a true condition. Otherwise, the following text lines are skipped.

```
#ifndef Directive
```

```
# ifndef identifier
```

This directive causes the following text lines to be compiled if the identifier is not a defined string or macro. This is considered a true condition. Otherwise, the following text lines are skipped.

#if directive

if constant expression

This directive causes the following text lines to be compiled if the constant expression evaluates to non-zero, or true. Otherwise, the following text lines are skipped.

#elif Directive

elif constant expression

If all of the preceding conditional directives of a sequence are false, the constant expression is evaluated. If non-zero, the following text lines are compiled, otherwise they are skipped. A non-zero value is considered a true value.

#else Directive

else

If any of the preceding conditional directives in a sequence are true, the text after the #else directive is skipped. Otherwise, the following text is compiled. The #else directive must be the last directive in a conditional sequence to appear before the #endif directive.

#endif Directive

endif

An #endif directive completes a conditional compilation sequence.

Line Number Control

line constant "filename"

line constant

This directive causes its warning messages, the line number, and source file name to be displayed on error. The filename continues until the next #line directive, or the end of the current include file. The line number of the next source line is taken to be the constant, and subsequent lines have progressively higher line numbers until a new #line statement is encountered or the end of the current include file is reached.

This directive is most useful when the compiler is used in conjunction with a front-end language that translates to C. Macros are expanded in this directive.

#pragma Directives

```
# pragma x_char_sequence new_line
```

This directive is intended to supply implementation specific extensions. The BTOS C Compiler recognizes such directives, but does not process them. A comment which begins on this directive can be continued onto another line, but any other text must be continued using the backslash (\), newline convention.

#error Directives

```
# error x_char_sequence new_line
```

This directive causes the preprocessor to terminate immediately. A fatal diagnostic is issued, formatted as follows:

```
Fatal: filename line-no: Error directive: x_char_sequence
```

The text on the directive is scanned to remove comments, but any remaining text is displayed. The text is not examined for embedded macros.

Null Directive

```
# new_line
```

This directive has no effect when processed. No other text can appear between the # and the new_line, except white space and comments.

Comments

Comments can be inserted between any adjacent tokens, and have no effect on the compilation of a C program, except for LINT comments that only affect diagnostic messages displayed.

Comments are begun with the characters /* and continue across text lines if necessary until a */ character sequence

is encountered. Comments are allowed in preprocessor directives, and if a comment begins on the same line as a directive, the comment can continue across multiple lines.

Comments cannot be nested in the standard definition of C and this is the default. A compile time option for the preprocessor is provided which allows nested comments, though it is not recommended where portability is a concern.

Lexical Conventions

Source Text Conventions

A C program is specified as a sequence of ASCII text lines, such as any text file produced by the BTOS Editor or the BTOS word processors. In this documentation we use the term “newline” as if it were a single character separating adjacent text lines. In UNIX and other systems this is true, but under CP/M and MSDOS text lines are separated by carriage–return / line–feed pairs. BTOS C ignores all carriage–return characters in the source program and treats the line–feed characters as newlines.

There are no constraints on where items must appear on a source line. C statements can be spread over as many source lines as desired for readability. The only restriction imposed is that keywords and other identifiers cannot be split across source lines.

White space (spaces, tabs, formfeeds, and newlines) can be ignored and left out in most circumstances. In strings or character constants spaces are significant. Also, spaces must be used to separate adjacent identifiers, such as in the expression ‘sizeof x’ or ‘int i’. White space also cannot be inserted in the middle of an identifier or operator.

Newlines can be placed anywhere between tokens. A newline character by itself cannot be placed inside a character constant or string unit. If you wish to include a newline character in a program data, such as in a string or character constant, you must use the escape sequence `\n`. If you wish to break up a long string across multiple source lines, you must use multiple string units, or you must precede any newlines inside the string with backslashes (`\`). A backslash – newline sequence does not appear at all in a string.

Identifiers

An identifier is a sequence of uppercase and lowercase letters, digits, and the underbar (`_`) character. An identifier must begin with a letter or the underbar. Identifiers can be of any length, but only the first 32 characters are significant. Identifiers which differ only beyond the first 32 characters are considered identical.

An identifier can name a global variable, a function, a function argument, an automatic (local) variable, a structure or union member, an enumeration constant, a preprocessor macro or define string, a typedef, a structure or union tag, or a goto label. An identifier can be used only within its scope. There are three kinds of scope defined in BTOS C: file, function, and block scope.

Identifiers declared in an external declaration have file scope. An identifier with file scope can be used from the point of the declaration to the end of the file being compiled. These identifiers include global variables, functions, structures and members, typedefs, and enumerations.

Be careful when you see the term external declaration. In C there is a confusion in terminology. An external declaration is simply a declaration that occurs outside any function. Such a declaration can define an object, can declare a typedef, or can declare an external object. The keyword `extern` is normally used to declare an external object. An external object is one defined in a different source file (and defined there in an external declaration). The `extern` keyword can be used in two other special circumstances: inside a function to declare an object that is defined in an external declaration in the same source file, or to declare an object that is defined later in the file in an external declaration. In each of these last two cases, the definition can be either global or static.

Identifiers declared as goto labels are the only identifiers with function scope. These identifiers can be used anywhere in the same function in which the label is located.

Identifiers with block scope include the same array of objects and entities as in file scope. Block scope identifiers are declared at the beginning of a block compound statement. These identifiers can be used from the point of

declaration to the end of the block in which they are defined. In addition, the arguments to a function have block scope extending to the end of the main block of the function. Uppercase and lowercase letters are distinct for all local variables, structure field names, and preprocessor define names. Thus, two structure members, 'abc' and 'AbC' are distinct.

The compiler itself treats uppercase and lowercase letters as distinct in global variables, but the BTOS linker does not distinguish the two cases. Any symbol passed to the linker is converted to uppercase. The linker indicates multiply defined symbols if two identifiers differ only in the case of the letters. Thus, the symbols 'strcpy' and 'Strcpy' are the same to the linker.

Keywords

The following list of identifiers are the keywords of C. These identifiers cannot be used to name any variable.

asm	auto	break
case	char	const
continue	default	do
double	else	enum
extern	far	float
for	fortran	goto
if	int	interrupt
long	near	plm
register	return	short
signed	sizeof	static
struct	switch	typedef
union	unsigned	void
volatile	while	_cs
_ds	_es	_ss

Numerical Constants

C allows different kinds of numerical constants in a source program: integer, character, escape, and floating.

Integer Constants

Integer constants can be specified in decimal, octal, or hexadecimal. Normally, integer constants have type `int`, but large valued integers, or constants with a trailing letter `L` have type `long`.

Integer constants can be any length, but if the constant overflows the size of a long integer there is no warning given. Negative integer constants cannot be directly specified. A source file containing, for example, a `-34` is actually specifying a positive constant of 34, which is then negated with the standard negation operator. Such a construction is treated as a constant expression evaluated at compile time.

Any integer constant can be followed by an uppercase or lowercase letter `L`, and/or an uppercase or lowercase letter `U`. The `L` suffix forces the constant to have long type, regardless of the magnitude of the constant. The `U` suffix forces the constant to have unsigned type, regardless of the magnitude of the constant.

A decimal integer constant is any string of digit characters beginning with a non-zero digit. An unsuffixed decimal constant that is greater than 32767 is treated as long, greater than 2147483647 is treated as unsigned long, and greater than 4294967295 overflows without warning and the resulting constant is the low-order bits of the actual value.

An octal integer constant is a string of octal digits (0 through 7) beginning with a zero. An unsuffixed octal constant that is greater than 077777 is treated as unsigned, greater than 0177777 is treated as long, greater than 017777777777 is treated as unsigned long, and greater than 037777777777 overflows without warning and the resulting constant is the low-order bits of the actual value.

A hexadecimal constant begins with a `0x` or `0X` and contains a string of digits, the lowercase letters `a` through `f`, and uppercase and letters `A` through `F`. An unsuffixed hexadecimal constant that is greater than

0x7FFF is treated as unsigned int, greater than 0xFFFF is treated as long, greater than 0x7FFFFFFF is treated as unsigned long, and greater than 0xFFFFFFFF overflows without warning and the resulting constant is the low-order bits of the actual value.

A constant suffixed with a U is treated as unsigned or unsigned long. If the constant is greater than 65535, the constant is treated as unsigned long, regardless of the radix of the constant.

Character Constants

A character constant is a mechanism for specifying integer values that correspond to ASCII characters. A character constant is a string of ASCII characters enclosed in single quotation marks, such as 'x' or 'gh'. A character constant is always an int quantity and is exactly equivalent to an integer constant with the same value. Character constants cannot contain newline characters. A double quotation mark can be included in a character constant, but a single quotation mark must be given as an escape sequence.

A character constant can be one or two characters long in this implementation. Character constants longer than one character are not portable and should be avoided. In BTOS C the first character is placed in the low-order byte of the resulting integer, and the second character in the high-order byte. If only one character is given, then the high-order byte is zero.

Escape Sequences

In order to specify non-printable characters, escape sequences are provided. All escape sequences begin with a backslash character (\). The simplest escape sequences consist of a backslash followed by a single letter. The following table identifies each escape sequence and the resulting character in hexadecimal and its ASCII abbreviation.

Sequence	Value	Character	Name
<code>\a</code>	0x07	BEL	Audible bell
<code>\b</code>	0x08	BS	Backspace
<code>\f</code>	0x0C	FF	Force
<code>\n</code>	0x0A	LF	Newline
<code>\r</code>	0x0D	CR	Carriage return
<code>\t</code>	0x09	HT	Horizontal tab
<code>\v</code>	0x0B	VT	Vertical tab
<code>\\</code>	0x5C	<code>\</code>	Backslash
<code>\'</code>	0x2C	<code>'</code>	Single quotation mark
<code>\"</code>	0x22	<code>"</code>	Double quotation mark
<code>\?</code>	0x3F	<code>?</code>	Question mark

In addition, octal escape sequences can be given by following the backslash with from one to three octal characters. A hexadecimal escape sequence can be given by specifying a backslash, letter x (or X) followed by one to three hex digits. Any 8-bit ASCII character value from 0 to 255 can be specified.

Valid escape sequences include `\n`, `\v`, `\004`, `\x0A`. A character constant of `'\058'` would be treated as two characters: an octal 05 and an ASCII '8', since 8 is not a legal octal digit.

Floating Constants

A floating point constant is given as a mantissa followed by an optional exponent and optionally terminated with a type suffix. The mantissa consists of a string of digits with a single decimal point. Legitimate mantissas can have the decimal at the start or end of the digit string, or embedded in the middle of digits. The exponent begins with an uppercase or lowercase letter E, followed by an optional sign (plus or minus) and then a string of digits. The type suffix can be an F or f, indicating float type, or an L or l, indicating long double type.

Floating point constants can contain up to 15 significant decimal digits in the mantissa and can have an exponent in the range plus or minus 306. Exponents beyond the range given cause overflow, but no error messages are displayed.

Strings

A string consists of one or more string units. Each string unit is a sequence of ASCII characters and/or escape sequences enclosed in double quotation marks. A newline cannot occur in the middle of a string unit. A string unit can be any length.

Multiple string units can be given, possibly on several source lines. The units are concatenated, so that long strings may be split among several source lines. For example:

```
“hello world, this is an example of ”
“a long string\n”
“spread across source lines\n”;
```

Prior to the ANSI C Standard, this same string would be declared as:

```
“hello world, this is an example of \
a long string\n\
spread across source lines\n”
```

Operators

An operator specifies an operation to be done that produces a value. The operators of C are built, for the most part, of punctuation and other special ASCII characters. `sizeof` is the only C operator that is a keyword. Operators consisting of more than one character must not be divided by white space. If more than one operator is given with no intervening spaces, the compiler chooses the longest operator to fit the initial sequence of characters. For example, `y---z` is interpreted as `y -- - z`.

The C operators are:

<code>++</code>	<code>--</code>	<code>&</code>	<code>!</code>	<code>%</code>	<code>^</code>
<code>*</code>	<code>-</code>	<code>+</code>	<code>=</code>	<code>==</code>	<code>!=</code>
<code><</code>	<code>></code>	<code><=</code>	<code>>=</code>	<code><<</code>	<code>>></code>
<code>%=</code>	<code>^=</code>	<code>&=</code>	<code>*=</code>	<code>--=</code>	<code>+=</code>
<code><<=</code>	<code>>>=</code>	<code>?:</code>	<code>,</code>	<code>/</code>	<code>/=</code>
	<code>-></code>	<code>sizeof</code>			

Punctuation

Punctuation is used to delimit or separate components of a C declaration or statement without causing any actual operations to be performed. The punctuation marks used with C are:

[] () { } , ;

Note that [], (), and { } occur only in matched pairs.

Trigraphs

To support portability to workstations without certain ASCII characters in their native character set, these trigraphs were added to the language. They should be thought of as substitutes for standard characters. Each trigraph begins with a ?? pair, since this sequence is not used in normal C programming. If a sequence of three characters beginning with ?? is encountered that is not in the following list, no translation is done on the characters.

The legal trigraphs, with the standard ASCII equivalents are:

??=	#
??([
??/	\
??)]
??'	^
??<	{
??!	
??>	}
??-	~

These trigraphs are recognized, but we do not recommend using them unless the corresponding C character is not available on your workstation.

Types

Every data object in C has a type. In addition, operations in expressions produce a value which also has a type. The type of an object or expression determines the operations allowed on it. For a data object the type also determines how much memory the object uses.

Basic Types

The following basic types are supported by this compiler:

char	unsigned char
short	unsigned short
int	unsigned int
long	unsigned long
float	double
long double	enum
void	

Integral Types

The basic types correspond closely to the kinds of data manipulated by the 8086 hardware. Char, short, int, and long types, plus the corresponding unsigned types, and enum (enumerations) are called the integral data types. Char, short, int, and long are considered distinct types as far as the abstract definition of C is concerned, but on most workstations int type objects are the same as either short or long objects. In this implementation int and short objects are the same.

Characters and Integers

Character data in the BTOS C Compiler implementation is treated as signed by default, in the range -128 to 127. Unsigned character data falls in the range 0 to 255. A character constant specified with an octal or hexadecimal escape sign extends (unless the -K compile option is given), so the value '\377' given in their book is represented as an integer with the value of -1.

Unsigned

Unsigned quantities can be 8, 16, or 32 bits wide, just as signed integral values are. All of the bits of the object are magnitude bits. All unsigned values are numerically non-negative and span a range of positive integers twice that of signed integers occupying the same number of bytes.

Floating Types

Float, double, and long double types implement the IEEE floating point standard data format used by the 8087 NCP. Float objects occupy 4 bytes, double and long double objects occupy 8 bytes. Long double and double types are treated identically.

Void Type

Void type only occurs in limited situations and is used to signify that the operation produces no value at all. This most commonly occurs when calling a function which returns nothing. No actual objects are of void type.

Void functions are provided as a means for explicitly documenting a function which does not return a value.

As a special construct, to explicitly indicate that the return value of a function is being ignored, an expression can be cast to void type. If the expression is a function call to a function that returns a type, LINT processing reports that the return value is being ignored unless such a void cast is present.

Enumerated Type

Enumerated type data is used to describe a discrete set of alternative values. In BTOS C, enumerated type data can be used without restriction as if it were an integer. The compiler does not check whether an enumeration constant is being assigned to a correct enumerated variable.

Composite Types

In addition to the basic types, the following composite types are supported:

pointer to data	pointer to a function
function	array
structure	union

Pointers

The two classes of pointers are pointers to data and pointers to functions. The full range of pointer arithmetic operators is available to these data pointers. Pointers to functions are restricted in their use. Function pointers can only be copied or used to call the function pointed to. Function pointers cannot be assigned to data pointers and vice versa, though they can be cast appropriately.

Part of the reason for strongly distinguishing these varieties of pointers is that, depending on the memory model used in compiling a program, pointers are of different sizes, and data pointers and function pointers cannot be the same size as each other. Note that addressing modifiers override the size of a pointer given by the memory model in effect.

A pointer type must be declared to point to some other specific type. A pointer can point to any other type. This implementation allows you to mix pointers to different kinds of data objects to be compared or assigned to one another, but you are warned when this happens.

A special pointer declaration, a pointer to void, is allowed. This does not mean a pointer to nothing. A pointer to void is a mechanism adopted to avoid defining a new language keyword. It means a pointer to any kind of data object, the type of which is not necessarily known. You can assign any pointer to a void pointer and vice versa without a cast. You cannot use the indirection operator with a void pointer.

Functions

Functions are declared to return a specific type when called. Functions can return any type except arrays, or other functions (although returning a pointer to a function is allowed).

As an extension introduced by BTOS C, a function declaration can include a language specifier following the function name. The allowed language specifiers are:

- **interrupt** for interfacing with workstation interrupts.
- **plm** for interfacing with PL/M modules.
- **near** for functions using near returns and calls.
- **far** for functions using far returns and calls.

Only one language modifier can be given for each function declarator, except that near or far can be combined with plm. The altered calling conventions have different effects on a program (refer to Function Calls). Function calls must match the language given with the function definition.

Interrupt functions are designed to be used with the 8086 interrupt vectors. An interrupt function saves all registers on entry and restores them on return. A far pointer to an interrupt function can be stored directly into an 8086 hardware interrupt vector, although under BTOS the SetIntHandler procedure should be used instead (refer to your BTOS documentation).

Arrays

Arrays are declared to be a collection of objects of a single type. Arrays must be declared to be a fixed size at compile time. An array occupies exactly the size of each object in the array times the number of objects. No extra information is maintained to check the bounds of the array at runtime. Arrays can be constructed from any type except void or a function type (but again pointers to functions are allowed). C declares multidimensional arrays by constructing an array of arrays as needed.

