Microware C Compiler User's Guide

for OS-9

Microware C Compiler User's Guide: for OS-9

Acknowledgements

The OS-9 C Compiler was written by James McCosh with OS-9 implementation assistance from Terry Crane and Kim Kempf. The Relocatable Assembler, Linker, and Profiler was edited by Wes Camden and Ken Kaplan.

Differences between Versions 1.1 and 1.0

Important Notice - Please Read Carefully

This package contains the OS-9 C Compiler Version 1.1. Many improvements and bug fixes have been incorporated since the V1.0 release. If you are upgrading from V1.0, be *absolutely sure* to install *all* the files from the V1.1 disks. None of the compiler sections or the library is compatible with V1.0 files. Any ".r" or ".a" files produced by the V1.0 compiler should not be assembled or linked with any ".a" or ".r" files produced by the V1.1 compiler. To be safe, recompile/reassemble *all* ".a" and ".r" files with V.1.1.

This update include appendices for the C Compiler User's Guide describing the compiler error messages, compiler phase command lines, interfacing C functions to BASIC09, and an overview of the relocating macro assembler.

The remainder of this notice describes the changes made since V1.0.

General:	The compiler code generator and c.opt have been improved to produce even smaller object code. This, and improved source coding, has resulted in a 1 page decrease in the size of c.comp and a 4 page decrease in c.pass1.
Executives (cc1 and cc2):	-x appearing on the cc1 command line causes the compiler to make the c.com command file but not execute it. -q on the cc2 command line causes the compiler to suppress filename and compiler phase messages.
Preprocessor (c.prep):	C.prep now prints a fatal error if a line exceeds 255 bytes.
Compiler (c.comp, c.pass1, c.pass2):	C.pass1 float/double conversion is now done properly rather than reporting error 7.
	Direct and static direct storage classes may now be initialized.
	Sizeof operator now reports an error when applied to an undefined identifier. Sizeof now allows any expression inside of parenthesis. Previously, only primaries were allowed.
	Various code generation problems involving certain long and floating operations have been fixed.
Optimizer (c.opt):	C.opt has been improved to use much less dynamic memory while performing optimizations.
	Some branches were erroneously converted to short branches when they should have been long.
Assembler (c.asm):	C.asm can now handle direct-page initialized data.
	Some out-of-range short branches were not detected.
	VSECT syntax changed to allow direct-page initializers. This make V1.0 assembly file incompatible with V1.1.
	Macro and repeat block facilities have been added.

Linker (c.link):

C.link can now handle direct-page initialized data.

C.link will now report if the direct page allocations exceeds 256 bytes.

C.link is about three times faster using the improved V1.1 standard library.

C.link can now output modules that can be entered by the BASIC09 "RUN" command.

Library (clib.l):

The standard library FILE structure has been changed to allow specification of buffersize for a file. In V1.0, the buffersize was fixed at 256 bytes. A new element of the FILE struct (_bufsiz) contains the desired buffer size. This may be used as follows:

```
main()
{
     FILE *fp;

     fp=fopen("file","r");
     fp->_bufsiz = 1024;

.....
}
```

A few restrictions exist on the use of this parameter. Initially the _bufsiz value is 0. The library routines will assign a buffer of 256 bytes to the file upon initial read or write. If the value is non-zero and the fp has not previously been accessed, that value is used as the buffersize. Note that due to the way the library routines work, once a buffer of a given size is allocated to an fp, a larger size cannot be used, even if the file is closed. Note that the buffers are allocated from the ibrk() so enough extra memory must be allocated by the linker to handle the bigger buffers.

Since the size of the _iobuf struct (FILE) in stdio.h has changed, all .r files must be re-compiled using the new header file.

Cstart.r can now handle direct page data initialization.

Fseek() now does not cause the buffer to be re-filled if the seek destination is already in the buffer.

Getc() now does "I$READ" on unbuffered SCF devices rather than "I$READLN".

Getc() performed on "stdin" flushes the "stdout" buffer.

Printf() has been changed to not flush the "stdout" buffer before returning.

Chown() has been fixed to not wipe out disks.

Toascii() has been added to stdio.h

Calls to scanf() now do not cause the linker to reports unresolved references to toupper() and tolower().

The floating point routines now report errors 40, 41, and 42 for floating point over/underflow, divide by zero, and float/long conversion instead of error #007.

Chapter 1. The C Compiler System

1.1. Introduction

The "C" programming language is rapidly growing in popularity and seems destined to become one of the most popular programming languages used for microcomputers. The rapid rise in the use of C is not surprising. C is an incredibly versatile and efficient language that can handle tasks that previously would have required complex assembly language programming.

C was originally developed at Bell Telephone Laboratories as an implementation language for the UNIX operating system by Brian Kernighan and Dennis Ritchie. They also wrote a book titled “The C Programming Language” which is universally accepted as the standard for the language. It is an interesting reflection on the language that although no formal industry-wide “standard” was ever developed for C, programs written in C tend to be far more portable between radically different computer systems as compared to so-called “standardized” languages such as BASIC, COBOL, and PASCAL. The reason C is so portable is that the language is so inherently expandable that is some special function is required, the user can create a portable extension to the language, as opposed to the common practice of adding additional statements to the language. For example, the number of special-purpose BASIC dialects defies all reason. A lesser factor is the underlying UNIX operating system, which is also sufficiently versatile to discourage bastardization of the language. Indeed, standard C compilers and Unix are intimately related.

Fortunately, the 6809 microprocessor, the OS-9 operating system, and the C language form an outstanding combination. The 6809 was specifically designed to efficiently run high-level languages, and its stack-oriented instruction set and versatile repertoire of addressing modes handle the C language very well. As mentioned previously, UNIX and C are closely related, and because OS-9 is derived from UNIX, it also supports C to the degree that almost any application written in C can be transported from a UNIX system to an OS-9 system, recompiled, and correctly executed.

1.2. The Language Implementation

OS-9 C is implemented almost exactly as described in 'The C Programming Language' by Kernighan and Ritchie (hereafter referred to as K&R).

Allthough this version of C follows the specification faithfully, there are some differences. The differences mostly reflect parts of C that are obsolete or the constraints imposed by memory size limitations.

1.3. Differences from the K & R Specification

- Bit fields are not supported.
- Constant expressions for initializers may include arithmetic operators only if all the operands are of type INT or CHAR.
- The older forms of assignment operators, '=+' or '=*', which are recognized by some C compilers, are not supported. You must use the newer forms '+=','*=' etc.
- "#ifdef (or #ifndef) ...[#else...] #endif" is supported but "#if <constant expression>" is not.
- It is not possible to extend macro definitions or strings over more than one line of source code.
- The escape sequence for new-line '\n' refers to the ASCII carriage return character (used by OS-9 for end-of-line), not linefeed. (hex 0A). Programs which use '\n' for end-of-line (which includes all programs in K & R), will still work properly.

1.4. Enhancements and Extensions

1.4.1. The “Direct” Storage Class

The 6809 microprocessor instructions for accessing memory via an index register or the stack pointer can be relatively short and fast when they are used in C programs to access "auto" (function local)

variables or function arguments. The instructions for accessing global variables are normally not so nice and must be four bytes long and correspondingly slow. However, the 6809 has a nice feature which helps considerably. Memory, anywhere in a single page (256 byte block), may be accessed with fast, two byte instructions. This is called the "direct page", and at any time its location is specified by the contents of the "direct page register" within the processor. The linkage editor sorts out where this could be, and it need not concern the programmer, who only needs to specify for the compiler which variables should be in the direct page to give the maximum benefit in code size and execution speed.

To this end, a new storage class specifier is recognized by the compiler. In the manner of K & R page 192, the sc-specifier list is extended as follows:

Sc-specifier:	auto	
	static	
	extern	
	register	
	typedef	
	direct	(extension)
	extern direct	(extension)
	static direct	(extension)

The new key word may be used in place of one of the other sc-specifiers, and its effect is that the variable will be placed in the direct page. "DIRECT" creates a global direct page variable. "EXTERN DIRECT" references an EXTERNAL-type direct page variable; and "STATIC DIRECT" creates a local direct page variable. These new classed may not be used to declare function arguments. "Direct" variables can be initialized but will, as with other variables not explicitly initialized, have the value zero at the start of program execution. 255 bytes are available in the direct page (the linker requires one byte). If all the direct variables occupy less than the full 255 bytes, the remaining global variables will occupy the balance and memory above if necessary. If too many bytes or storage are requested in the direct page, the linkage editor will report an error, and the programmer will have to reduce the use of DIRECT-type variables to fit the 256 bytes addressable by the 6809.