Structures and Unions

Structures and unions are similar to each other. A structure allows the definition of a set of named members, each with its own type. A structure takes up as much memory as the sum of the sizes of the members. If the structure is compiled with alignment selected (option -a), padding bytes are added as needed to ensure that any non-char members are an even offset from the beginning of the structure.

Unions differ from structures in that the value of a union is only one of the individual members at a time. The members of a union are stored on top of one another so that all members have an offset of zero. The size of a union is the size of the largest member. If alignment is selected, a union is padded to make an even size in bytes. Assigning a value to one member of a union and then extracting the value of a different member can produce unpredictable and certainly non-portable results.

Bitfields

Structures, but not unions, can contain bitfields. Bitfields normally allow packing of several objects into a single workstation word. A bitfield can be either a signed or unsigned int type, and can occupy from 1 to 16 bits. Objects in the BTOS C implementation occupy a whole number of bytes. Other implementations can have different rules for allowed bitfield types and sizes.

Type Modifiers

The type modifiers are `const` and `volatile`. Any object can be declared with these modifiers. The effect of the `const` modifier is to prevent any assignments to the object, or other side effects such as increment or decrement. A `const` object can be initialized, even an auto `const` object.

`Volatile` objects have a less obvious effect. `Volatile` objects can be modified in some invisible way, such as by an interrupt routine, so the compiler is directed that these objects cannot be specially optimized. These declarations only affect a program if it is compiled using the `-Z` optimization option. `Volatile` objects are always treated as if no `-Z` optimization were in effect.

Pointers to `const` or `volatile` objects can be declared. Without a cast, a pointer to a non-`const` object can be assigned to a pointer to a `const` object, but not the other way around. Similarly, a pointer to a non-`volatile` object can be assigned to a pointer to a `volatile` object, but not the reverse. Any pointer can always be explicitly cast to any other pointer type.

The pointer type modifiers are `const`, `volatile`, and the addressing modifiers.

The `const` modifier states that the pointer cannot be modified, though the object pointed to can be.

The `volatile` modifier states that the pointer itself is volatile.

Addressing modifiers have been introduced by BTOS C as an extension to support the complex addressing capabilities of the 8086 family of microprocessors. The addressing type modifiers are:

- default memory model addressing (no modifier)
- near pointer
- far pointer
- `_ds` pointer (same as near)
- `_es` pointer
- `_ss` pointer
- `_cs` pointer

Note that `_ds` is fully equivalent to near for data pointers in the small and medium memory models. Similarly `_cs` is fully equivalent to near for function pointers in the small memory model. Data pointers in the large and huge memory models are equivalent to far pointers. Function pointers in the medium, large, and huge memory models are equivalent to far pointers.

Only one addressing qualifier can be present in a single pointer. (Obviously, if a declaration contains more than one pointer declarator, each can have its own addressing modifier). Use addressing modifiers sparingly, since understanding how they interact can be obscure at times.

Declarations

All declarations, whether of global variables, functions, structure members, or local variables use the same general syntax. The purpose of a declaration is to do one of the following:

- define a structure, union, or enumeration
- define an alias for some type (A typedef)
- define a data object
- define a function
- describe a global data object or function defined in some other source file

In addition to the components of a declaration, the context of the declaration is important in determining the exact nature of the objects being declared. There are six contexts in which a declaration can occur:

- external context (outside of any function, structure, or union definition)
- structure context (inside a structure)
- union context (inside a union)
- formal parameter context (after the function declarator and before the function code block)
- block context (at the beginning of any block)
- function prototype context (inside a function declarator)

You begin with a storage-class. Then you can follow that with a declarator and possibly an initializer. A declaration can be as simple as a type by itself. The allowed combinations of storage-class specifiers, types, declarators, and initializers depends on the context.

At least one of the type and storage-class specifiers must be present in a declaration and/or type specifier.

If a storage-class is allowed in a given context it can be omitted. If it is omitted, a default storage-class is chosen depending on the context. If a type is omitted, the default type of `int` is supplied.

A declarator is used to name the identifier being declared and provide any additional type information such as pointer, array, or function types. The syntax of a declarator was designed to mimic the use of the identifier. The rule of thumb is that if an identifier is declared with some declarator, the same string of tokens in an expression has the same type as the type specifier of the declaration. Thus, in the following example `ip` is declared to be an array of five pointers to integers. The expression given produces a value of type `int`

```
int      *ip[5];  
        ...  
        *ip[3];
```

After a declarator, external and block declaration contexts allow an initializer to be supplied. This is a value to be assigned to the object being declared. For simple, scalar

objects, the initializer is an expression. For arrays or structures, an initializer is a set of expressions, as many as one for each member of the composite object.

More than one declarator can be given in a single declaration by separating the declarators with commas. Thus, in the following declaration, *i* is an int, *j* is a pointer to an int initialized to the address of *i*, and *k* is an array of seven integers:

```
int    i, *j = &i, k[7];
```

In general, you can refer to an object only after the object has been declared in a file. In some situations you may prefer or be required to define an object, most often a function, after the first reference to the object. To do this, you must give a forward declaration for the object. This is only allowed for externally defined objects. A forward declaration is simply the declaration of an object occurring early in a source file and followed in the same file by the definition of the object. Both the forward declaration and the subsequent definition can include the static keyword.

Storage Class Specifiers

Only one storage class specifier can be given in a single declaration. Structure or union member declarations cannot have a storage class specifier. External definitions can have extern, static, or typedef storage class specifiers. Formal parameter declarations can only have the register storage class. Block declarations can have any of the storage class specifiers.

The storage class specifiers are:

Auto

The automatic (auto) storage class is the memory local to a function, created on entry to the function and destroyed on exit. This memory class is implemented on most microprocessors as a stack, and this is true of the 8086 implementations. Auto variables can only be declared in block declarations. If no storage class specifier is given in a block declaration, the storage class is set to auto.

Extern

Extern storage is the storage class of objects defined in the external definition context. These objects are defined in one source file, and can be referred to in other source files. The extern keyword is not used with the defining occurrence of an object. In any referring declaration of a data object, the extern keyword must be given. Referring declarations to functions can omit the extern keyword. A declaration containing the extern keyword can appear as an external declaration or as a block declaration.

In the defining source file for an object, the object must be declared with no storage-class specifier in an external definition. In all other source files referring to an external object, the storage class specifier extern must be included in a declaration for the object.

Any declaration of a function, except for the function definition itself, is implicitly considered to have the extern storage class, even if the keyword is absent.

Register

Register variables are automatic or parameter objects which are kept in high-speed workstation registers during the execution of the program. Like automatic storage, register variables are created on entry to a function and destroyed on return. The number of register variables is limited by the hardware. In the BTOS C implementation, two register variables are supported. If you do not provide register keywords in a function, the compiler chooses two register variables for you.

Any formal parameter, function prototype, or block declaration can include the register storage class specifier. The only limitation in the use of register variables is that they have no workstation address, so the & operator cannot be applied to a register variable. Within this constraint any integer, short, or pointer object can be placed in a register. Depending on the memory model used, only pointers which are two bytes wide can be placed in a register.

In register declarations beyond the first two, or on a declaration of an object that cannot be placed in a register, the register keyword is ignored. If the register keyword is being ignored the declared object has auto storage class.

Static

Static declarations can be given in external definitions or in block declarations. Static objects exist for the duration of a program. The static keyword in an external definition limits the scope of the object so that it can not be referred to in another source file. Static objects declared in a block are known locally to the block containing the declaration.

Typedef

Typedef is not a true storage class. Any identifier declared as a typedef can be used afterward in the source file as a synonym for the type of the declaration. A typedef can occur in external or block declarations.

Type Specifiers

Basic Arithmetic Type Specifiers

The seven keywords char, short, int, long, unsigned, float, and double are used to declare the arithmetic data types. These keywords can be given in any order and/or combinations. You can use the list of combinations and their duplicates as follows:

char	
signed char	(same as char if -K not used)
unsigned char	(same as char if -K is used)
short	
signed short	(same as short)
unsigned short	
short int	(same as short)
unsigned short int	(same as unsigned short)
int	
signed int	(same as int)
unsigned int	
long	
long int	(same as long)
signed long	(same as long)
unsigned long	
unsigned long int	(same as unsigned long)

float
long float (same as double)
double
long double

Structures and Unions

The word structure in this subsection refers to both structures and unions, unless specifically stated otherwise.

Structures are defined by using the following:

- the keyword struct or union
- an optional identifier called the structure or union tag.
- the definitions of each of the members enclosed in curly braces

Structures may be referred to by using the tag identifier.

Named structures can be referred to by giving a type specifier of the struct or union keyword followed only by the tag of the structure.

Each distinctly named structure is a distinct type. No two structure definitions in the same scope can have the same tag. In addition, all references to a tag in subsequent declarations must use the same struct or union keyword used in the original definition.

Unnamed structures are each distinct types, even if two structures have identically defined members.

The members of a structure are defined as a sequence of declarations, each of which can be any previously defined type, except the structure type being declared.

Normally, a member declaration is a normal declaration with a type and a list of one or more declarators, separated by commas. Storage class specifiers and initializers are not allowed in a member declaration.

A structure type can be declared containing only the tag identifier before the tag has been defined. This is only allowed in situations where the size of the structure is not needed. By the time the size is needed, the structure tag must have been defined. Typedefs defining a synonym for the structure, pointers to a structure, or declarations (but not definitions) of functions returning a structure are the

allowed declarations of an undefined structure tag. In declaring a function returning a structure, the tag must be defined before the function is called or defined.

A structure or union must not contain itself, but can contain a pointer to itself.

Enumerations

An enumeration can be defined in a manner similar to that of defining a structure. The **enum** keyword is followed by an optional name, followed by a list of enumerator specifiers separated by commas and enclosed in curly braces.

An enumerator specifier is either a simple identifier, or an identifier followed by an = and a constant expression. For example, the following are valid enumeration definitions:

```
enum day_of_week { sunday, monday, tuesday,  
wednesday, thursday, friday, saturday }
```

```
enum coins { cent = 1, nickel = 5, dime = 10,  
quarter = 25 }
```

The identifiers in enumerator specifiers are defined to be integer valued constants (for all intents and purposes int type objects). BTOS C does not restrict the use of enumerators to objects. In effect, an enumeration is a method for documenting a related set of constants and could be defined (a little less conveniently) using a preprocessor #define statement.

If no enumerators with = appear, the first identifier is given the value 0 and each subsequent identifier increases in value by 1. An identifier with = is assigned the value of the constant expression and subsequent identifiers continue the progression by 1 from the assigned value.

Void

The only declarations permitted using the void type specifier are functions returning void and pointers to void. For example, to declare a function `f` returning void, use the following:

```
void    f(...)
        ...
        {
        ...
        }
```

To declare a pointer to void, for example:

```
void    void *pdata;
```

Declarators

Declarators are probably the most confusing and least understood aspect of C. Normally when one is discussing C types, one uses an English description like 'array of pointers to functions returning a pointer to a function returning long'. In C, declaring the variable `x` to be this type is:

```
long    ((*x[ ])( ))( );
```

This declaration is not at all clear. In fact, it is quite an exercise to figure out which pair of empty parentheses corresponds to which 'function' in the English-form type description.

The simplest form of declarator is an unadorned identifier. In this form, the identifier is declared to be of the type given by the type specifier in the declaration.

There are three basic type declarators which can accompany an identifier: (1) A `*` and perhaps some pointer type modifiers before an identifier declares a pointer to some type. (2) A possibly empty pair of parentheses possibly preceded by language modifiers after an identifier declares a function returning some type. (3) A pair of square brackets, either empty or containing a constant expression, following an identifier declares an array of some type.

Extra parentheses can be used to force grouping of declarators. For example:

```
int  a;      /*a is an integer          */
int  *a;     /*a is a pointer to an integer */
int  a( );   /*a is a function returning int */
int  a[5];   /*a is an array of 5 integers  */
```

When more than one type declarator is used in a single declarator, you must use an inside-out reading rule. Beginning with the identifier, read each of the function or array modifiers from left to right. After any such modifiers, then read from right to left the pointer modifiers. Parentheses can be used to group type declarators and force precedence. Always read all the modifiers inside a given set of parentheses before working outside.

Applying the inside-out reading rule to the declaration:

```
long    ((*x[ ])( ))( );
```

the square brackets ([]) declare an array and are read first. Then the innermost * corresponds to the first pointer. So, (*x[]) declares an array of pointers. The first empty pair of parentheses () corresponds to the first function in the description. Then the leftmost * corresponds to the second pointer in the description. At this point you have an array of pointers to functions returning pointers. Then you read the rightmost pair of empty parentheses (), and finally the type specifier long, which complete the type description.

Therefore, we may conclude that x is an array of pointers to functions returning a pointer to a function that returns a long.

Pointer

A declarator beginning with a * optionally followed by pointer type modifiers declares a pointer. The following examples clarify the use of type modifiers.

```
char          *cp;
```

This specifies a pointer to a char (the memory model determines the size).

int *** far cp;**

This specifies a far pointer to an integer.

const char ***cp;**

This specifies a pointer to a constant char.

long *** far const xp;**

This specifies a constant far pointer to a
long.long **(* near q) plm ();**

This specifies a near pointer to a plm function returning a
long.

int **(* near g) far ();**

This specifies a near pointer to a far function returning an
int.

Function

A function declarator is a declarator with a trailing optional set of language modifiers and a pair of possibly empty parentheses.

A function declarator with an empty parenthesis declares a function with no parameter information given. This is the only allowed function declarator.

All other legal function declarators are function prototypes. These are declarators that include information about the function parameters that are used by the compiler to check function calls for validity.

A function declarator with a parenthesis containing the single keyword `void` indicates a function that takes no arguments at all.

Otherwise, the parenthesis of a function declarator contains a list of parameter types separated by commas. These types can be any type allowed in a function parameter. The type declaration of a parameter can include a declarator. The type can be in the form of a cast and not include an identifier, or an identifier can be included. If an identifier is included it has no effect except to be included in the diagnostic messages when a parameter type mismatch occurs.

A function prototype normally defines a function accepting a fixed number of parameters. For C functions, such as

printf, that accept a variable number of parameters a function prototype can end with an ellipsis (...). The fixed initial function parameters are included with their types and, after the last fixed parameter, the prototype is finished with a comma and ellipsis. With this form of prototype, the fixed parameters are checked at compile time and the variable parameters are passed as if no prototype were present.

The following examples should clarify this:

```
int    f( );
```

This specifies a function returning an int with no parameter information.

```
int    p(int, long);
```

This specifies a function returning an int that accepts two parameters, the first an int and the second a long.

```
int    q plm(void);
```

This specifies a plm function returning an int that accepts no parameters at all.

```
char   *s far(char *source, int kind);
```

This specifies a far function returning a pointer to a char, and accepting two parameters, the first a pointer to a char and the second an int. The names **source** and **kind** appear in diagnostics if there is a parameter type mismatch.

```
int    printf(char *format, ...);
```

This specifies a function returning an int and accepting a pointer to a char fixed parameter and any number of additional parameters of unknown type.

```
int    (*fp)(int);
```

This specifies a pointer to a function returning an int and accepting a single int parameter.

Array

An array declarator is a declarator with a trailing pair of square braces, possibly enclosing an integral constant expression. The constant expression that specifies the size of the array must have integral type and a value greater than zero. It must be present, except that the first size may be omitted when an array is being declared as a formal

parameter of a function, or when the array declaration has storage-class specifier `extern` and the definition that actually allocates storage is given elsewhere. It may also be omitted when the declaration is followed by initialization; in this case, the size is determined by the number of initializers supplied.

These examples can help:

```
double    d[5];
```

This specifies an array of five doubles.

```
char     *p[4];
```

This specifies an array of four pointers to chars.

```
int      x[5][6];
```

This specifies a two-dimensional array of five rows and six columns of ints.

```
long     z[ ][7];
```

This specifies a two-dimensional array with an unspecified number of rows and seven columns of longs. This declaration is only legal if it is given as the declaration of a function parameter (since it is converted to a pointer), or if the declaration includes an `extern` keyword (since it does not reserve storage), or if an initializer is included in the declaration. In the last case, the unspecified dimension of the array is determined from the number of initializers.

Bitfields

For structures only, not unions, special bit field declarations are allowed. These can be declared to be of `int` or `unsigned int` type only. The number of bits in a bit field is given by following the declarator of the field with a colon followed by a constant expression in the range from 1 to 16. Dummy, padding fields can be defined by giving a type, followed immediately by a colon and a constant expression in the range from 1 to 16.

Unnamed bitfields ensure that the specified number of unused bits are reserved within a word. This is useful for defining externally specified bits in such things as hardware status bytes.

Bitfields are allocated from low-order to high-order bit within a word. Thus, the following declarations are allocated as shown:

```

struct      a      {
    int          i : 2;
    unsigned     j : 5;
    int          : 4;
    unsigned     k : 4;
};

```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
x		x	x	x		x	x	x	x	x	x	x	x	x	x	
		<- k ->				<- unused ->					<- j ->				<- i ->	

You can force the next bit fields in a sequence to align to the next word by including a bit field declaration with a width of 0.

Integer fields are stored in twos complement form with the left-most bit being the sign bit. A signed integer bit field one bit wide can only hold the values -1 and 0, for example.

Type names

A declaration with storage class typedef defines a typedef. Subsequent declarations can then use the identifier just declared as a synonym for the original type. For example, the following declarations are legal:

```

typedef struct { double real, imaginary; } complex;
static complex z[5];

```

The effect of a typedef in a declaration is as if you were to substitute the full type of the name into the declaration. In general, there is no difference between using a typedef name and explicitly including the full type in a declaration. The exception is that a structure with no tag in a typedef produces the same type in all uses of the typedef, but the two tag-less structures separately defined are considered distinct types. An example is the complex typedef. All complex objects are of the same type.