It should be kept in mind that "direct" is unique to this compiler, and it may not be possible to transport programs written using "direct" to other environments without modification.

1.4.2. Embedded Assembly Language

As versatile as C is, occasionally there are some things that can only be done (or done at maximum speed) in assembly language. The OS-9 C compiler permits user-supplied assembly-language statements to be directly embedded in C source programs.

A line beginning with "#asm" switches the compiler into a mode which passes all subsequent lines directly to the assembly-language output, until a line beginning with "#endasm" is encountered. "#endasm" switches the mode back to normal. Care should be exercised when using this directive so that the correct code section is adhered to. Normal code from the compiler is in the PSECT (code) section. If your assembly code uses the VSECT (variable) section, be sure to put a ENDSECT directive at the end to leave the state correct for following compiler generated code.

1.4.3. Control Character Escape Sequences

The escape sequences for non-printing characters in character constants and strings (see K & R page 181) are extended as follows:

```
linefeed (LF):  \l (lower case 'ell')
```

This is to distinguish LF (hex 0A) from \n which on OS-9 is the same as \r (hex 0D).

```
bit patterns:  \NNN    (octal constant)
               \dNNN   (decimal constant)
               \xNN    (hexadecimal constant)
```

For example, the following all have a value of 255 (decimal):

```
\377          \xff             \d255
```

1.5. Implementation-dependent Characteristics

K & R frequently refer to characteristics of the C language whose exact operations depend on the architecture and instruction set of the computer actually used. This section contains specific information regarding this version of C for the 6809 processor.

1.5.1. Data Representation and Storage Requirements

Each variable type requires a specific amount of memory for storage. The sizes of the basic types in bytes are as follows:

Data Type	Size	Internal Representation
CHAR	1	two's complement binary
INT	2	two's complement binary
UNSIGNED	2	unsigned binary
LONG	4	two's complement binary
FLOAT	4	binary floating point (see below)
DOUBLE	8	binary floating point (see below)

This compiler follows the PDP-11 implementation and format in that CHARs are converted to INTs by sign extension, "SHORT" or "SHORT INT" means INT, "LONG INT" means LONG and "LONG FLOAT" means DOUBLE. The format for DOUBLE values is as follows:

```
(low byte)                                  (high byte)
+-+-----------------------------------------+----------+
! !     seven byte                          ! 1 byte   !
! !      mantissa                          ! exponent !
+-+-----------------------------------------+----------+
 ^ sign bit
```

The form of the mantissa is sign and magnitude with an implied "1" bit at the sign bit position. The exponent is biased by 128. The format of a FLOAT is identical, except that the mantissa is only three bytes long. Conversion from DOUBLE to FLOAT is carried out by truncating the least significant (right-most) four bytes of the mantissa. The reverse conversion is done by padding the least significant four mantissa bytes with zeros.

1.5.2. Register Variables

One register variable may be declared in each function. The only types permitted for register variables are int, unsigned and pointer. Invalid register variable declarations are ignored; i.e. the storage class is made auto. For further details see K & R page 81.

A considerable saving in code size and speed can be made by judicious use of a register variable. The most efficient use is made of it for a pointer or a counter for a loop. However, if a register variable is

used for a complex arithmetic expression, there is no saving. The "U" register is assigned to register variables.

1.5.3. Access To Command Line Parameters

The standard C arguments "argc" and "argv" are available to "main" as described in K & R page 110. The start-up routine for C programs ensures that the parameter string passed to it by the parent process is converted into null-terminated strings as expected by the program. In addition, it will run together as a single argument any strings enclosed between single or double quotes (""" or "'"). If either is part of the string required, then the other should be used as a delimiter.

1.6. System Calls and the Standard Library

1.6.1. Operating System Calls

The system interface supports almost all the system calls of both OS-9 and UNIX. In order to facilitate the portability of programs from UNIX, some of the calls use UNIX names rather than OS-9 names for the same function. There are a few UNIX calls that do not have exactly equivalent OS-9 calls. In these cases, the library function simulates the function of the corresponding UNIX call. In cases where there are OS-9 calls that do not have UNIX equivalents, the OS-9 names are used. Details of the calls and a name cross-reference are provided in the "C System Calls" section of this manual.