Type Equivalence

Two types are equivalent if, after substituting the definitions of all typedefs in the types, they are the same. Named structures, unions, or enums are considered the same if the tags are the same. Unnamed structures, unions, or enums are the same in two different types only if the structure, union or enum was defined under some typedef common to the expansion of the two original types.

Thus, in the following example, types `dog` and `cat` are the same, but `bird` is not.

```
typedef struct    { int legs; int ears; } body;

typedef body     dog;
typedef body     cat;

typedef struct    { int legs; int ears; } bird;
```

Initialization

An initial value can be given for block defined objects or externally defined objects. After the declarator for a given object, an initializer is represented by an `=` followed by the initializer. Unions cannot be initialized.

The simplest initializer is a simple expression. When initializing arrays or structures, the initializer can be a list of other initializers enclosed in curly braces and separated by commas. The array elements or structure members are given in increasing element or member order. If an aggregate object contains subaggregates, this rule applies recursively to the members of the aggregate.

Curly braces can be omitted in an initializer if the initializer begins with a left brace. Then the succeeding comma-separated list of initializers initializes the members of the aggregate. If however, the initializer does not begin with a left brace, only as many initializers as needed to fill the aggregate are used, and any extra initializers are used

to initialize the next element of the parent aggregate of which the current aggregate is a part. For example, the following is a completely enclosed initializer:

```
double z[4][3] = {  
    { 1.5, 2.4, 3.3 },  
    { 7, 5.0, 6.2 },  
    { 45.55, 1e4, 100.0 },  
};
```

Note that in the preceding example, the list of initializers can have a trailing comma. This comma has no effect but is allowed for convenience in editing tables of initializers.

This initializer can be abbreviated as:

```
double z[4][3] =  
    { 1.5, 2.4, 3.3, 7, 5.0, 6.2, 45.55, 1e4, 100.0 };
```

Note that the fourth row of the above array is set to zero.

If too few initializers are given to name all the members of an aggregate, the remaining members are set to zero.

Static or global objects must be initialized to arithmetic constant expressions, or to the address of a static or global object or function plus or minus a constant offset. Static initializations are stored at compile and link time into the executable file of the program and are present when the program is started. These values can be subsequently changed during the execution of the program, unless the object being initialized is declared with a `const` type modifier.

Scalar (automatic or register) objects can be initialized to any valid expression. Auto aggregate objects can be initialized in exactly the same manner as static aggregates. Auto aggregate initializers must be constant, or an address of a static object plus or minus a constant. The initialization occurs at the entry to each block as if an assignment statement were being executed. Auto aggregate initializations are accomplished by making a static image of the initializer and then copying it into the auto on entry to the block. If a block is entered via a `goto` or a `switch` statement, the initializers are not executed.

When initializing a scalar object, such as an int or a pointer, a single expression is given, possibly enclosed by curly braces. The initial value of the object is the value of the expression. The same conversions as for assignment are performed.

When an initializer is present in the definition of an array, the size of the array can be omitted. The size is set to the number of elements in the initializer. For example:

```
int    x[ ] = { 1, 3, 7 };
```

In this example, x has type “array of three integers.”

When initializing an array of characters you can use a string literal as a shorthand. If the array has a definite size given, the string literal can be as long as the array. If the literal is exactly the same length as the array, a null byte is not appended to the end of the array. If the array has no size given, a null byte is appended to the literal and the array size is set to the length of the literal plus one for the null byte. For example:

```
char a[5] = "abcde";    /* array of 5 characters */  
char b[ ] = "sample";1 /* array of 7 characters */
```

Scope of Identifiers

There are four distinct classes of identifiers in this implementation as follows:

1 Variables, typedefs, and enumeration members

Each member must be unique within the block in which it is defined. Externally declared identifiers must be unique among externally declared variables.

2 Structure, union, and enumeration tags

Each tag must be unique within the block in which it is defined. Tags declared outside of any function must be unique within all tags defined externally.

3 Structure and union members

Member names must only be unique within the structure or union in which they are defined. There is no restriction on the type or offset of members in different structures with the same identifier.

4 Goto labels

Goto labels must be unique within the function in which they are declared.

For identifiers declared within a block, an externally declared identifier can be redeclared. This new identifier masks the outer definition for the duration of the current block. Similarly, inner blocks can redefine identifiers declared in outer blocks.

Linkage

In a multiple file program, each global variable can be declared with the keyword `extern` in any of the source files, but the global variable can only be declared once without that keyword. The file in which the variable declaration occurs without `extern` is the file where space is reserved for the variable, and the only place where an initializer can be supplied.

Global variables use the first 32 characters of the identifier, just as local variables do (and this length can be changed by the `-i` command line option). Because the linker does not distinguish between uppercase and lowercase, all global variable identifiers have lowercase letters mapped to uppercase. Care should be taken to ensure that identifiers are kept unique.

Expressions

This implementation definitely evaluates subexpressions in an order intended to minimize the number of registers needed. For this reason, side effects do not occur in an easily predictable manner except for those operators (comma, `&&`, and `!`) where the order of evaluation is required by the language definition.

Operators involving side effects (such as assignments) can be evaluated in any order the compiler chooses to produce the most efficient code. The only constraint is that any side effect is accomplished by the time the expression evaluation is done. Function calls occurring within a parameter to another function call are completed before the outer function call is begun.

Because floating point computations are sensitive to round-off errors, most floating point algorithms need to control the order and grouping of operations. For this reason, floating point operations are carried out as grouped by parentheses and, for expressions without parentheses, by operator precedence, even though the operations can be commutative or associative.

Lvalues

The term lvalue is used to mean any expression which can appear on the left-hand side of an assignment. Expressions of type array or function are never lvalues.

Simple non-array and non-function identifiers are lvalues. In addition, certain operators produce lvalues and other operators require lvalues as operands. In the discussion of specific operators that follows, any requirements with regard to lvalues are specified.

Primary Expressions

A primary expression is the simplest component of an expression in C.

An identifier is a primary expression, provided it has been suitably declared. Any extern, auto, static or register object, formal parameter, or enumerator can appear as a primary expression. The type of an identifier is the type given in its declaration, except for functions and arrays. Identifiers of these types are implicitly converted when they are used. An identifier of type function is converted to type pointer to function. An identifier of type array of A is converted to type pointer to A. This conversion is performed in all contexts, except when the identifier is an operand of a sizeof operator.

A constant is a primary expression. Its type depends on its form as described.

A string is a primary expression. The type of a string is 'array of const char', and except in a sizeof expression is converted to 'pointer to const char'. The value is a pointer to the initial character of the string.

An expression within parentheses is a primary expression. The type is the type of the enclosed expression.

A `sizeof` keyword followed by a cast is a primary expression. The type is in bytes, of the type named by the cast.

Postfix Operators

The postfix operators are `++`, `--`, function call, subscript, and member-access (`.` and `->`). These operators are evaluated from left to right following the primary expression which acts as the operand of the expression.

Postfix Increment and Decrement Operators

The operand of a postfix increment or decrement (`++` or `--`) operator must have integral or pointer type and must be an lvalue. The value of the expression is the value of the operand.

After that value is extracted, the lvalue is incremented or decremented by 1.

Function Calls

A function call is a primary expression of type function followed by a possibly empty list of assignment expressions, separated by commas and enclosed in parentheses. The list of expressions in parentheses are the actual arguments to the function call. These arguments correspond to the formal arguments declared in the function definition being called. The value of a function call is the value returned by the function. The type of a function call is `x` if the type of the primary expression operand is function returning `x`.

The language modifiers for a function definition must match the modifiers used in the declaration of the function at all calls to the function.

If the primary expression in a function call is an identifier that has not been declared previously, it is implicitly declared as if it had been declared in the innermost block containing the call with the following declaration:

```
extern int identifier( );
```

Integral arguments to a function call when a function prototype has not been previously declared are converted according to the integral widening rules. Float type arguments are converted to double before being passed. (For more information, refer to Conversions.) When a function prototype is in scope, the argument given is converted to the type of the declared parameter as if by assignment.

When a function prototype includes an ellipsis (...), all function arguments given up to the ellipsis are converted normally and any arguments given beyond the fixed parameters are widened according to the normal rules for function arguments when a prototype is not present.

A function can modify the values of its formal parameters, but this has no effect on the actual arguments supplied, except for interrupt functions. If a pointer is passed as an argument, the object the pointer points to can be changed in all functions.

If the type and number of arguments passed to a function do not match the formal parameters of the function without a function prototype, the results are unpredictable, except that in this implementation extra arguments beyond the formal parameters are ignored. In this implementation, if fewer arguments are supplied than there are formal parameters, there is no harm as long as the unsupplied formal parameters are not actually accessed.

If a prototype is present the number of arguments must match (unless an ellipsis is present in the prototype). The types must be compatible only to the extent that an assignment can legally convert them. An explicit cast can always be used to convert an argument to a type acceptable to a function prototype.

The following example should clarify these points:

```

int      strcmp(char *s1, char *s2);
          /* Full prototype */
char     *strcpy();
          /* No prototype */
int      sampl(float, int, ...);
          /* Full prototype */

samp2()
{
  char   *sx, *cp;
  double z;
  long   a;
  float  q;

  if     (strcmp(sx, cp))
          /* 1. Correct */
        strcpy(sx, cp, 44);
          /* 2. OK in C, not portable */
  sampl(3, a, q); /* 3. Correct */
  strcpy(cp);    /* 4. Bad */
  sampl(2);     /* 5. Compile Error */
}

```

The five calls illustrate different points about function calls as follows.

- In point 1, the use of `strcmp` exactly matches the prototype and everything is proper.
- In point 2, the call to `strcpy` has an extra argument (`strcpy` is defined for two arguments). In this case, BTOS wastes a little time and code pushing an extra argument but there is no syntax error because the compiler has not been told about the arguments to `strcpy`. A LINT compile complains about the extra argument and it is certainly not portable. In fact, if you compile this file using the `-p` compile option (and recompile the library to be compatible), the extra parameter almost certainly causes the program to crash.
- In point 3, the prototype directs that the first argument to `sampl` be converted to a float and the second argument to an int. The compiler warns about possible loss of significant digits because a conversion from long to int chops the upper bits. An explicit cast to int eliminates the warning. The third argument, `q`, lines up with the ellipsis in the prototype, so it is converted to double according to the default rules and the whole call is correct. Even if a `-p` compile option is used, this call correctly passes the arguments.

- In point 4, `strcpy` is again called now with too few arguments. This causes an execution error, and knowing how `strcpy` works it can crash the program. A LINT compile uncovers the error, but a normal compile says nothing even though the number of parameters differs from that in a previous call to the same function.
- In point 5, `samp1` is called with too few arguments. Since `samp1` requires a minimum of two arguments, this statement is an error and produces a message about too few arguments in a call. This error happens both in LINT compiles and in normal compiles.

It should be noted that if any of the prototypes given do not match the function definition, this fact is not known in a normal compile. You must run a LINT compile to be sure that what you say a function accepts is really accurate.

PL/M Functions

A function declared with the `plm` language modifier uses the PL/M calling sequence. This calling convention alters several different aspects of function calling. The order in which the arguments are pushed onto the stack is reversed from the normal C calling conventions. Also, the registers used to return certain data types (such as pointers) differ from the C convention. Finally, the function itself is responsible for removing the function arguments at return time where the normal C calling convention has the caller remove the arguments. This last point effectively eliminates the possibility of using an ellipsis (i.e. a variable arguments function) in a `plm` function definition.

Since PL/M does not support register variables, a C function calling a `plm` function must save the values of the register variables before each call and restore them on each return.

Using the `-p` compile time option effectively turns all non-variable arguments functions into `plm` functions with one important exception. As long as a function does not explicitly carry the `plm` language modifier it uses register variables, even if the `-p` option is present in the compile.

Interrupt Functions

Interrupt functions normally should be declared to be of type `void`. An interrupt function is compiled with extra function entry and exit code so that registers `AX`, `BX`, `CX`, `DX`, `SI`, `DI`, `ES`, and `DS` are preserved. The other registers, `BP`, `SP`, `SS`, `CS`, and `IP`, are preserved as part of the C calling sequence or as part of the interrupt handling process. A typical definition might be:

```
static      void      handler interrupt( )
{
    ...
}
```

Interrupt functions can be declared in any memory model. As in the huge model, `DS` is set to the program data segment. Note that for small and medium model programs, there is no guarantee that `SS` is currently set to the program data segment (if the interrupt is connected to a device), since the interrupt could have occurred during the execution of another program or during the execution of `BTOS`.

For this reason, small and medium model programs must observe the following restriction when coding functions which are interrupt handlers, or can be called by interrupt handlers. Interrupt handling code in the small and medium model can assign the address of a function parameter or automatic variable to a `_ss` or far pointer only. Most runtime library routines obey this restriction, but notably the `printf` family of functions do use an array on the stack with unqualified pointers.

Floating point arithmetic can be used by interrupt handlers in all memory models but should not use any 8087 without saving the state of the chip and restoring it on exit from the handler. If the handler is not using inline 8087 instructions, you can force use of the emulation routines by setting `_8087` to 0 at entry to the handler, and restoring `_8087` to its prior value on leaving the handler.

An interrupt handler routine can be defined with parameters in order to access the registers of the interrupted routine. This is particularly useful for interrupt services designed to be called via INT instructions. The layout of the parameters are:

```
void handler interrupt(bp, di, si, ds, es, dx, cx, bx,  
                      ax, ip, cs, flags, caller stack)
```

An interrupt function can modify its parameters, and by changing the declared parameters, modify the corresponding register when the interrupt handler returns.

Array Subscripts

A postfix expression followed by an expression enclosed in square brackets is a subscripting expression. The postfix expression preceding the square brackets must be of array or pointer type (since arrays are implicitly cast to pointers the two are equivalent). If the type of the postfix expression operand is 'pointer to A', the type of the result is 'A'.

The expression enclosed in square brackets is converted to type int or long depending on the memory model used to compile the program. If the `-ml` or `-mh` command line options are used to compile the file, the expression is converted to long, otherwise to int.

A subscripting expression is an lvalue unless the resulting type is an array.

A subscripting expression `E1 [E2]` is defined to be equivalent to the expression `(* (E1 + E2))`. The two forms are freely interchangeable. As a degenerate case `E1 [0]` is equivalent to `(* E1)`.

The value of the expression `E1 [E2]` is the `E2` element of the `E1` array (counting from zero).

Member Access Operations

An expression followed by a dot (`.`) or an arrow (`->`) and an identifier is a postfix expression. The identifier following the dot or arrow must be the name of a structure or union member.

In the case of dot, the operator before the dot should have structure or union type and must be an lvalue. In the case of an arrow, the operand before the arrow should be a

pointer to a structure or a union, but can be any pointer type. Proper usage requires that the member name given be a valid member of the structure given on the left. The value of the expression is the value of the member named. The expression is an lvalue.

If the left-hand side is not a structure, the expression may still be legal. If the named member is unique among all structures defined in the current scope, or if all members of the same name have the same type and structure offset, only a warning is given and the expression is allowed. The left-hand side is implicitly converted to the appropriate structure type. If two or more members of the same name, with different offsets or types, are known then the reference is ambiguous and is an error.

Unary Operators

A postfix expression can be preceded by one or more unary operators. Unary operators are evaluated from right to left.

Prefix Increment and Decrement Operators

The operand of a prefix increment or decrement must have integral or pointer type and must be an lvalue. The value 1 is added to or subtracted from (depending on the operator) the operand. The value of the expression is the value of the operand after the operation.

The `++i` is fully equivalent to `(i += 1)`, and `--i` is equivalent to `(i -= 1)`.

Address and Indirection

The value of the unary `&` operator is the address of the object that is the operand. Unary `&` requires an lvalue. A register variable cannot be used with the unary `&` operator. The type of the result is 'pointer to A' where the type of the operand is 'A'.

Unary `&` is redundant with objects of type array or function. While these types are not normally lvalues, you can use unary `&` with objects of these types, but the compiler warns you that the operator is superfluous.

The operand unary `*`, or indirection operator, must be a pointer. The result is the object pointed to by the pointer. The expression is an lvalue. If the pointer points to a function the result is an expression that can be used to call the function.

If `P` is a pointer type, and `*A` is a valid lvalue, then `*(P)A` is an lvalue of the type pointed to by `P`.

Another identity is that if `A` is an lvalue, `&A` is an lvalue equal to `A`. Similarly, if `A` has pointer type, `*A` is equal to `A`.

Unary Arithmetic Operators

The unary arithmetic operators are: unary `+` operator, unary `-` operator, unary `~` operator, `!` operator, and `sizeof` operator. A description of how they function follows:

- The result of the unary `+` operator is the value of its operand. The operand must have arithmetic type. The integral widening conversions are performed and the result has the widened type. The compiler does not reorganize expressions across a unary plus. Normally the compiler regroups expressions, rearranging commutative operators such as binary `+` in an effort to create an efficiently compilable expression. This means that a floating point expression that is sensitive to precision errors or overflow can be controlled by means of a unary `+` operator, without having to be split up into separate expressions involving assignments to temporaries.
- The unary `-` operator produces the arithmetic negative of the operand. The normal arithmetic widening conversions are performed and the result has the widened type. The operand must have arithmetic type. Unary `-` of an unsigned expression still has unsigned type.
- The unary `~` operator produces the bitwise complement of the operand. The operand must have integral type. The integral widening conversions are applied and the result has the widened type. For signed objects, `~E` is equivalent to `(-E + 1)`. For unsigned types, `~E` is equivalent to `(MAX_UNSIGNED - E)` where `MAX_UNSIGNED` is the maximum unsigned value for the widened type. Its effect is to reverse each 0 bit to 1 and each 1 bit to 0.