1.6.2. The Standard Library

The C compiler includes a very complete library of standard functions. It is essential for any program which uses functions from the standard library to have the statement:

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

See the "C Standard Library" section of this manual for details on the standard library functions provided.

IMPORTANT NOTE: If output via printf(), fprintf() or sprintf() of long integers is required, the program MUST call "pflinit()" at some point; this is necessary so that programs not involving LONGS do not have the extra LONGs output code appended. Similarly, if FLOATs or DOUBLEs are to be printed, "pffinit()" MUST be called. These functions do nothing; existence of calls to them in a program informs the linker that the relevant routines are also needed.

1.7. Run-time Arithmetic Error Handling

K & R leave the treatment of various arithmetic errors open, merely saying that it is machine dependent. This implementation deal with a limited number of error conditions in a special way; it should be assumed that the results of other possible errors are undefined.

Three new system error numbers are defined in <errno.h>:

```
#define  EFPOVR  40   /* floating point overflow of underflow */
#define  EDIVERR 41   /* division by zero */
#define  EINTERR 42   /* overflow on conversion of floating point
                         to long integer */
```

If one of these conditions occur, the program will send a signal to itself with the value of one of these errors. If not caught or ignored, the will cause termination of program with an error return to the parent process. However, the program can catch the interrupt using "signal()" or "intercept()" (see C System Calls), and in this case the service routine has the error number as its argument.

1.8. Achieving Maximum Program Performance

1.8.1. Programming Considerations

Because the 6809 is an 8/16 bit microprocessor, the compiler can generate efficient code for 8 and 16 bit objects (CHARs, INTs, etc.). However, code for 32 and 64 bit values (LONGs, FLOATs, DOUBLEs) can be at least four times longer and slower. Therefore don't use LONG, FLOAT, or DOUBLE where INT or UNSIGNED will do.

The compiler can perform extensive evaluation of constant expressions provided they involve only constants of type CHAR, INT, and UNSIGNED. There is no constant expression evaluation at compile-time (except single constants and "casts" of them) where there are constants of type LONG, FLOAT, or DOUBLE, therefore, complex constant expressions involving these types are evaluated at run time by the compiled program. You should manually compute the value of constant expressions of these types if speed is essential.

1.8.2. The Optimizer Pass

The optimizer pass automatically occurs after the compilation passes. It reads the assembler source code text and removes redundant code and searches for code sequences that can be replaced by shorter and faster equivalents. The optimizer will shorten object code by about 11% with a significant increase in program execution speed. The optimizer is recommended for production versions of debugged programs. Because this pass takes additional time, the "-O" compiler option can be used to inhibit it during error-checking-only compilations.

1.8.3. The Profiler

The profiler is an optional method used to determine the frequency of execution of each function in a C program. It allows you to identify the most-frequently used functions where algorithmic or C source code programming improvements will yield the greatest gains.

When the "-P" compiler option is selected, code is generated at the beginning of each function to call the profiler module (called "_prof"), which counts invocations of each function during program execution. When the program has terminated, the profiler automatically prints a list of all functions and the number of times each was called. The profiler slightly reduces program execution speed. See "prof.c" source for more information.

1.9. C Compiler Component Files and File Usage

Compilation of a C program by cc requires that the following files be present in the current execution directory (CMDS).

Table 1.1. OS-9 Level I Systems

cc1	compiler executive program
c.prep	macro pre-processor
c.pass1	compiler pass 1
c.pass2	compiler pass 2
c.opt	assembly code optimizer
c.asm	relocating assembler
c.link	linkage editor

Table 1.2. OS-9 Level II Systems

cc2	compiler executive program

c.prep	macro pre-processor
c.comp	compiler proper
c.opt	assembly code optimizer
c.asm	relocating assembler
c.link	linkage editor

In addition a file called "clib.l" contains the standard library, math functions, and systems library. The file "cstart.r" is the setup code for compiled programs. Both of these files must be located in a directory named "LIB" on the system's default mass storage device, which is specified in the OS-9 "INIT" module and is usually the disk drive the system is booted from.

If, when specifying "#include" files for the pre-processor to read in, the programmer uses angle brackets, "<" and ">", instead of parentheses, the file will be sought starting at the "DEFS" directory on whichever drive is the default system drive for the system running.

1.9.1. Temporary Files

A number of temporary files are created in the current data directory during compilation, and it is important to ensure that enough space is available on the disk drive. As a rough guide, at least three times the number of blocks in the largest source file (and its included files) should be free.

The identifiers "etext", "edata", and "end" are predefined in the linkage editor and may be used to establish the addresses of the end of executable text, initialized data, and uninitialized data respectively.

1.10. Running the Compiler

The are two commands which invoke distinct versions of the compiler. "cc1" is for OS-9 Level I which uses a two pass compiler, and, "cc2" is for Level II which causes a single pass version. Both versions of the compiler works identically, the main difference is that cc1 has been divided into two passes to fit the smaller memory size of OS-9 Level I systems. In the following text, "cc" refers to either "cc1" or "cc2" as appropriate for your system. The syntax of the command line which calls the compiler is:

`cc` [option-flags] *`file`*...

One file at a time can be compiled, or a number of files may be compiled together. The compiler manages the compilation up to four stages: pre-processor, compilation to assembler code, assembly to relocatable code, and linking to binary executable code (in OS-9 memory module format).

The compiler accepts three types of source files, provided each name on the command line has the relevant postfix as shown below. Any of the above file types may be mixed on the command line.

Table 1.3. File Name Suffix Conventions

Suffix	Usage
.c	C source file
.a	assembly language source file
.r	relocatable module
none	executable binary (OS-9 memory module)

There are two modes of operation: multiple source file and single source file. The compiler selects the mode by inspecting the command line. The usual mode is single source and is specified by having only one source file name on the command line. Of course, more than one source file may be compiled together by using the "#include" facility in the source code. In this mode, the compiler will use the name obtained by removing the postfix from the name supplied on the command line, and the output file (and the memory module produced) will have this name. For example:

```
cc prg.c
```

will leave an executable file called "prg" in the current execution directory.

The multiple source mode is specified by having more than one source file name on the command line. In this mode, the object code output file will have the name "output" in the current execution directory, unless a name is given using the "-f=" option (see below). Also, in multiple source mode, the relocatable modules generated as intermediate files will be left in the same directories as their corresponding source files with the postfixes changed to ".r". For example:

```
cc prg1.c /d0/fred/prg2.c
```

will leave an executable file called "output" in the current execution directory, one file called "prg1.r" in the current data directory, and "prg2.r" in "/d0/fred".

1.11. Compiler Option Flags

The compiler recognizes several command-line option flags which modify the compilation process where needed. All flags are recognized before compilation commences so the flags may be placed anywhere on the command line. Flags may be ran together as in "-ro", except where a flag is followed by something else; see "-f=" and "-d" for examples.

`-A` suppresses assembly, leaving the output as assembler code in a file whose name is postfixed ".a".

`-E`=<number> Set the edition number constant byte to the number given. This is an OS-9 convention for memory modules.

`-O` inhibits the assembly code optimizer pass. The optimizer will shorten object code by about 11% with a comparable increase in speed and is recommended for production versions of debugged programs.

`-P` invokes the profiler to generate function frequency statistics after program execution.

`-R` suppresses linking library modules into an executable program. Outputs are left in files with postfixes ".r".

`-M`=<memory size> will instruct the linker to allocate <memory size> for data, stack, and parameter area. Memory size may be expressed in pages (an integer) or in kilobytes by appending "k" to an integer. For more details of the use of this option, see the "Memory Management" section of this manual.

`-L`=<filename> specifies a library to be searched by the linker before the Standard Library and system interface.

`-F`=<path> overrides the above output file naming. The output file will be left with <filename> as its name. This flag does not make sense in multiple source mode, and either the -a or -r flag is also present. The module will be called the last name in <path>.

`-C` will output the source code as comments with the assembler code.

`-S` stops the generation of stack-checking code. -S should only be used with great care when the application is extremely time-critical and when the use of the stack by compiler generated code is fully understood.

`-D`<identifier> is equivalent to "#define <identifier>" written in the source file. -D is useful where different versions of a program are maintained in one source file and differentiated by means of the "#ifdef" of "#ifndef" pre-processor directives. If the <identifier> is used as a macro for expansion by the pre-processor, "1"(one) will be the expanded "value" unless the form "-d<identifier>=<string>" is used in which case the expansion will be <string>.