- The ! operator is the logical negation operator. If the value of the operand is zero, the result is 1. If the value of the operand is non-zero, the result is 0. The result is always type int.
- The sizeof operator, with an optional cast following, can be used with any prefix expression following. The result is of type unsigned int and has the value of the size in bytes of the type of the operand. The operand itself, even if it contains a function call or other side effects, does not produce executable code.

Casts

A cast is an abstract type enclosed in parentheses. Casts can appear following a sizeof keyword or can be used as a unary operator to convert the operand expression to the named type. Both the operand and the cast must have scalar type.

In the following example, the usage is an error:

```
sizeof (int) x
```

This is an error because the (int) binds to the sizeof, and the result appears as two consecutive primary expressions, a syntax error.

An abstract type is a type specifier followed by an abstract declarator. The abstract declarator can usually be formed by first writing a normal declarator, then removing the identifier. In the following example, this produces an ambiguous result:

```
int      (i);           /* declares an int */
( int ( ) )           /* constructs a function
                       returning int */
```

must be coded as:

```
( int )
```

In general, if the model declaration is written with a minimum of parentheses, the result obtained by removing the identifier is an abstract type producing the same type as the model.

Binary Operators

The binary operators bind unary expressions or other binary expressions as left and right operands to form binary expressions. The following table lists the operators ordered by precedence:

operator	precedence
*	10
/	10
%	10
+	9
-	9
<<	8
>>	8
<	7
>	7
<=	7
>=	7
==	6
!=	6
&	5
'	4
	3
&&	2
	1

When there is a choice of binding unary operands to one of two binary operators, the operands bind first to the operator with the highest precedence. If the two operators have equal precedence, the left-hand operator binds first.

Normal Arithmetic Operators

Each of the operands must have an arithmetic type, except for the % operator which must have integral type. The + and - operators also allow pointer types as described. The operands of the +, -, /, *, and % operators are converted according to the usual arithmetic conversions as follows:

- The * operator multiplies the two operands and produces the product, with the type of the converted type of the operands.
- The / operator divides the two operands and produces the quotient, with the type of the converted type of the operands.
- The % operator divides the two operands and produces the remainder, with the type of the converted type of the operands.
- The + operator adds the two operands and produces the sum, with the type of the converted type of the operands.

The + operator can have one, but not both, of the operands be of pointer type. The other operand must be of integral type. The integral operand is multiplied by the size of the object pointed to by the pointer operand and then added to the pointer. The result has the type of the pointer.

- The - operator subtracts the two operands and produces the difference, with the type of the converted type of the operands.

The - operator can have one or both of the operands be of pointer type. If only one operand is of pointer type the other operand must be of integral type. The integral operand is multiplied by the size of the object pointed to by the pointer operand and then subtracted from the pointer. The result has the type of the pointer.

If both operands are pointers, they must be pointers to the same type. The difference is computed and the result is divided by the size of the object pointed to by the pointers. The result has int or long type depending on the memory model used. If the memory model option given in the compile is -ml or -mh, the result type is long.

Shift Operators

The operands of a shift operator must be of integral type. The normal integral widening rules are applied to the left-hand operand. The right-hand operand is converted to `int`. The result is the widened type of the left-hand operand.

Shift operators shift the bits of the integral quantity on the left either to the left (or right ()) by the amount given in the right side operand.

The largest meaningful shift value for an `int` object, signed or not, is 15. The largest meaningful shift value for a `long` object, signed, or unsigned, is 31.

A left shift zero fills on the right side. A right shift zero fills on the left if the left operand is unsigned. A right shift of a signed quantity propagates the sign bit.

Negative shift values or values larger than the largest meaningful value produces zero on all left shifts and unsigned right shifts. Signed right shifts set all bits to the value of the sign bit.

Relational Operators

The operands can have arithmetic type, or can both be pointers. If one operand is the constant 0, the other can be any kind of pointer. The result is always of type `int` and has the value of either 0 or 1.

If both operands have arithmetic type, the usual arithmetic conversions are applied.

If two pointers are compared, the result depends on the relative positions of the objects pointed to. The comparison is done as if they were unsigned integers. The only way to guarantee that the comparison of two pointers is meaningful is if they point to the same aggregate object.

Each of the operators `<` (less than), `>` (greater than), `<=` (less than or equal), `>=` (greater than or equal), `==` (equal), and `!=` (not equal) produce the value 1 if the relation is true, the value 0 if false.

Bitwise Boolean Operators

The binary `&` (and), `^` (exclusive-or), and `|` (inclusive-or) operators perform boolean arithmetic. Both operands must have integral type. The normal arithmetic conversions are applied. The result is the converted type of the operands.

Logical Operators

The `&&` (logical and) and `||` (logical or) operators are used to do logical testing, where the order of the tests is important. The operands must have scalar type.

If the first operand of `&&` is zero, the result of the operation is 0 (false) and the right-hand operand is not evaluated. If the result of the left-hand operand is non-zero (true) the result is the result of the right-hand operand.

If the first operand of `||` is non-zero (true), the result of the operation is 1 (true) and the right-hand operand is not evaluated. If the result of the left-hand operand is zero (false) the result is the result of the right-hand operand.

Conditional Expressions

At a lower precedence than any previous operator, the `?:` ternary operator allows conditional computation of a value in an expression. A conditional expression is of the form:

```
test ? true_expression : false_expression
```

Each of the operands can have any scalar type. If the value is non-zero, the `true_expression` is evaluated, otherwise the `false_expression` is evaluated. The `true_expression` and `false_expression` have the usual arithmetic conversions applied. The type of the result is the type of the converted expressions. The value is the value of the true or false expression actually evaluated.

Simple Assignment

The basic assignment operator (`=`) is used to copy a value from one object or expression to a destination object. The left-hand operand must be an lvalue.

The type of the right-hand side is converted to that of the left.

Any arithmetic type can be assigned.

Assignment of a pointer must be to a pointer of the same type. If not, the compiler produces a warning message. Assigning a long quantity to an integer variable produces a warning message about possible loss of significant digits. On some computers integer quantities are the same size as long quantities and when moving code from such a workstation to an 8086, this can be a cause for error.

Structures and unions can be assigned. The structures or unions being assigned must have the same type.

The type of the result is the type of the left-hand side, and the value is the value of the right-hand side converted to the type of the left.

Compound Assignment

The compound assignments correspond to the binary operators as follows:

*	*=
/	/=
%	%=
+	+=
-	-=
&	&=
^	^=
	=
<<	<<=
>>	>>=

An expression of the form $A \text{ op} = B$ is equivalent to $(A = A \text{ op } (B))$, except that any side effects caused by operators in A are performed only once.

The types allowed are the same as those allowed for the binary operators. In addition, for $+=$ and $-=$ only, the left-hand side can be a pointer. In this case, the right-hand side must have integral type.

Comma Operator

At the lowest precedence of all operators, the comma operator can be used to connect two other expressions. It is guaranteed that the left-hand operand is executed first, followed by the right-hand operand. The result value and type is that of the right-hand operand.

In function call arguments a comma operator must be enclosed in parentheses to distinguish it from an argument separator. For example:

```
f(i, (t = 2, t - 5), c);
```

This call passes three arguments, the second with the value `-3`.

Constant Expressions

Any expression exclusively involving constant operands and/or `sizeof` operators are evaluated at compile time. A function call, `++`, `--`, unary `*`, array subscripting, member access (`.` and `->`), the unary `&` or any assignment operator cannot appear in a constant expression.

Integral constant expressions cannot include floating point constants, unless the expression is explicitly cast to an integral type.

Preprocessor directives involving constant expressions cannot include the `sizeof` operator, any casts, nor any floating point constants.

Conversions

Conversion can be caused explicitly by means of a cast, or implicitly as part of some operator. Some conversions do not affect the actual bit value of the object being converted, such as converting from signed `int` to unsigned `int`. Other conversions, such as from `double` to `float` cause a transformation of the data.

Integral Widening Conversions

When converting a type according to the integral widening rules, the following conversions are performed.

If the starting type is char, signed char, unsigned char, short, or any enumerated type, it is converted to int. Signed char type data is sign extended to int size. Char type is signed extended if the `-K` option is not used, otherwise, it is zero filled. Unsigned char type is always zero filled.

If the starting type is unsigned short, it is converted to unsigned int.

Usual Arithmetic Conversions

When performing the usual arithmetic conversions, two types are present. Before matching up the two types each type is individually widened according to the following rules:

If the starting type is char, signed char, unsigned char, short or an enumerated type, it is converted to int. Char and signed char type data is sign extended to int size.

Note that if the `-K` option is selected, converting char type to int does not sign extend, instead they zero-fill.

If the starting type is unsigned short it is converted to unsigned int.

If the starting type is float, it is converted to double.

Once these conversions are performed, the following conversions are performed (the order of the two types in the operands is unimportant):

- If either operand is of type long double, the other operand is converted to long double
- otherwise, if either operand is of type double, the other operand is converted to double
- otherwise, if either operand is of type unsigned long, the other operand is converted to unsigned long
- otherwise, if either operand is of type long, the other operand is converted to long

- otherwise, if either operand is of type unsigned, the other operand is converted to unsigned
- otherwise, both operands are of type int

Other Types

Pointers to functions and pointers to data cannot be meaningfully converted to one another. Pointers to data type can be freely converted from one to another. There are no alignment restrictions on the 8086 family of processors. On an 8086, word accesses on an odd address boundary are somewhat slower, but work correctly. On an 8088, word accesses are always the same speed regardless of alignment. The compiler makes no attempt to align data. Structures, in particular, are not padded after an odd number of bytes. If alignment is desired, the programmer should take care to declare structures appropriately or use the `-a` option.

Enumeration data is implicitly converted to or from int as used. Enumerators are equivalent to integers in all respects.

Statements

Statements are the executable instructions of a program. Each statement is executed in order within a statement list unless directed otherwise by control-flow statements.

Labeled Statements

identifier :
case constant_expression :
default :

Any statement can be preceded by an identifier and a colon, declaring the identifier to be a statement label. This identifier can only be used as a target of a goto statement. The identifier can be used in a goto anywhere in the function where the label is defined.

Case and default labels can be used within switch statements. A case label begins with the case keyword, and is followed by an integral constant expression and a colon. A default label consists of the default keyword followed by a colon.

Blocks

```
{ declarations statements }
```

A block (or compound statement) allows you to group a statement list into a single unit. Each block can also have its own set of declarations. All of the declarations of a block must precede any of the statements in the block.

Objects declared with automatic storage and initialized are created and initialized on each entry to the block.

Expression Statement

```
expression ;
```

An expression statement is simply an expression followed by a semicolon. This expression is evaluated for its side effects (assignments and function calls).

Null Statement

A null statement is simply a semicolon with no expression before or after it. It is most commonly used as the body of an iteration statement. A null statement does nothing.

Selection Statements

```
if ( expression ) statement
```

```
if ( expression ) statement  
    else statement
```

```
switch ( expression ) statement
```

Selection statements choose one of several flows of control. The expression controlling an if statement must have scalar type. An else is associated with the nearest previous else-less if statement that is in the same block and not any enclosing block.

The expression controlling a switch statement must have integral type and is converted to int. The switch body statement is normally a block. Control is transferred to the case label matching the value of the switch control expression. If no case label matches, control transfers to any default label supplied. If no default label is present, control passes to the next statement after a switch.

Initializers in auto declarations in a block that is a switch body have no effect since they occur before any case labels.

Iteration Statements

`while (expression) statement`

`do statement`

`while (expression) ;`

`for (expr1 ; expr2 ; expr3) statement`

In a while statement the controlling expression is evaluated, and if non-zero, the statement body is executed. The controlling expression is then evaluated again and this is repeated until the controlling expression evaluates to zero. If the controlling expression is zero on the first evaluation, the loop body is not executed. The type of the controlling expression must have scalar type.

In a do statement the loop body statement is executed, then the controlling expression is evaluated, and if non-zero, the statement body is repeated. This is repeated until the controlling expression evaluates to zero. The loop body is always executed at least once. The type of the controlling expression must have scalar type.

In a for statement, `expr1` is evaluated once. This is the initialization expression. Then `expr2` is evaluated as a loop control expression. If non-zero, the loop body statement is executed. Then `expr3` is executed (the increment part). The control test `expr2` is then re-evaluated and the loop repeated until `expr2` has value 0.

`Expr1` and `expr3` can have any type, including void. `Expr2` must have scalar type.

Any of the three expressions can be omitted. If `expr1` or `expr3` is omitted, nothing is done at that point in the loop. If `expr2` is omitted, there is no test performed and the loop continues forever, unless some method is used to explicitly exit the loop, such as a `break` or `return`.

A for loop is equivalent to the following sequence involving a while loop:

```
expr1;
while (expr2){
    statement;

    expr3;
}
```

A for loop differs from this construct in that `expr2` is allowed to be null, where a while statement must have some expression. Second, a `continue` statement within the loop body transfers control to `expr3` in a for loop, but to `expr2` in a while loop.

Jump Statements

The jump statements are as follows:

- `goto identifier ;`
A `goto` statement immediately transfers control to the label given by `identifier`. The label must be within the same function as the `goto` statement.
- `break ;`
A `break` statement can only occur inside a `switch`, `while`, `do`, or `for` statement. The statement causes control to immediately transfer to the statement following the innermost enclosing `switch`, `while`, `do`, or `for` statement.
- `continue ;`
A `continue` statement can only occur inside a `while`, `do`, or `for` loop statement. Control is immediately transferred to the controlling expression in a `do` or `while` statement, and to the increment expression in a `for` loop.
- `return ;`
A `return` statement causes the current function to immediately return to the caller. If control reaches the closing curly brace of a function, an equivalent of a `return` statement is executed with no return value.

- return expression ;

A return statement can include an expression. The value of that expression is converted to the type of the function and the value becomes the value of the function call in the calling point of the program.

If a return with no expression is executed and the call to the function expects a value to be returned, the results are undefined.

Inline Assembly Statements

The two inline assembly statements are as follows:

```
asm char_sequence newline
asm char_sequence ;
```

The BTOS C Compiler supports the use of inline assembly language. An inline assembly language statement is introduced with the keyword `asm`. From the `asm` keyword to either the end of the current source line or a semicolon is treated as a single assembly statement. Assembly statements cannot be continued across more than a single line. The statement is passed through unmodified to the assembly output file. Assembly statements count as a statement when used with `if` or `while`. For example:

```
int    i;
register int    x;

if    (i > 0)
    asm    mov    x,4
else
    i = 7;
```

This construct is a valid C `if` statement. `asm` statements are the only statements in C which depend upon the occurrence of an end-of-line. This is admittedly not in keeping with the rest of the language, but this is the convention adopted by most C compilers.

Assembly statements can be used as an executable statement inside a function, or as an external declaration outside a function. When used outside a function, the assembly statements are inserted in the data segment portion of the program, while assembly statements inside a function are inserted in the code segment.

Variables can be referred to by name if the programmer uses the following conventions.

Using Inline Assembly Language

C is very good for most tasks, but for some things on the 8086, assembly language is necessary. Rather than force the programmer to create a completely separate assembly language module, BTOS C allows the programmer to intermix assembly language statements in the C source. Also, with these statements you can use C symbols, including structure offsets.

The inline assembly facility of the compiler is intended for the programmer who has some experience with assembly language programming, especially the 8086 BTOS Assembler. How to include assembly language programming in C source programs follows.

An inline assembly statement consists of the `asm` keyword, followed by an opcode, and the instruction operands, if any.

The instructions are copied to the output, substituting any C symbols with appropriate assembly language equivalents. The inline assembly facility is not a complete assembler so many errors are not immediately detected. The `-S` option and the assembler must be used to compile programs with inline assembly language. Any errors are caught by the assembler. The assembler is not very good at identifying the location of errors, since the original C source line number is lost.

Inline assembly statements located outside any function are placed in the DATA segment, and assembly statements located inside functions are placed in the CODE segment.

Instruction Mnemonics

Any of the 8086 instruction mnemonics can be included as inline assembly statements. There are four classes of instructions allowed by the BTOS C Compiler: normal instructions, string instructions, jump instructions, and assembly directives. Regardless of instruction type, operands are allowed by the compiler, even if they are erroneous or disallowed by the assembler. The exact format of the operands is not enforced by the compiler.

The following is a list of the normal instruction mnemonics which can be used as normal instructions:

aaa	aam	aas	adc	add
and	arpl	bound	call	cbw
clc	cld	cli	cmc	cmp
cts	cwd	daa	das	dec
div	enter	f2xm1	fabs	fad
faddp	fbld	fbstp	fchs	fclex
fcom	fcomp	fcompp	fdecstp	fdisi
fdiv	fdivp	fdivr	fdivrp	feni
ffree	fiadd	ficom	ficomp	fidiv
fidivr	fist	fimul	fincstp	finit
fild	fistp	fisub	fisubr	fld
fld1	fldcw	fldenv	fldl2e	fldl2t
fldig2	fldln2	fldpi	flde	fmul
fmulp	fnclx	fndisi	fneni	fninit
fnop	fnsave	fnstcw	fnstenv	fmstsw
fpatan	fprem	fptan	frndint	frstor
fsave	fscale	fsqrt	fst	fstcw
fstenv	fstp	fstsw	fsub	fsubp
fsubr	fsubrp	ftst	fwait	fxam
fxch	fxtract	fyl2x	fyl2xp1	hlt
idiv	imul	in	inc	ins
int	into	iret	lahf	lar
lds	lea	leave	les	lgdt
lidt	lldt	lmsw	lsl	ltr
mov	mul	neg	not	or
out	outs	pop	popa	popf
push	pusha	pushf	rcl	rcr
ret	rol	ror	sahf	sal
sar	sbb	sgdt	shl	shr
sidt	sldt	smsw	stc	std
sti	str	sub	test	verr
verw	wait	xchg	xlat	xor

BTOS C does not support 8087, 80286, or 80386 instruction mnemonics.