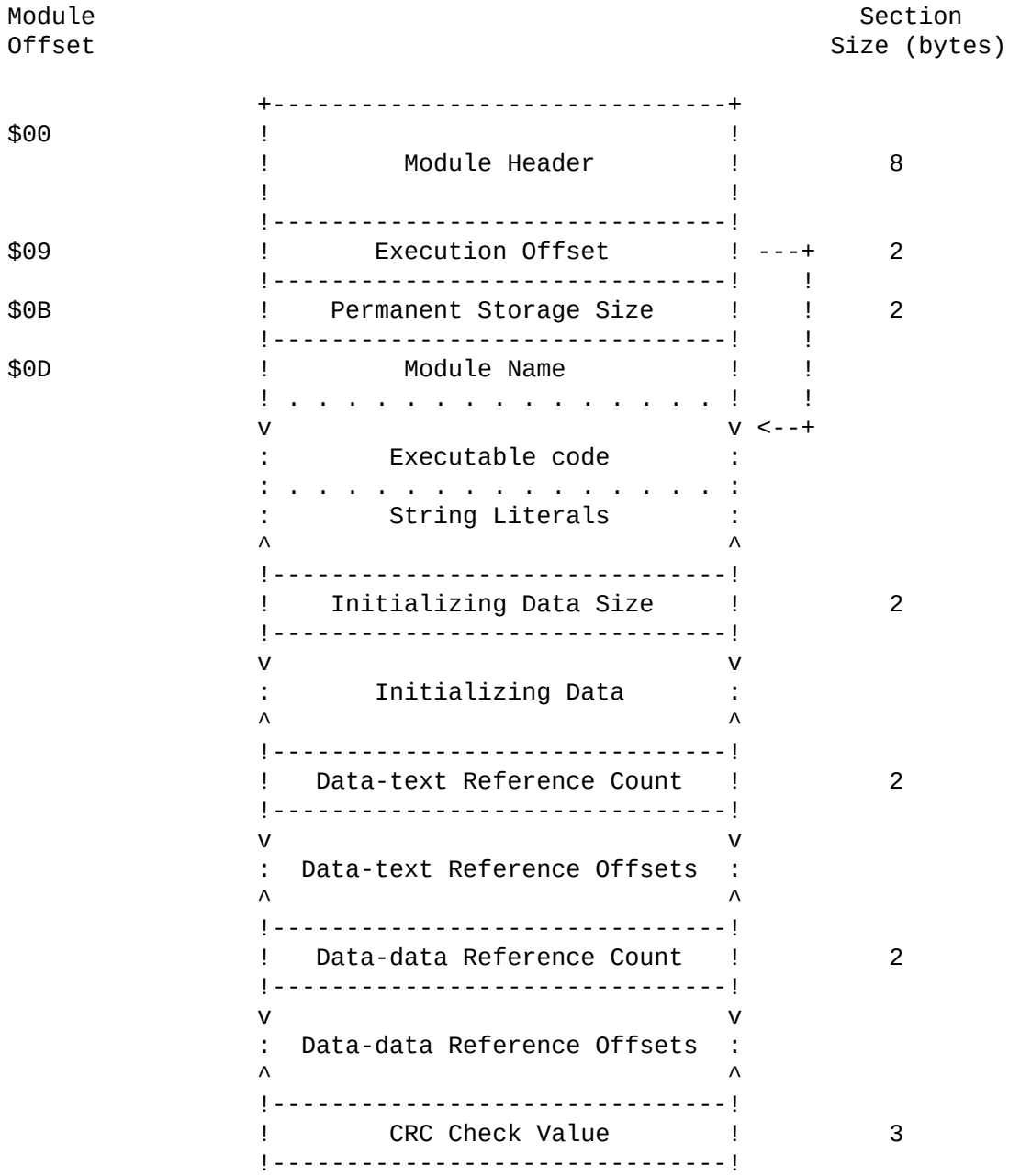
Table 1.4. Command Line and Option Flag Examples

command line	action	output file(s)
cc prg.c	compile to an executable program	prg
cc prg.c -a	compile to assembly language source code	prg.a
cc prg.c -r	compile to relocatable module	prg.r
cc prg1.c prg2.c prg3.c	compile to executable program	prg1.r, prg2.r, prg3.r, output
cc prg1.c prg2.a prg3.r	compile prg1.c, assemble prg2.a and combine all into and executable program	prg1.r, prg2.r
cc prg1.c prg2.c -a	compile to assembly language source code	prg1.a, prg2.a
cc prg1.c prg2.c -f=prg	compile to executable program	prg