The following string instruction mnemonics can be used alone or with repeat prefixes:

cmps	cmpsb	cmpsw	lods	lodsb
lodsw	movs	movsb	movsw	scas
scasb	scasw	stos	stosb	stosw

The following repeat prefixes can be used:

rep	repe	repne	repnz	repz
-----	------	-------	-------	------

Jump instructions are treated specially. Since a label cannot be included on the instruction itself, jumps must go to C labels. The jump instruction mnemonics are:

ja	jae	jb	jbe	jc
jcxz	je	jg	jge	jl
jle	jmp	jna	jnae	jnb
jnbe	jnc	jne	jng	jnge
jnl	jnle	jno	jnp	jns
jnz	jo	jp	jpe	jpo
js	jz	loop	loope	loopne
loopnz	loopz			

The following assembler directives are allowed in inline assembly statements:

db	dd	dw	extrn
----	----	----	-------

Inline Assembler References to Data and Functions

C symbols can be used in inline assembly code and are automatically converted to appropriate assembly language operands. Any symbol can be used, including automatic variables, register variables, and function parameters. In general, a C symbol can be used in any position where an address operand would be legal. A register variable can be used wherever a register would be a legal operand.

If an identifier is encountered in parsing the operands of an inline assembly instruction, the identifier is searched for in the C symbol table. The names of the 8086 registers

are excluded from this search. Either uppercase or lowercase forms of the register names can be used.

The first two register declarations in a function are treated as register variables and all subsequent register declarations are treated as automatic variables. If the register keyword occurs in a declaration which cannot be a register, the keyword is ignored. Only short, integer (or the corresponding unsigned types), or 2-byte pointer variables can be placed in a register. SI and DI are the 8086 registers used for register variables. Inline assembly code can freely use SI or DI as scratch registers if no register declarations are given in the function. The C function entry and exit code automatically saves and restores the caller's SI and DI. If there is a register declaration in a function, inline assembly can use or change the value of the register variable by using SI or DI, but the preferred method is to use the C symbol in case the internal implementation of register variables ever changes.

The BP register is used in C functions as a base address for arguments and automatic variables. Parameters have positive offsets which vary depending on the memory model and the number of registers saved on function entry. BP always points to the saved previous BP value. Functions that have no parameters and declare no arguments do not use or save BP at all.

Automatic variables are given negative offsets from BP, with the first automatic variables having the smallest magnitude negative offset and subsequent variables given increasing magnitude offsets.

For example, a function with the following automatic declarations at the beginning of the function would generate the corresponding offsets:

```
int    i;      BP-02
long  il;     BP-06
char  c[5];   BP-11
short *p;    BP-13
```

Note that if the `-a` flag is present in the command line for compiling the above declarations, the pointer `p` is given an offset of `BP-14`, leaving one unused byte.

A programmer need not be concerned with the exact offsets of local variables, however. Simply using the name includes the correct offsets.

It can be necessary to include appropriate WORD PTR, BYTE PTR, or other size overrides on assembly instruction. These overrides are often needed when using static or global C symbols (since the compiler defines all static and global variables as BYTE objects). A DWORD PTR override is needed on LES or indirect far call instructions.

C Structure Members

Any member of any C structure can be used in an inline assembly statement (assuming the reference is in the scope of the declaration). The member name can be used in any position where a numeric constant is allowed in an assembly statement operand. The structure member must be preceded by a dot (.) to signal that a member name is being used and not a normal C symbol.

Thus:

```
struct    a    {
          int  a_b;
          int  a_c;
          ...
          };

subroutine ( )
{
  ...
  asm    mov  ax,[di].a_c
  ...
}
```

In the above sequence, the assembler statement would be the equivalent of the following:

```
asm      mov      ax,2[di]
```

Member names are replaced in the assembly output by the numeric offset of the structure member, but no type information is retained. Thus, members can be used indiscriminately as compile time constants in assembly statements.

Jump Instructions and Labels

Any of the conditional and unconditional jump instructions, plus the loop instructions, can be used in inline assembly. They are only valid inside a function. Since no labels can be given in the asm statements, jump instructions must use C goto labels as the object of the jump. Direct far jumps cannot be generated.

Indirect jumps are also allowed. To use an indirect jump, use either a register name as the operand of the jump instruction or else include an operand defining the address to jump to inside square brackets.

Thus, in the following code the first jump goes to the C goto label a. The second jump goes to the address contained in the integer a.

```
int      x( )
{
    int      a;

    ...

    a:

    ...

    asm      jmp      a
    asm      jmp      [a]

    ...
}
```

Comments on Inline Assembler Statements

Assembler style comments cannot be used. When commenting inline assembly statements, you should use C style comments. Assembler style comments begin with a semicolon and continue to the end of the current line. Using this convention can cause the compiler to become confused, since it tries to interpret the comments as assembly language operands.

External Definitions

External definitions are the function and data definitions described.

Function Definitions

An old style function definition consists of a function declaration, which is like any other declaration producing type 'function returning A', except that instead of empty parentheses to denote the function, the parentheses contain the names of any formal parameters, which are listed with comma separators. After the function declaration, an optional set of declarations can be given for the formal parameters. After the formal parameter declarations, the function body is given.

In an old style function definition, a typedef name cannot be given as a parameter name. If the compiler encounters a typedef name in the parameter list it assumes that the name is a type and begins some form of function prototype.

A formal parameter declaration can only include a register storage class specifier. A formal parameter can be declared to have any type, except void and function. Formal parameters declared to be array of A are converted to pointer to A. `sizeof` reports the correct size of all formal parameter types, except array declarations converted to pointer. These return `sizeof` as the `sizeof` of a pointer.

A formal parameter declared to be float is converted on function entry, since the normal widening rules of function arguments require that arguments of type float must be passed as double. A float formal argument is converted from double to float on function entry.

A formal parameter with no declaration is implicitly declared to have type `int`.

In BTOS C, a function can be defined using the function prototype syntax. In this case, the list of types given with the function declarator being defined must be accompanied by the parameter name identifiers. The parameter name identifiers are treated exactly like those of the old style function definition.

The normal widening rules for function parameters apply for most types. However, for float type parameters defined using this syntax the parameter passes as a float and takes up less room on the stack. Note that when you define a function using this syntax, you should also declare it using a prototype wherever the function is called. Float types are the only types affected in this way.

In general, if all types in the prototype definition are widened (that is, there are no `char`, `short`, or `float` types, either signed or unsigned), then the function can be called without a prototype being in the scope of the call.

The function body is a block containing the executable code of the function.

Data Definitions

There must be one data definition for each object declared with storage class specifier `extern`. This definition must be given without the `extern` keyword.

If no initializer is given for a data object it is initialized to zero.

Portability Considerations

This implementation supports two 16-bit register variables, with excess register declarations ignored.

The size of each basic type is as follows:

char, unsigned char, signed char	1 byte
short, unsigned short	2 bytes
int, unsigned	2 bytes
long, unsigned long	4 bytes
float	4 bytes
double, long float	8 bytes
far pointer	4 bytes
near pointer	2 bytes
enumerated data	2 bytes

Diagnostic Messages

This appendix describes the diagnostic messages that you may encounter while compiling or executing your C program. Also refer to the *BTOS II Systems Status Codes Reference Manual* for operating system and other errors.

BTOS C Compiler diagnostic messages fall into three classes: Fatal, Error, and Warning Messages.

Fatal errors typically involve bad file names, disk write errors, or the compiler needing additional core memory. A fatal error may also indicate a compiler error. When a fatal error occurs, compilation immediately stops. Appropriate action must be taken and then compilation may be restarted.

Errors will indicate some sort of syntax or semantic error in the source program. The compiler will complete the current phase of the compilation and then stop. An attempt is made to find as many real errors in the source program as possible during each phase.

Warnings do not prevent the compilation from finishing. They indicate conditions which are suspicious, but which are legitimate as part of the language. Also, use of obsolete syntax or machine-dependent constructs will generate warnings.

Messages are printed by the compiler with the message class first, then the source file name and line number where the condition was detected, and finally the text of the message itself. In this appendix, messages are presented alphabetically within message classes. With each message a probable cause and remedy are provided.

Messages are only generated as they are detected. Because C does not force any restrictions on placing statements on a line of text, the true cause of the error may be one or more lines before the line number mentioned. In the following message list, the messages which often are displayed on lines after the real cause are indicated.

Fatal Messages

Error directive: XXXX

This message is issued when a #error directive is processed in the source file. The text of the directive is displayed in the message.

Error writing assembler file

This message indicates some type of system error while writing the assembler output file. Most often this indicates a full disk or diskette.

Error writing lint file

This error occurs most often when the work disk is full. It could also indicate a faulty diskette. If the diskette is full, try deleting unneeded files and restart the compilation.

Error writing output file

This error most often occurs when the work disk is full. It could also indicate a faulty diskette. If the diskette is full, try deleting unneeded files and restart the compilation.

Expression table full

An extremely complicated expression was parsed. You should break the statement up into multiple expressions. A total of 1000 expression tree nodes are allocated. (As reference, a function call with two simple arguments takes about seven nodes).

Identifier table full

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Incorrect command line arguments

This error occurs if CCOMPILER is executed with incorrect command line arguments.

Inline assembly cannot generate object code

This error occurs only in generating object files. It is generated when an inline assembly language statement is encountered. The object file generator currently does not have the facilities to translate assembly language statements.

Irreducible expression tree

This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. The offending expression should be avoided. Consult Unisys Customer Support if this error is encountered.

Lint file bad format

The LINT file being read contains some bad information. Either the file is not really a LINT file or the file was somehow corrupted.

Must have one filename

A source file was not specified in the CCOMPILER command form. Re-execute the compile, specifying a source file to be compiled.

Out of memory

This error occurs when the total working storage has been exhausted. You should try compiling this program in a partition with more memory (at least 250 Kb).

Pass did not finish properly

This error indicates that some pass either was not run for the named file or it encountered errors.

Register allocation failure

This is a sign of some form of compiler error. Some expression on the indicated line of the source file was too complicated for the code generator to be able to generate code for it. Simplify the offending expression, and if this fails, avoid it. Contact Unisys Customer Support if you encounter this error.

Too many goto labels

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Too many members of structures/unions

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Too many structure/union tags

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Too many variables

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Type table full

The compiler needs more memory to complete the compilation. Either modify the source file to reduce the number of symbols that caused this message, or re-execute the compile in a partition with more memory (at least 250 Kb).

Unable to create assembler file 'XXXXXXXXX.XXX'

This is probably caused by a full disk or diskette directory, or else the named assembler file already exists with the read-only bit set on.

Unable to create output 'XXXXXXXXX.XXX'

This error occurs if the work directory is full. If the directory is full, delete unneeded files and restart the compilation.

Unable to create temp file 'XXXXXXXXX.XXX'

This error occurs if the work directory is full. If the directory is full, delete unneeded files and restart the compilation.

Unable to open lint file 'XXXXXXXXX.XXX'

This error occurs if the work directory is full. If the directory is full, delete unneeded files and restart the compilation.

Unable to open source file 'XXXXXXXXX.XXX'

This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is in the proper directory.

Unexpected end of file in intermediate file

A format problem was detected in the intermediate file written by the parser. Try re-executing the compile to recreate the intermediate file. If the problem persists, notify Unisys Customer Support.

Error Messages

operator not followed by macro argument name

In a macro definition, the # may be used to indicate 'string-izing' a macro argument. The # must be followed by a macro argument name.

'XXXXXXXX' not an argument

The named identifier was declared as a function argument but was not in the function argument list.

Ambiguous symbol 'XXXXXXXX'

The named structure field occurs in more than one structure with different offsets and/or types. The variable or expression used to refer to the field is not a structure containing the field. Cast the structure to the correct type, or correct the field name if it is wrong.

Argument ## in call to 'XXXXXXXX' has wrong type

The argument given by number (argument 1 is the left-most in the call) disagrees with the type declared in the function. The source filename and line number given in the diagnostic is the location of the call.

Argument list syntax error

An argument was followed by a character other than comma or right parenthesis. Arguments to a function call must be separated by commas and closed with a right parenthesis.

Array bounds missing]

An array was declared in which the array bounds were not terminated by a right bracket.

Array size too large

The declared array would be too large to fit in the available memory of the processor.

Assembler statement too long

Inline assembly statements may not be longer than 512 bytes.

Bad filename format in include statement

#Include filenames must be within quotation marks or angle brackets. The filename was missing the opening quotation mark or angle bracket.

Bad ifdef statement syntax

An #ifdef statement must contain a single identifier and nothing else as the body of the statement.

Bad ifndef statement syntax

An #ifndef statement must contain a single identifier and nothing else as the body of the statement.

Bad undef statement syntax

An #undef statement must contain a single identifier and nothing else as the body of the statement.

Bit field size syntax

A bit field must be defined by a constant expression from 1 to 16 bits in width.

Call of non-function

The function being called is declared as a non-function. This is commonly caused by incorrectly declaring the function or misspelling the function name.

Call to undefined function 'XXXXXXXX'

The named function has no declaration in the files.

Case outside of switch

A case statement was encountered outside a switch statement. This is often caused by mismatched curly braces.

Case statement missing :

A case statement must have a constant expression followed by a colon. The expression in the case statement was either missing a colon or had some extra symbol before the colon.

Cast syntax error

A cast contains some incorrect symbol.

Character constant too long

Character constants may only be one or two characters long.

Compound statement missing }

The end of the source file was reached and no closing brace was found. This is most commonly caused by mismatched braces.

Conflicting type modifiers

This occurs when a declaration is given that includes, for example, both near and far keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier may be given on a function.

Constant expression required

Arrays must be declared with constant size. This error is commonly caused by misspelling a define constant.

Declaration missing ;

A struct or union field declaration was not followed by a semicolon.

Declaration needs type or storage class

A declaration must include at least a type or a storage class. This means a statement like the following is not legal:

```
i[ ] = { 4, 5, 6 };
```

Declaration syntax error

A declaration was missing some symbol or had an extra symbol added to it.

Default outside of switch

A default statement was encountered outside a switch statement. This is most commonly caused by mismatched curly braces.

Define statement needs an identifier

The first non-white space characters after a #define must be an identifier. A different character was found.

Division by zero

A divide or remainder in an #if statement has a zero divisor.

Do statement must have while

The closing while keyword was missing from a do statement.

Do-while statement missing (

No left parenthesis was found after the while keyword in a do statement.

Do-while statement missing)

No right parenthesis was found after the test expression in a do statement.

Do-while statement missing ;

No semicolon was found after the closing parenthesis in a do statement test expression.

Duplicate case

Each case of a switch statement must have a unique constant expression value.

Duplicate declaration of 'XXXXXXXX'

The named global variable is declared in more than one source file. The source filename given in the diagnostic is the source file of the second declaration found.

Duplicate declaration of function 'XXXXXXXX', also in 'XXXXXXXX.XXX'

The named function is declared in more than one file. The two source filenames are given in the diagnostic.

Duplicate definition of 'XXXXXXXX'

The #define statement is for an already defined identifier. The new definition supercedes the old.

Enum syntax error

An enum declaration did not contain a properly formed list of identifiers.

Enumeration constant syntax error

The expression given for an enumerator value was not a constant.

Expression syntax

This is a general error message that appears when an expression is being parsed and some serious error was encountered. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

Expression syntax error in #elif statement

The expression in an #elif statement is badly formed: a mismatched parenthesis, extra or missing operator, or missing or extra constant.

Expression syntax error in #if statement

The expression in an #if statement is badly formed due to a mismatched parenthesis, extra or missing operator, or missing or extra constant.

External declaration type mismatch for 'XXXXXXXX'

The external declaration on the given line of the named source file disagrees with the declaration of the global variable.

Extra parameter in call

A call to a function via a pointer defined with a prototype had too many arguments given.

Extra parameter in call to XXXXXXXX

A call to the named function (which was defined with a prototype) had too many arguments given in the call.

For statement missing (

No left parenthesis was found after the for keyword in a for statement.

For statement missing)

No right parenthesis was found after the control expressions in a for statement.

For statement missing ;

No semicolon was found after one of the expressions in a for statement.

Function 'XXXXXXXX' undefined

The named function is called or referred to and no definition for the function was found in the files being checked.

Function 'XXXXXXXX' return value declared inconsistently

The named function has been declared to return some type in the calling file (indicated by the source filename and line number) different from that declared with the function itself. This is often caused by neglecting to declare external functions which return non-integer values, such as the math functions.

Function call missing)

The function call argument list had some type of syntax error such as a missing or mismatched closing parenthesis.

Function declarator missing left parenthesis

A function declaration had a language modifier, but not left parenthesis.

Function definition out of place

A function definition may not be placed inside another function. Any declaration inside a function that looks like the beginning of a function with an argument list is considered a function definition.

Function doesn't take a variable number of arguments

The `va_start` macro was used inside a function that does not accept a variable number of arguments.

Goto statement missing label

The `goto` keyword must be followed by an identifier.

If statement missing (

No left parenthesis was found after the `if` keyword in an `if` statement.

If statement missing)

No right parenthesis was found after the test expression in an `if` statement.

Illegal character 'C' (0xXX)

Some invalid character was encountered in the input file. The octal value of the offending character is printed.

Illegal character in constant expression 'X'

Some character not allowed in a constant expression was encountered. If a letter is the character, this indicates a probably misspelled identifier.

Illegal initialization

Initializations must be either constant expressions, or else the address of a global, external or static variable plus or minus a constant.

Illegal octal constant

An octal constant was found containing a digit of 8 or 9.

Illegal pointer subtraction

This is caused by attempting to subtract a pointer from a non-pointer.

Illegal storage class

Register or auto was used in a declaration outside a function. Or typedef, extern, auto, or static was used in a function argument declaration.

Illegal structure operation

Structures may only be used with dot (.), address-of (&) or assignment (=) operators, or be passed to or from a function. A structure was encountered being used with some other operator.

Illegal use of floating point

Floating point operands are not allowed in shift, bitwise boolean, conditional (?:), indirection, or certain other operators. A floating point operand was found with one of these prohibited operators.

Illegal use of pointer

Pointers may only be used with addition, subtraction, assignment, comparison, indirection, or arrow (->) operators. A pointer was used with some other operator.

Improper use of a typedef symbol

A typedef symbol was used where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.

Incompatible storage class

The extern keyword was used on a function definition. Only static or no storage class at all is allowed.

Incompatible type conversion

An attempt was made to convert one type to another which were not convertible. These include converting a function to or from a non-function, converting a structure or array to or from a scalar type, or converting a floating point value to or from a pointer type.

Incorrect macro call of 'XXXXXXXX'

The named #define macro was used without a left parenthesis immediately following. Only white space may occur between a macro name and its arguments, and the arguments must be given with every call.

Incorrect number format

A decimal point was encountered in a hexadecimal number.

Incorrect use of default

No colon was found after the default keyword.

Initializer syntax error

An initializer has a missing or extra operator, mismatched parenthesis, or is otherwise malformed.

Invalid indirection

The indirection operator (*) requires a pointer as the operand.

Invalid macro argument separator

In a macro definition arguments must be separated by commas. Some other character was encountered after an argument name.

Invalid pointer addition

An attempt was made to add two pointers together.

Invalid use of arrow

An identifier must immediately follow an arrow operator (→).

Invalid use of dot

An identifier must immediately follow a dot operator (.).

Left side must be an address

The left-hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.

Macro argument syntax error

An argument in a macro definition must be an identifier. Some non-identifier character was encountered where an argument was expected.

Macro expansion too long

A macro may not expand to more than 4096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself. This error is also caused when a macro is nested more than 100 levels.

'main' needs 2 arguments: argc and argv

The function main is allowed to be declared in two ways; with no parameters at all, or with argc and argv. The following is a sample declaration of main using the parameters:

```
main(argc, argv)
int      argc;
char     *argv[ ];
{
...
}
```

The two parameters do not have to be named argc and argv, but their types must match.

Misplaced break

A break statement was encountered outside a switch or looping construct.

Misplaced continue

A continue statement was encountered outside a switch or looping construct.

Misplaced decimal point

A decimal point was encountered in a floating point constant as part of the exponent.

Misplaced elif statement

An #elif directive was encountered without any matching #if, #ifdef, or #ifndef directive.

Misplaced else

An else statement was encountered without a matching if statement. This could be caused by an extra semicolon, missing curly braces, or some syntax error in a previous if statement.

Misplaced else statement

An #else directive was encountered without any matching #if, #ifdef, or #ifndef directive.

Misplaced endif statement

An #endif directive was encountered without any matching #if, #ifdef, or #ifndef directive.

Must take address of memory location

The address-of operator (&) was used with an expression which cannot be used that way, for example a register variable.

No filename ending

The filename in an include statement was missing the correct closing quotation mark or angle bracket.

Non-portable pointer assignment

An assignment was made of a pointer to a non-pointer, or vice versa. An assignment of a constant zero to a pointer is allowed as a special case. A cast should be used to suppress this warning if the assignment is proper.

Non-portable pointer comparison

A comparison was made between a pointer and a non-pointer other than the constant zero. A cast should be used to suppress this warning if the comparison is proper.

Non-portable return type conversion

The expression in a return statement was not the same type as the function declaration. This is only triggered if the function or the return expression is a pointer. The exception to this is that a function returning a pointer may return a constant zero. The zero will be converted to an appropriate pointer value.

Not an allowed type

Some sort of forbidden type was declared; for example, a function returning a function or array.

Redeclaration of 'XXXXXXXX'

The named identifier was previously declared.

Size of structure or array not known

Some expression (such as a sizeof or storage declaration) occurred with an undefined structure or array of empty length. Structures may be referenced before they are defined as long as their size is not needed. Arrays may be declared with empty length if the declaration does not reserve storage or if the declaration is followed by an initializer giving the length.

Statement missing ;

An expression statement was encountered without a semicolon following it.

Structure or union syntax error

The struct or union keyword was encountered without an identifier or opening curly brace following it.

Structure size too large

A structure was declared which reserved too much storage to fit in the memory available.

Subscripting missing]

A subscripting expression was encountered which was missing its closing bracket. This could be caused by a missing or extra operator or mismatched parentheses.

Switch statement missing (

No left parenthesis was found after the switch keyword in a switch statement.

Switch statement missing)

No right parenthesis was found after the test expression in a switch statement.

Too few arguments in call to 'XXXXXXXX'

The call to the named function has too few arguments. For a function which takes a variable number of arguments, this diagnostic implies that fewer than the minimum number of arguments were passed. When too few arguments are passed, only the arguments actually passed are checked for consistency.

Too few parameters in call

A call to a function with a prototype via a function pointer had too few arguments. Prototypes require that all parameters are given.

Too few parameters in call to 'XXXXXXXX'

A call to the named function (declared using a prototype) had too few arguments.

Too many arguments in call to 'XXXXXXXX'

The call to the named function contains more arguments than were declared. Strictly speaking this will not cause a program to fail if the function is a normal C function, since any extra arguments are ignored, but this often implies that the call was not coded correctly.

Too many cases

A switch statement is limited to 256 cases.

Too many decimal points

A floating point constant was encountered with more than one decimal point.

Too many default cases

More than one default statement was encountered in a single switch.

Too many exponents

More than one exponent was encountered in a floating point constant.

Too many Initializers

More initializers were encountered than were allowed by the declaration being initialized.

Too many storage classes in declaration

A declaration may never have more than one storage class.

Too many types in declaration

A declaration may never have more than one of the basic types: char, int, float, double, struct, union, enum, or typedef-name.

Too much auto memory in function

More automatic storage was declared in the current function than there is room for in the memory available.

Too much global data defined in file

The sum of the global data declarations exceeds 64 Kb. Check the declarations for any array that may be too large. Also consider reorganizing the program if all the declarations are needed.

Two consecutive dots

An ellipsis contains three dots, and a decimal point or member selection operator uses one dot. Two consecutive dots cannot legally occur in a program.

Type mismatch in parameter ##

The function called via a function pointer was declared with a prototype and the given parameter (counting left-to-right from 1) could not be converted to the declared parameter type.

Type mismatch in parameter ## in call to 'XXXXXXXX'

The named function was declared with a prototype and the given parameter (counting left-to-right from 1) could not be converted to the declared parameter type.

Type mismatch in parameter 'XXXXXXXX'

The function called via a function pointer was declared with a prototype and the named parameter could not be converted to the declared parameter type.

Type mismatch in parameter 'XXXXXXXX' in call to 'XXXXXXXX'

The named function was declared with a prototype and the named parameter could not be converted to the declared parameter type.

Type mismatch in redeclaration

A variable was redeclared with a different type than was originally declared for the variable. This can occur if a function is called and subsequently declared to return a value other than an integer. If this has happened, an extern declaration of the function must be inserted before the first call to it.

Unable to open file 'XXXXXXXX.XXX'

The LINT file named in the command line argument could not be opened. Check for misspellings.

Unable to open #include file 'XXXXXXXXXX.XXX'

The named file could not be found. This could also be caused if an #include file included itself. Check whether the named file exists.

Undefined label 'XXXXXXXX'

The named label has a goto in the function, but no label definition.

Undefined structure 'XXXXXXXX'

The named structure was used in the source file on a line previous to the indicated location of the error, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

Undefined symbol 'XXXXXXXX'

The named identifier has no declaration. This could be caused by a misspelling either at the point of the error or at the declaration. This could also be caused if there was an error in the declaration of the identifier.

Unexpected end of file in comment started on line #

The source file ended in the middle of a comment. This is normally caused by a missing close of comment.

Unexpected end of file in conditional started on line #

The source file ended before a #endif was encountered.
The #endif was either missing or misspelled.

Unknown preprocessor statement

A # character was encountered at the beginning of a line,
and the statement name following was not define, undef,
line, if, ifdef, ifndef, include, else, or endif.

Unterminated character constant

An unmatched apostrophe was encountered.

Unterminated string

An unmatched quotation mark was encountered.

Unterminated string or character constant

A string or character constant was begun and no
terminating quotation mark was found.

Variable 'XXXXXXXX' undefined

External declarations were found for the named variable,
but no global declaration was found.

While statement missing (

No left parenthesis was found after the while keyword in
a while statement.

While statement missing)

No right parenthesis was found after the test expression
in a while statement.

Wrong number of arguments in call of 'XXXXXXXX'

The named macro was called with an incorrect number of
arguments.

Warning Messages

'XXXXXXXX' declared but never used

The named variable was declared as part of the block just ending, but was never used. The error is indicated when the closing curly brace of the compound statement or function is encountered. The declaration of the variable occurs at the beginning of the compound statement or function.

'XXXXXXXX' is assigned a value which is never used

The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the closing curly brace is encountered.

'XXXXXXXX' not part of structure

The named field was not part of the structure on the left-hand side of the dot (.) or arrow (→), or else the left-hand side was not a structure or a pointer to a structure, respectively.

'XXXXXXXX' return value is ignored

This message is displayed if the function is declared to return a value and the value is ignored by a function call. The message is flagged on the call. Void functions produce no such warning, even if the call to the function was not declared to be void. This warning message may be suppressed by explicitly casting the return value to void, such as the following:

```
(void)printf("hello world\n");
```

Note that printf does return a value which is almost always ignored. This can produce many extra warning messages.

Ambiguous operators need parentheses

This warning is displayed whenever two shift, relational or bitwise-boolean operators are used together without parentheses. Also, if an addition or subtraction operator appears unparenthesized with a shift operator, this warning appears.

Both return and return of a value used

This warning is issued when a return statement is encountered which disagrees with some previous return statement in the function. It is almost certainly an error for a function to not return a value in only some of the return statements.

Code has no effect

This warning is issued when a statement is found with some operators which have no effect. For example the statement:

```
a + b;
```

has no effect on either variable. The operation is unnecessary and probably indicates a bug.

Constant is long

A decimal constant greater than 32767 or an octal or hexadecimal constant greater than 65535 was encountered without a letter L following it. The constant is treated as a long.

Conversion may lose significant digits

A conversion from long or unsigned long to int or unsigned int type is required for an assignment operator or other circumstance. Since on some workstations integer type and long type variables have the same size, this kind of conversion may alter the behavior of a program being ported to a new workstation.

Degenerate constant expression

A comparison involving either two constant sub-expressions, or one constant sub-expression which was outside the range allowed by the other sub-expression type. For example, comparing an unsigned quantity to -1 makes no sense. To get an unsigned constant greater than 32767 (in decimal), you should either cast the constant to unsigned (i.e. (unsigned)65535) or append the letter u to the constant (i.e. 65535u).

Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a char expression to 4000, the code will still perform the test. This also means that comparing an unsigned expression to -1 will do something useful, since an unsigned expression can have the same bit pattern as a -1 on the 8086.

Duplicate definition of 'XXXXXXXX'

The named macro was redefined using text that was not exactly the same as the first definition of the macro. The new text replaces the old.

Function 'XXXXXXXX' unused

The named function is not called in the LINT files given. The function may be removed from the program to save space.

Function should return a value

The current function was declared to return some type other than int or void, but a return with no value was encountered. This is usually some type of error.

Non-portable pointer assignment

An assignment was made of a pointer to a non-pointer, or vice versa. An assignment of a constant zero to a pointer is allowed as a special case. A cast should be used to suppress this warning if the assignment is proper.

Non-portable pointer comparison

A comparison was made between a pointer and a non-pointer other than the constant zero. A cast should be used to suppress this warning if the comparison is proper.

Non-portable return type conversion

The expression in a return statement was not the same type as the function declaration. This is only triggered if the function or the return expression is a pointer. The exception to this is that a function returning a pointer may return a constant zero. The zero will be converted to an appropriate pointer value.

Parameter 'XXXXXXXX' is never used

The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by a misspelling of the parameter. This warning can also occur if the identifier is redeclared as an automatic variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

Possible use of 'XXXXXXXX' before definition

The named variable was used in an expression before it was assigned a value. The compiler performs a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the use.

Possibly incorrect assignment

This warning is generated when an assignment operator is encountered as the main operator of a conditional expression (i.e. part of an if, while, or do-while statement). This more often than not is a typographical error for the equality operator. If you wish to suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. For example:

```
if (a = b)
```

should be rewritten as:

```
if ((a = b) != 0) ...
```

Structure passed by value

If the `-s` flag is provided on the compile command line, this warning is generated anytime a structure is passed by value as an argument. It is a frequent error to leave an `&` operator off a structure when passing it as an argument. Because structures can be passed by value, this omission is not an error. The `-s` flag provides a way for the programmer to be warned of this mistake.

Superfluous `&` with function or array

An address-of operator (`&`) is not needed with an array name or function name. Any such operators are discarded.

Suspicious pointer conversion

Some conversion of a pointer to point to a different type was encountered. A cast should be used to suppress this warning if the conversion is proper.

Undefined structure 'XXXXXXXX'

The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.

Unknown assembler instruction

An inline assembly statement was encountered with a disallowed opcode. Check the spelling of the opcode. Also check the list of allowed opcodes to see if the instruction is acceptable.

Unreachable Code

A break, continue, goto, or return statement was not followed by a label or the end of a loop or function. While, do, and for loops with a constant test condition are checked and an attempt is made to recognize loops which cannot fall through.

Void functions may not return a value

The current function was declared as returning void, but a return statement with a value was encountered. The value of the return statement will be ignored.

Zero length structure

A structure was declared whose total size was zero. Any use of this structure would be an error.

Command Line Options Summary

This appendix provides an alphabetical list of BTOS C compiler options beginning with uppercase, then lowercase, and then numeric options. Each one is briefly described. Refer to section 3 for a more detailed discussion of each option and when to use it.

Note: Compiler options must be separated by one or more spaces.

-A	Compile using no BTOS C extensions.
-C	Allow nested comments.
-Didentifier	Defines identifier to '1'.
-Diden=string	Defines identifier to string.
-E	Allow elastic type matches.
-G	Generate code for speed rather than size.
-Idirectory	Define an include directory.
-K	Treat char type as unsigned.
-L	Do a LINT cross-check.
-Lfilename	Do a LINT compile to LINT file filename.
-N	Generate stack overflow logic.
-O	Compile with optimizer.
-Q	Place only definitions in LINT files.
-S	Compile to assemble source and stop.
-T	Warn about explicit casts.
-Uidentifier	Undefine identifier.
-Y	Generate full function entry and exit code.
-Z	Suppress redundant register loads.
-a	Force word alignment of integers.

- b** Do not warn about unreachable breaks.
- d** Suppress long to int conversion warnings.
- f** Generate inline 8087 instructions.
- g** Make all code and data segments PUBLIC, allowing segments with the same name to share the same segment/selector.
- h** Suppress heuristic test warnings.
- i#** Set identifier length.
- j** Place all module data into code segment.
- k** Exit with error code 1 if errors are detected (to stop submit file processing).
- mh** Generate huge model, 20-bit pointer arithmetic.
- mhf** Generate huge model, 16-bit pointer arithmetic.
- ml** Generate large model, 20-bit pointer arithmetic.
- mlf** Generate large model, 16-bit pointer arithmetic.
- mm** Generate medium model, 20-bit pointer arithmetic.
- mmf** Generate medium model, 16-bit pointer arithmetic.
- ms** Generate small model, 20-bit pointer arithmetic.
- msf** Generate small model, 16-bit pointer arithmetic.
- n1path** Place CC0 output in named path directory.
- n2path** Place CC1 output in named path directory.

-nopath	Place CC3 output in named path directory.
-p	Generate PL/M calling sequence.
-q	Suppress undefined symbols in LINT executions.
-r	Suppress register variables.
-s	Warn about passing structures as arguments.
-t	Generate overlay compatible code (use with -S , -mlf , -r , and -Y).
-w	Suppress all warnings.
-wxxx	Suppress the specified warning.
-x	Warn about unused externs.
-zAname	Set code segment class.
-zBname	Set uninitialized data segment class.
-zCname	Set code segment name.
-zDname	Set uninitialized data segment name.
-zGname	Set uninitialized data group name.
-zPname	Set code group name.
-zRname	Set initialized data segment name.
-zSname	Set initialized data group name.
-zTname	Set initialized data class name.
-l	Generate 80186 instructions.

Library Summary

This library summary provides a short description of each function in the library, grouped by general category.

Input–Output Functions

Input–output can be done in any of several ways. It is important to be consistent in the methods used for any given file. For example, if a file is opened using UNIX I/O, UNIX I/O should be used for all operations whenever possible.

Standard I/O

<code>clearerr</code>	Clears the error status of a file.
<code>fclose</code>	Closes a file.
<code>feof</code>	Returns non–zero if end of file was reached on the last input operation on a stream.
<code>ferror</code>	Returns non–zero if an error status has occurred on a stream.
<code>fflush</code>	Flushes any incomplete output buffers by writing them to the disk or device.
<code>fgetc</code>	Reads a character from a file.
<code>fgets</code>	Gets a text line from a file.
<code>fopen</code>	Opens an existing or new file.
<code>fprintf</code>	Does a formatted print to a file.
<code>fputc</code>	Writes a character to a file.
<code>fputs</code>	Writes a string to a file.
<code>fread</code>	Reads one or more records from a file.
<code>freopen</code>	Closes and reopens a file using the same file pointer.