Chapter 2. Characteristics of Compiled Programs

2.1. The Object Code Module

The compiler produces position-independent, reentrant 6809 code in a standard OS-9 memory module format. The format of an executable program module is shown below. Detailed descriptions of each section of the module are given on following pages.



2.1.1. Module Header

This is a standard module header with the type/language byte set to $11 (Program + 6809 Object Code), and the attribute/revision byte set to $81 (Reentrant + 1).

2.1.2. Execution Offset

Used by OS-9 to locate where to start execution of the program.

2.1.3. Storage Size

Storage size is the initial default allocation of memory for data, stack, and parameter area. For a full description of memory allocation, see the section entitled “Memory Management” located elsewhere in this manual.

2.1.4. Module Name

Module name is used by OS-9 to enter the module in the module directory. The module name is followed by the edition byte encoded in cstart. If this situation is not desired it may be overridden by the -E= option in cc.

2.1.5. Information

Any strings preceded by the directive "info" in an assembly code file will be placed here. A major use of this facility is to place in the module the version number and/or a copyright notice. Note that the '#asm' pre-compiler instruction may be used in a C source file to enable the inclusion of this directive in the compiler-generated assembly code file.

2.1.6. Executable Code

The machine code instructions of the program.

2.1.7. String Literals

Quoted string in the C source are placed here. They are in the null-terminated form expected by the functions in the Standard Library. NOTE: the definition of the C language assumes that strings are in the DATA area and are therefore subject to alteration without making the program non-reentrant. However, in order to avoid the duplication of memory requirements which would be necessary if they were to be in the data area, they are placed in the TEXT (executable) section of the module. Putting the strings in the executable section implies that no attempt should be made by a C programmer to alter string literals. They should be copied out first. The exception that proves the rule is the initialization of an array of type char like this:

```
char message[] = "Hello world\n";
```

The string will be found in the array 'message' in the data area and can be altered.

2.1.8. Initializing Data and its Size

If a C program contains initializers, the data for the initial values of the variables is placed in this section. The definition of C states that all uninitialized global and static variables have the value zero when the program starts running, so the startup routine of each C program first copies the data from the module into the data area and then clears the rest of the data memory to nulls.

2.1.9. Data References

No absolute addresses are known at compile time under OS-9, so where there are pointer values in the initialization data, they must be adjusted at run time so that they reflect the absolute values at that time. The startup routine uses the two data reference tables to locate the values that need alteration and adjusts them by the absolute values of the bases of the executable code and data respectively.

For example, suppose there are the following statements in the program being compiled:

```
char *p = "I'm a string!";
char **q = &p;
```

These declarations tell the compiler that there is to be a char pointer variable, 'p', whose initial value is the address of the string and a pointer to a char pointer, 'q', whose initial value is the address of 'p'. The variables must be in the DATA section of memory at run time because they are potentially alterable, but absolute addresses are not known until run time, so the values that 'p' and 'q' must have are not known at compile time. The string will be placed by the compiler in the TEXT section and will not be copied out to DATA memory by the startup routine. The initializing data section of the program module will contain entries for 'p' and 'q'. They will have as values the offsets of the string from the base of the TEXT section and the offset of the location of 'p' from the base of the DATA section respectively.