<code>fscanf</code>	Does a formatted read from a file.
<code>fseek</code>	Changes the position of the next read or write in a file, for random access.
<code>ftell</code>	Reports the current position of a file.
<code>fwrite</code>	Writes one or more records to a file.
<code>getc</code>	Reads a character from a file.
<code>getchar</code>	Reads a character from the standard input file.
<code>gets</code>	Gets a text line from the standard input file.
<code>getw</code>	Reads a word (two bytes) from a file.
<code>printf</code>	Does a formatted print to the standard output.
<code>putc</code>	Writes a character to a file.
<code>putchar</code>	Writes a character to the standard output.
<code>puts</code>	Writes a line of text to the standard output.
<code>putw</code>	Writes a word to a file.
<code>rewind</code>	Returns the position of a file to the beginning.
<code>scanf</code>	Does a formatted read from the standard input.
<code>setbuf</code>	Sets a buffer for file activities.
<code>setvbuf</code>	Sets buffering method for file operations.
<code>ungetc</code>	Push a character back onto an input file to be read later
<code>vfprintf</code>	Does a formatted print to a file with the argument list supplied as an array.
<code>vscanf</code>	Extracts formatted values from a file with the argument list supplied as an array.

<code>vprintf</code>	Does a formatted print to the standard output with the argument list supplied as an array.
<code>vscanf</code>	Extracts formatted values from the standard input with the argument list supplied as an array.

Block I/O

<code>close</code>	Closes a file.
<code>creat</code>	Creates a file.
<code>lseek</code>	Changes the current position of a file for random access.
<code>open</code>	Open an existing file.
<code>read</code>	Reads from a file.
<code>write</code>	Writes to a file.

File Management

<code>unlink</code>	Deletes a file.
---------------------	-----------------

String Handling

<code>index</code>	Searches a string for the first occurrence of a second string.
<code>isalnum</code>	Tests whether a character is alphanumeric.
<code>isalpha</code>	Tests whether a character is alphabetic.
<code>isascii</code>	Tests whether a character is ASCII.
<code>isctrl</code>	Tests whether a character is an ASCII control character.
<code>isdigit</code>	Tests whether a character is a digit.

isgraph	Tests whether a character is a non-blank printable character.
islower	Tests whether a character is a lowercase letter.
isprint	Tests whether a character is a printable character.
ispunct	Tests whether a character is a punctuation.
isspace	Tests whether a character is a space, tab, newline, or form-feed.
isupper	Tests whether a character is an uppercase letter.
isxdigit	Tests whether a character is a hexadecimal digit.
memchr	Searches an array for a given character.
memcmp	Compares two fixed size arrays.
memcpy	Copies a fixed size block of memory.
memset	Sets a block of memory to a given value.
movmem	Copies a fixed size block of memory.
rindex	Searches ^{! FOR S} a string for the last occurrence of a second string.
setmem	Sets a block of memory to a particular value.
sprintf	Formats values into a string.
sscanf	Extracts formatted values from a string.
strcat	Concatenates two strings.
strchr	Returns a pointer to the first occurrence of a character in a string.
strcmp	Compares two strings.
strcpy	Copies a string into another.

<code>strcspn</code>	Returns the length of the initial string not containing any characters from a given set.
<code>strlen</code>	Returns the length of a string.
<code>strncat</code>	Concatenates two strings with a maximum length.
<code>strncmp</code>	Compares two strings up to a maximum length.
<code>strncpy</code>	Copies a string up to a maximum length into another string.
<code>strpbrk</code>	Returns the first occurrence of any of a set of characters in a string.
<code>strrchr</code>	Finds the last occurrence of a character in a string.
<code>strspn</code>	Returns the length of the initial string composed of characters from a given set.
<code>strtok</code>	Scans through a string extracting tokens.
<code>swab</code>	Swaps the bytes of a string. Used for moving data between incompatible systems.
<code>toascii</code>	Strips any eighth bit from a character to make it a 7-bit ASCII character.
<code>tolower</code>	Converts uppercase letters to lowercase and leaves other values unchanged.
<code>_tolower</code>	Converts uppercase letters to lowercase. Only works for uppercase letters.
<code>toupper</code>	Converts lowercase letters to uppercase and leaves other values unchanged.
<code>_toupper</code>	Converts lowercase letters to uppercase. Only works for lowercase letters.

<code>vsprintf</code>	Formats values into a string with the argument list supplied as an array.
<code>vsscanf</code>	Extracts formatted values from a string with the argument list supplied as an array.

Memory Management

<code>calloc</code>	Allocate a block from the heap and clear it to zero.
<code>cfree</code>	Free a block back to the heap.
<code>free</code>	Free a block back to the heap.
<code>malloc</code>	Allocate a block from the heap.
<code>realloc</code>	Change the size of a block on the heap.

Miscellaneous Arithmetic

<code>abs</code>	Computes the absolute value of a number.
<code>atof</code>	Converts an ASCII string to a floating point number.
<code>atoi</code>	Converts an ASCII string to an integer.
<code>atol</code>	Converts an ASCII string to a long integer.
<code>ecvt</code>	Convert a floating point number to a string in the <code>printf %e</code> form.
<code>fcvt</code>	Convert a floating point number to a string in the <code>printf %f</code> form.
<code>gcvt</code>	Convert a floating point number to a string in the <code>printf %g</code> form.
<code>rand</code>	Returns a random integer.
<code>srand</code>	Sets the random number generator seed.

<code>strtod</code>	Converts an ASCII string to a floating point number.
<code>strtol</code>	Converts an ASCII string to a long integer.

Searching and Sorting

<code>bsearch</code>	Performs a binary search of a table.
<code>lsearch</code>	Performs a linear search of a table and updates it.
<code>qsort</code>	Sorts a table using a quick-sort algorithm.
<code>ssort</code>	Sorts a table using a shell-sort algorithm.

Program Control

<code>assert</code>	Debugging test macro, aborts program if a test fails.
<code>exit</code>	Exits the current program, closing all open files and flushing any incomplete output buffers.
<code>_exit</code>	Exits the current program without closing files and flushing output buffers.
<code>gsignal</code>	Generate a software signal.
<code>longjmp</code>	Perform a jump out of the normal function call sequence.
<code>setjmp</code>	Set a location for a long jump to later jump to.
<code>ssignal</code>	Set to catch a software signal.

Date and Time Management

asctime	Converts a date and time structure to an ASCII string.
ctime	Converts a BTOS date/time to an ASCII string.
gmtime	Converts a BTOS date/time to a date and time structure in Greenwich Mean Time.
localtime	Converts a BTOS date/time to a date and time structure in Local Mean Time.
stime	Set the date and time using a BTOS date/time.
time	Returns the current date and time as a BTOS date/time.

Hardware Functions

check8087	Determine if an 80x87 NCP is present.
init8087	Initialize the 80x87 NCP.
inport	Input a word value from a hardware port.
inportb	Input a byte value from a hardware port.
outport	Output a word value to a hardware port.
outportb	Output a byte value to a hardware port.
peek	Fetch a word value from anywhere in memory.
peekb	Fetch a byte value from anywhere in memory.
poke	Set a word value anywhere in memory.
pokeb	Set a byte value anywhere in memory.
segread	Store the segment registers in a C structure.

Mathematical Library

<code>acos</code>	Computes the arc-cosine of a floating point number.
<code>asin</code>	Computes the arc-sine of a floating point number.
<code>atan</code>	Computes the arc-tangent of a floating point number.
<code>atan2</code>	Computes the arc-tangent given a cartesian point of two floating point numbers.
<code>ceil</code>	Returns the smallest integer not less than the parameter. Returns a floating point value.
<code>cos</code>	Computes the cosine of a floating point number.
<code>cosh</code>	Computes the hyperbolic cosine of a floating point number.
<code>exp</code>	Computes the exponential function of a floating point number.
<code>fabs</code>	Returns the absolute value of the floating point parameter.
<code>floor</code>	Returns the largest integer not greater than the parameter. Returns a floating point value.
<code>fmod</code>	Returns the fractional part of x modulo y , where x and y are the two parameters.
<code>frexp</code>	Split a floating point number to fractional part and exponent.
<code>ldexp</code>	Load an exponent and fractional part into a single floating point number.
<code>log</code>	Computes the natural logarithm of a floating point number.
<code>log10</code>	Computes the base 10 logarithm of a floating point number.

modf	Splits a floating point number into integer and fractional parts.
pow	Computes the power function (x raised to the y power) for two floating point numbers.
sin	Computes the sine of a floating point number.
sinh	Computes the hyperbolic sine of a floating point number.
sqrt	Computes the square root of a floating point number.
tan	Computes the tangent of a floating point number.
tanh	Computes the hyperbolic tangent of a floating point value.

Glossary

Application partition. An application partition is a section of user memory reserved for the execution of an application.

Arithmetic Operators. An arithmetic operator is a symbol used in an arithmetic expression to indicate the type of arithmetic operation to be performed: the standard operators are add (+), subtract (-), multiply (*), and divide (/).

Array Declarator. An array declarator is a declarator with a trailing pair of square brackets, possibly enclosing an integral constant expression. If no expression is given, the array has unknown size. Otherwise, the expression is the number of elements in the array.

ASCII. ASCII, the American Standard Code for Information Interchange, defines the character set codes used for information exchange between equipment.

Assemble. ASSEMBLE is the Executive command you use to display the Assembler command form.

Assembler. The Assembler translates Assembly 8086 programs into BTOS object modules (machine code).

Binary Operators. A binary operator binds unary expressions or other binary expressions as left and right operands to form binary expressions.

Bind. Bind is a command that activates the Linker to create a version 6 run file. Version 6 run files are required for protected mode compatibility.

BSWA. See Byte Stream Work Area.

Byte Stream Work Area. The Byte Stream Work Area (BSWA) is a 130-byte memory work area for the exclusive use of SAM procedures.

Cast. A cast is an abstract type enclosed in parentheses. Casts can appear following a sizeof keyword, or can be used as a unary operator to convert the operand expression to the named type. Both the operand and the cast must have scalar type.

Class name. A class name is a symbol used to designate a class.

Code listing. A code listing is an English-language display of compiled code.

Glossary-2

Code segment. A code segment is a variable-length (up to 64 Kb) logical entity consisting of reentrant code, and containing one or more complete procedures.

Compiler. BTOS compilers translate high level language programs into BTOS object modules (machine code).

Configuration file. Configuration files specify the options for the C Compiler.

CTOS.lib. The CTOS.lib is part of the Language Development software; it is a library of object modules that provide operating system runtime support.

DGroup. DGroup usually includes data, constant, and stack Linker segments.

8086 Assembly. 8086 Assembly language is the low level language you can use to write BTOS programs. You use the BTOS Assembler to convert the programs into BTOS object modules.

Executive. The Executive is the BTOS user interface program; it provides access to many convenient utilities for file management.

Expressions. In a program, an expression is a combination of various constants, variables, operators, and parentheses used to perform a desired computation.

External reference. An external reference is a reference from one object module to variable and entry points of other object modules.

File access methods. Several file access methods augment the file management system capabilities. File access methods are object module procedures located in the standard BTOS library. They provide buffering and use the asynchronous input/output capabilities of the file management system to overlap input/output and computation.

Function Declarators. A function declarator is a declarator with a trailing optional set of language modifiers and a pair of possibly empty parentheses.

Group. A group is a named collection of linker segments that the BTOS loader addresses at runtime with a common hardware segment register. To make the addressing work, all the bytes within a group must be within 64 Kb of each other.

Identifiers. An identifier is a sequence of uppercase and lowercase letters, digits, and the underbar (`_`) character. An identifier must begin with a letter or the underbar. It can be of any length, but only the first 32 characters are significant.

Language Development. The BTOS Language Development software provides the Linker, Librarian, and Assembler programs (`BIND`, `LINK`, `LIBRARIAN`, and `ASSEMBLE` Executive commands).

.lib. `.lib` is the standard file name suffix for library files.

Librarian. The Librarian is a program that creates and maintains object module libraries. The Linker can search automatically in such libraries to select only those object modules that a program calls.

Library. A library is a stored collection of object modules (complete routines or subroutines) that are available for linking into run files.

Library file. A library file can contain one or more object modules. The file name normally includes the suffix `.lib`.

LINK. `LINK` is the Executive command that activates the linker to create version 4 run files. Version 4 run files are not protected mode compatible.

Linked-list data structure. A linked-list data structure contains elements that link words or link pointers connect.

Linker. The Linker is a program that combines object modules (files that Compilers and Assemblers produce) into run files.

Linker segment. A Linker segment is a single entity consisting of all segment elements with the same segment name.

List file. The Linkerlist file (suffix `.map`) contains an entry for each Linker segment, identifying the segment relative address and length in the memory image. You can direct the Linker to list public symbols and line numbers.

Macros. A macro (short for macroinstruction) is a single instruction that represents a given sequence of instructions. The macro is defined to represent a set of instructions and can be used each time to represent that set.

.map. `.map` is the standard file name suffix for Linker list files.

NCP. See Numeric Co-Processor.

Numeric Co-Processor. An Intel 8087, 80x87, or 80387.

Glossary-4

.obj. .obj is the standard file name suffix for object module files.

Object module. An object module is the result of a single Compiler or Assembler function. You can link the object module with other object modules into BTOS run files.

Overlay. An overlay is a code segment made up of the code from one or more object modules. An overlay is loaded into memory as a unit and is not permanently memory-resident. See also Virtual code segment management.

Parameter. A parameter is a variable or constant that is transferred to and from a subroutine or program.

Physical address. A physical address is an address that does not specify a segment base and is relative to memory location 0.

Pointer. A pointer is an address that specifies a storage location for data.

Pointer Declarators. A pointer declarator is a declaration beginning with an asterisk (*), optionally followed by pointer type modifiers.

Postfix Expressions. A postfix expression is an expression followed by a dot(.) or an arrow (→) and an identifier. The identifier must be the name of a structure or union member.

Process. A process is a program that is running.

Public procedure. A public procedure is a procedure that has a public address; a module other than the defining module can reference the address.

Public symbol. A public symbol is an ASCII character string associated with a public variable, a public value, or a public procedure.

Public value. A public value is a value that has a public address; a module other than the defining module can reference the address.

Public variable. A public variable is a variable that has a public address; a module other than the defining module can reference the address.

Relocation. The BTOS Loader relocates a task image in available memory by supplying physical addresses for the logical addresses in the run file.

Relocation directory. The relocation directory is an array of locators that the BTOS Loader uses to relocate the task image.

Resident. The resident portion of a program remains in memory throughout execution.

.run. .run is the standard file name suffix for run files.

Run file. A run file is a complete program: a memory image of a task in relocatable form, linked into the standard format BTOS requires. You use the Linker to create run files.

Run-file checksum. The Run-file checksum is a number the Linker produces based on the summation of words in the file. The system uses the checksum to check the validity of the run file.

Segment. A segment is a contiguous area of memory that consists of an integral number of paragraphs. Segments are usually classified into one of three types: code, static data, or dynamic data. Each kind can be either shared or nonshared.

Segment address. The segment address is the segment base address. For an 8086/80186 microprocessor, a segment address refers to a paragraph (16 bytes).

Segmented address. A segmented address is an address that specifies both a segment base and an offset.

Segment element. A segment element is a section of an object module. Each segment element has a segment name.

Segment override. Segment override is operating code that causes the 8086/80186 to use the segment register specified by the prefix instead of the segment register that it would normally use when executing an instruction.

Segment registers. There are four segment registers referred to in this manual. They are the Code Segment (CS) register, Data Segment (DS) register, Extra Segment (ES) register, and Stack Segment (SS) register.

Shift Operators. A shift operator must have operands of integral type. A shift operator shifts the bits of the integral quantity on the left either to the left (<<) or right (>>) by the amount given in the right-side operand.

Short-lived memory. Short-lived memory is the memory area in an application partition. When BTOS loads a task, it allocates short-lived memory to contain the task code and data. A client process can also load short-lived memory in its own partition.

Stack. A stack is a region of memory accessible from one end by means of a stack pointer.

Glossary-6

Stack frame. The stack frame is a region of a stack corresponding to the dynamic invocation of a procedure. It consists of procedural parameters, a return address, a saved-frame pointer, and local variables.

Stack pointer. A stack pointer (SP) is the indicator to the top of a stack. The stack pointer is stored in the registers SS:SP.

Statements. A statement is an executable instruction. Each statement is executed in order within a statement list unless otherwise directed by control-flow statements.

.sym. .sym is the standard file name suffix for the symbol file.

Symbol. Symbols can be alphanumeric and/or any other characters, such as underscore, period, dollar sign, pound sign, or exclamation mark.

Symbol file. The Linker symbol file (suffix .sym) contains a list of all public symbols.

Symbolic instructions. Symbolic instructions are instructions containing mnemonic characters corresponding to Assembly language instructions. These instructions cannot contain user-defined public symbols.

Task. A task consists of executable code, data, and one or more processes.

Task image. A task image is a program stored in a run file that contains code segments and/or static data segments.

Text file. A text file contains bytes that represent printable characters or control characters (such as tab, newline, etc.).

Virtual code segment management. Virtual code segment management is the virtual memory method BTOS supports (overlays).

The method works as follows: The Linker divides the code into task segments that reside on disk (in the run file). As the run file executes, only the task segments that are required at a particular time reside in the application partition's main memory; the other task segments remain on disk until the application requires them. When the application no longer requires a task segment, another task segment overlays it.

Index

- +** addition operator, 7-21, 7-51
- =** assignment operator, 7-21
- +=** assignment operator, 7-21
- &** bitwise AND operator, 7-21, 7-58
- ^** bitwise exclusive OR operator, 7-21, 7-58
- ,** comma operator, 7-21, 7-60
- ?:** ternary operator, 7-21, 7-58
- decrement operator, 7-21
- /** division operator, 7-21, 7-56
- ==** equality operator, 7-21, 7-57
- >=** greater or equal operator, 7-21, 7-57
- >** greater than operator, 7-21, 7-57
- |** inclusive OR operator, 7-58
- ++** increment operator, 7-21
- *** indirection operator, 7-21
- !=** inequality operator, 7-21, 7-57
- <<** left shift operator, 7-21, 7-57
- <=** less or equal operator, 7-21, 7-57
- <** less than operator, 7-21, 7-57
- &&** logical AND operator, 7-58
- !** logical negation operator
- |** logical OR operator, 7-58
- %** modulus operator, 7-21, 7-56
- *** multiplication operator, 7-21, 7-56
- >>** right shift operator, 7-21, 7-57
- >** structure pointer operator, 7-21
- subtraction operator, 7-21, 7-56
- unary minus operator, 7-21
- +** unary plus operator, 7-21, 7-56

A

- abs,** 6-9
- acos,** 6-65
- Address,** 7-52
- Advanced options,** 3-6
- Alignment,** 3-6
- APPEND command,** 3-11
- Arithmetic**
 - conversions, 7-61
 - fast pointer, 4-7
 - floating point NCP, 3-7
 - miscellaneous, C-6
 - operators, 7-56
- Arithmetic conversions,** 7-61
- Array**
 - declarators, 7-38
 - subscripts, 7-51

Index-2

Arrays, 7-26
asctime, 6-17
asin, 6-65
ASSEMBLE command, 1-1, 3-3
Assembler, 1-1
 directives, 7-69
 inline, 7-69
Assembly language
 file structure, 5-6
 interface, 5-1
 modules, 5-8
 Using inline, 7-67
assert, 6-10
Assignment operator, 7-58
atan, 6-65
atan2, 6-65
atof, 6-11
atoi, 6-11
atol, 6-11

B

B26, 1-1
B27, 1-1
B28, 1-1, 7-3
B38, 1-1, 7-3
B39, 1-1, 7-3
Basic assignment operator, 7-58
Basic types, 7-23
Binary operators, 7-55
BIND command, 1-1, 3-19
Bitfields, 7-27, 7-39
Bitwise Boolean operators, 7-58
Block I/O, C-3
Blocks, 7-63
bsearch, 6-13

BTOS
 APPEND command, 3-11
 BTOS II, vi, 2-1
 C features, 7-1
 cluster workstation, 1-1
 Context Manager, 2-3
 I/O services, 6-3
 Linker, 3-1, 3-18, 3-20
 workstation, 1-1

C**C**

- basic types, 7-24
- calling sequence, 5-2
- Compiler overview, 1-1
- diagnostic messages, A-1
- features, 1-2
- Installation, 2-1
- keywords, 7-17
- library functions, 3-11, 6-8
- memory requirements, 1-2
- operators, 7-21
- pointers, 3-15
- programming language, 7-1
- runtime library, 6-1
- software installation, 2-1
- structure members, 7-71
- using the, 1-1

Calling

- calling sequence, PL/M, 5-2
- functions, 5-4
- sequence, C

Calling sequence

- PL/M, 3-14

calloc, 6-38**Casts, 7-54****CCompiler command, 3-2****CCompiler.CFG file, 3-19****ceil, 6-24****cfree, 6-38****Character constants, 7-19****Characters, 7-23****check8087, 3-7, 6-14****clearerr, 6-23****close, 6-15****Code**

- generator translations, 7-5

Comma operator, 7-60**Command line**

- options, 3-2

- syntax, 3-2

Comment control option, 3-5**Comments, 7-14**

- inline assembler statements, 7-73

- LINT source file, 3-21

- nested, 3-6

Compatibility options, 3-13

Index-4

Compilation

- activity options, 3-3
- conditional, 7-4
- switch, 1-1

Compile time switches, 1-2

Compiler

- error, A-1
- limits, 7-5
- operation, 3-23

Composite types, 7-25

Compound assignment, 7-59

Conditional compilation, 7-11

Conditional expressions, 7-58

Configuring Context Manager, 2-3

Constant expressions, 7-12, 7-60

Constants

- character, 7-19
- floating, 7-20
- integer, 7-18
- numerical, 7-18

Context Manager

- application information, 2-3
- configuring the, 2-3

Conventions, vi

- lexical, 7-15
- source text, 7-15

Conversions, 7-60

- arithmetic, 7-61
- integral widening, 7-61

cos, 6-65

cosh, 6-67

creat, 6-16

ctime, 6-17

D

Data definitions, 7-74

Date management, C-8

Debugging options, 3-13

Declaration statements, 7-63

Declarations, 7-28

Declarators, 7-35

- array, 7-38
- function, 7-37
- pointer, 7-36

Define

- macros, 7-7

#define macros, 3-5

Defining

- data constants, 5-7
- functions, 5-7

Delimiters, 7-22

DGROUP, 3-16

Diagnostic

checking, 3-10

messages, A-1

Diden

string, 3-5

Didentifier, 3-5

Directives

#error, 7-14

null, 7-14

#pragma, 7-14

Disk requirements, 2-1

Disk usage options, 3-5

Diskettes, 2-1

E

ecvt, 6-19

ENTER instructions, 3-6

Enumerated type, 7-24

Enumerations, 7-34

#error directives, 7-14

Error messages, A-5

Escape sequences, 7-19

Executable instructions, 7-62

exit, 6-20

exp, 6-21

Expression statement, 7-63

Expressions, 7-44

conditional, 7-58

constant, 7-12, 7-60

primary, 7-45

External definitions, 7-73

External variable names, 5-1

F

fabs, 6-24

Fast calling sequence option, 3-14

Fast pointer arithmetic, 4-7

Fatal messages, A-2

fclose, 6-22

fcvt, 6-19

Features, 1-2

feof, 6-23

ferror, 6-23

fflush, 6-22

fgetc, 6-30

fgets, 6-31

File management, C-3

Index-6

Files

- closing, 6-2
- header, 6-4
- opening, 6-2
- temporary, 3-23

Floating

- constants, 7-20
- point arithmetic, 3-7
- types, 7-24

floor, 6-24

fmod, 6-24

fopen, 6-25

fprintf, 6-46

fputc, 6-51

fputs, 6-52

fread, 6-27

free, 4-3, 6-38

freopen, 6-25

frexp, 6-28

fscanf, 6-56

fseek, 6-29

ftell, 6-29

Function

- arguments, 5-2
- calls, 7-46
- declarators, 7-37

Function definitions, 7-73

Functions, 7-25

- calling, 5-3
- defining, 5-6
- hardware, C-8
- I/O, C-1
- interrupt, 7-50
- mathematical, 6-3
- PL/M, 7-49

fwrite, 6-27

G

gcvt, 6-19

getc, 6-30

getchar, 6-30

gets, 6-31

getw, 6-30

Global

- data, 5-8
- symbols, 5-1
- variable, 5-7

Glossary, Glossary-1

gmtime, 6-17

gsignal, 6-68

H

Hardware functions, C-8
Header files, 6-4
Heap, 4-3
Huge model, 1-2, 3-1, 3-4, 5-9

I

Identifiers, 7-16
 scope of, 7-43
Idirectory, 3-5
#include directive, 7-6
#include files, 3-5
index, 6-32
Indirection, 7-52
init8087, 6-14
Initialization, 7-41
Inline
 assembler references, 7-69
 assembler statements, 7-73
 assembly language, 7-67
 assembly statements, 7-66
inport, 6-33
inportb, 6-33
Installation
 C Compiler, 2-1
 diskette, 1-2
Instruction mnemonics, 7-67
Integer constants, 7-18
Integers, 7-23
Integral types, 7-23
Integral widening conversions, 7-61
Interrupt functions, 7-50
I/O
 block, C-3
 error condition, 6-2
 functions, C-1
 operations, 6-1
 pipes, 1-2
 redirection, 1-2
 standard, 6-2
 UNIX compatible, 6-2
I/O services, BTOS, 6-3
isalnum, 6-34
isalpha, 6-34
isascii, 6-34
iscntrl, 6-34
isdigit, 6-34
isgraph, 6-34
islower, 6-34
isprint, 6-34

Index-8

ispunct, 6-34
isspace, 6-34
isupper, 6-34
isxdigit, 6-34
Iteration statements, 7-64

J

Jump
instructions, 7-69, 7-72
labels, 7-72
Jump statements, 7-65

K

Keywords, 7-17
size of, 7-21

L

Labeled statements, 7-62
Large model, 1-2, 3-1, 3-4, 5-9
ldexp, 6-28
LEAVE instructions, 3-6
Lexical
conventions, 7-15

Library
C runtime, 6-1
mathematical, C-9
overview, 6-1
reference, 6-1
runtime, 1-2, 3-19
summary, C-1

Library C functions, 6-8
Line number control, 7-13
Link files, 3-19
Linkage, 7-44
Linking a program, 3-19

LINT
compile, 3-11
facility, 1-2
files, 3-11, 3-22
options, 3-10
source file comments, 3-20

LINT and warning message options, 3-9
localtime, 6-17
log, 6-21
log10, 6-21
Logical operators, 7-58
longjmp, 6-6, 6-63
lsearch, 6-36
lseek, 6-37
Lvalues, 7-45

M**Macro**

arguments, 7-7

Macros

define, 7-3, 7-7

malloc, 4-3, 6-38**Mathematical**

functions, 6-3

library, C-9

Medium model, 1-2, 3-1, 3-4, 5-9**Member access operations, 7-51****memchr, 6-40****memcmp, 6-40****memcpy, 6-40****Memory**

layout, 4-3

management, C-6

organization, 4-3

requirements, 1-2, 3-3

utilization, 3-1

Memory model, 3-1, 3-19, 4-3

huge, 1-2

large, 1-2

medium, 1-2

options, 3-3

small, 1-2

Memory organization, 4-3**memset, 6-40****Message control option, 3-6****Messages**

diagnostic, A-1

error, A-5

fatal, A-2

warning, 3-6, A-19

Mixed model, 3-4**Mixed model programming, 1-2****mnemonics, 7-67****modf, 6-28****movmem, 6-41****N****n1path, 3-5****n2path, 3-5****NCP, 3-7****nopath, 3-5****Normal arithmetic operators, 7-56****NOTREACHED comment, 3-22****Null directive, 7-14****Null statement, 7-63****Numerical constants, 7-18**

Index-10

O

- object files, 1-1**
- Object files, 3-1**
- open, 6-42**
- Operation**
 - compiler, 3-23
- Operators, 7-21**
 - binary, 7-55
 - implemented as subroutines, 6-1
 - normal arithmetic, 7-56
 - pointer arithmetic, 7-26
 - postfix, 7-46
 - unary, 7-52
- Optimization**
 - options, 3-7
 - switches, 3-7
- Optimizer translations, 7-4**
- Options**
 - advanced, 3-6
 - command line, 3-2, B-1, 3-5
 - compatibility, 3-13
 - debugging, 3-13
 - disk usage, 3-5
 - fast calling sequence, 3-14
 - field, 3-2
 - line, 3-2
 - LINT, 3-9
 - message control, 3-6
 - message suppression, 3-12
 - optimization, 3-7
 - overlay support, 3-17
 - pipng, 4-1
 - redirection, 4-1
 - segment naming, 3-16
 - warning messages, 3-9
- outport, 6-43**
- outportb, 6-43**
- Overlay Manager, 3-17**
- Overlay support options, 3-17**
- Overview, 1-1**

P

- Parser translations, 7-4**
- Passing**
 - return values, 5-5
- peek, 6-44**
- peekb, 6-44**
- Pipes, 1-2, 4-2**
- Piping options, 4-1**

PL/M
calling sequence, 3-14, 7-49
functions, 7-49

Pointer
arithmetic, 3-4, 4-4
declarators, 7-36

Pointers, 3-16, 7-25

poke, 6-45

pokeb, 6-45

Portability
considerations, 7-75
enhancement, 3-13

Postfix
operators, 7-46

pow, 6-21

#pragma directives, 7-14

Prefix
decrement operator, 7-52
increment operator, 7-52

Preprocessing, 7-5

Preprocessor, 7-5
control options, 3-5
translations, 7-3

Primary expressions, 7-45

printf, 6-46

Product information, vi

Program
control, C-7
execution, 4-1
linking, 3-19

PUBLIC FAR procedure, 3-17

Punctuation, 7-22

putc, 6-51

putchar, 6-51

puts, 6-52

putw, 6-51

Q

qsort, 6-53

R

rand, 6-54

read, 6-55

realloc, 6-38

Redirection, 1-2
options, 4-1
parameters, 4-1

Reference material, vi

Relational operators, 7-57

rewind, 6-29

rindex, 6-32

Index-12

RUN command, 4-2

Runtime

library, 3-19, 6-1

support, 6-1

Runtime environment, 4-1

Runtime library, 1-2

S

scanf, 6-56

Scope of indentifiers, 7-43

Searching, C-7

Segment naming options, 3-16

Segmented memory architecture, 4-4

segread, 6-5, 6-60

Selection statements, 7-63

setbuf, 6-61

setjmp, 6-6, 6-63

setmem, 6-64

setvbuf, 6-61

Shift operators, 7-57

Simple assignment, 7-58

sin, 6-65

sinh, 6-67

sizeof keyword, 7-21, 7-54

Small model, 1-2, 3-1, 3-4, 5-9

Software installation, 1-2, 2-1

on a BTOS workstation, 2-1

on an XE520 master, 2-2

submit files, 1-2

Sorting, C-7

Source

file inclusion, 7-6

files Assembler, 3-1

text conventions, 7-15

Source files, 1-1

Specifiers

basic arithmetic type, 7-32

storage class, 7-30

type, 7-32

sprintf, 6-46

sqrt, 6-21

srand, 6-54

sscanf, 6-56

ssignal, 6-68

ssort, 6-53

Stack size, 3-20

Standard I/O, 6-2

Statements, 7-62

STDIO.H, 3-15

stime, 6-69

Storage class specifiers, 7-30

strcat, 6-70
strchr, 6-71
strcmp, 6-72
strcpy, 6-73
strcspn, 6-75
String handling, C-3
Strings, 7-21
strlen, 6-74
strncat, 6-70
strncmp, 6-72
strncpy, 6-73
strpbrk, 6-71
strrchr, 6-71
strspn, 6-75
strtod, 6-11
strtok, 6-76
strtol, 6-11
Structure members, 7-71
Structures, 7-26, 7-33
Submit files, 1-2
swab, 6-77

T

tan, 6-65
tanh, 6-67
Temporary disk storage, 1-1
Temporary files, 3-23
time, 6-78
Time management, C-8
toascii, 6-79
tolower, 6-79
toupper, 6-79
Translation
 limits, 7-3
 parser, 7-4
 phases, 7-3
 preprocessor, 7-3
Translations, 7-5
 optimizer, 7-4
Trigraphs, 7-22
Type
 basic, 7-23
 basic arithmetic, 7-32
 composite, 7-25
 enumerated, 7-24
 equivalence, 7-41
 floating, 7-24
 integral, 7-23
 modifiers, 7-27
 names, 7-40
 other, 7-62

Index-14

Type (continued)

- size of, 7-75
- specifiers, 7-32
- void, 7-24

Types, 7-23

U

Uidentifier, 3-5

- exit, 6-20
- toascii, 6-79
- tolower, 6-79
- toupper, 6-79

Unary arithmetic operators, 7-53

Unary operators, 7-52

ungetc, 6-80

Unions, 7-26, 7-33

UNIX

- compatible I/O, 6-2
- operating system, 1-1
- pipes, 7-3
- pipes, emulation of, 4-2

unlink, 6-81

Unsigned quantities, 7-24

V

VARARGSn comment, 3-22

vfprintf, 6-82

vfscanf, 6-83

Void, 7-24, 7-35

Void type, 7-24

vprintf, 6-82

vscanf, 6-83

vsprintf, 6-82

vsscanf, 6-83

W

Warning messages, 3-6, A-19

write, 6-84

X

XE520 master system, 1-1

8086 support, 3-6

8087 support, 3-7

Help Us To Help You

Publication Title _____

Form Number _____

Date _____

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

Addition

Deletion

Revision

Error

Comments _____

Name _____

Title _____

Company _____

Address (Street, City, State, Zip) _____

Telephone Number _____

Help Us To Help You

Publication Title _____

Form Number _____

Date _____

Unisys Corporation is interested in your comments and suggestions regarding this manual. We will use them to improve the quality of your Product Information. Please check type of suggestion:

Addition

Deletion

Revision

Error

Comments _____

Name _____

Title _____

Company _____

Address (Street, City, State, Zip) _____

Telephone Number _____



BUSINESS REPLY MAIL

First Class

Permit No. 817

Detroit, MI 48232

Postage Will Be Paid By Addressee

Unisys Corporation
ATTN: Corporate Product Information
P.O. Box 418
Detroit, MI 48232-9975 USA



BUSINESS REPLY MAIL

First Class

Permit No. 817

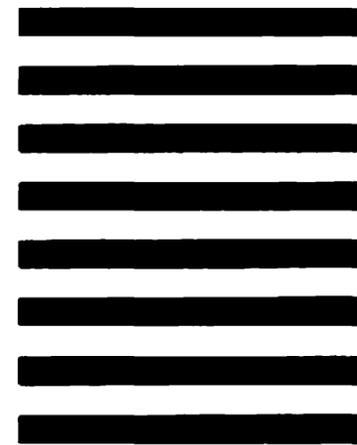
Detroit, MI 48232

Postage Will Be Paid By Addressee

Unisys Corporation
ATTN: Corporate Product Information
P.O. Box 418
Detroit, MI 48232-9975 USA



No Postage
necessary
if mailed in the
United States



No Postage
necessary
if mailed in the
United States