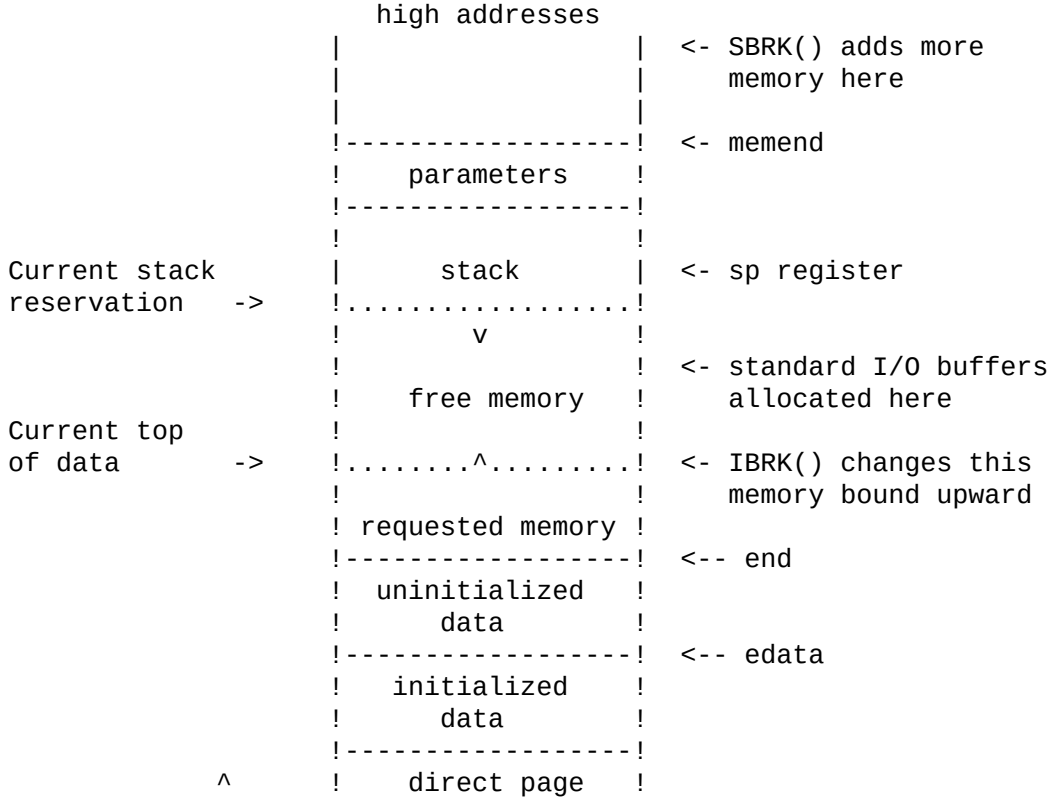
The startup routine will first copy all the entries in the initializing data section into their allotted places in the DATA section. Then it will scan the data-text reference table for the offsets of values that need to have the addresses of the base of the TEXT section added to them. Among these will be the "p" which, after updating, will point to the string which is in the TEXT section. Similarly, after a scan of the data-data references, "q" will point to (contain the absolute of) "p".

2.2. Memory Management

The C compiler and its support programs have default conditions such that the average programmer need not be concerned with details of memory management. However, there are situations where advanced programmers may wish to tailor the storage allocation of a program for special situations. The following information explains in detail how a C program's data area is allocated and used.

2.2.1. Typical C Program Memory Map

A storage area is allocated by OS-9 when the C program is executed. The layout of this memory is as follows:



```
dpsiz    !     variables    !
  v      +------------------+  <-- y,dp registers
             low addresses
```

The overall size of this memory area is defined by the "storage size" value stored in the program's module header. This can be overridden to assign the program additional memory if the OS-9 Shell "#" command is used.

The parameter area is where the parameter string from the calling process (typically the OS-9 Shell) is placed by the system. The initializing routine for C programs converts the parameters into null-terminated strings and makes pointers to them available to 'main()' via 'argc' and 'argv'.

The stack area is the currently reserved memory for exclusive use of the stack. As each C function is entered, a routine in the system interface is called to reserve enough stack space for the use of the function with an addition of 64 bytes. The 64 bytes are for the use of user-written assembly code functions and/or the system interface and/or arithmetic routines. A record is kept of the lowest address so far granted for the stack. If the area requested would not bring this lower then the C function is allowed to proceed. If the new lower limit would mean that the stack area would overlap the data area, the program stops with the message:

```
**** STACK OVERFLOW ****
```

on the standard error output. Otherwise, the new lower limit is set, and the C function resumes as before.

The direct page variables area is where variables reside that have been defined with the storage class 'direct' in the C source code or in the 'direct' segment in assembly code source. Notice that the size of this area is always at least one byte (to ensure that no pointer to a variable can have the value NULL or 0) and that it is not necessarily 256 bytes.

The uninitialized data area is where the remainder of the uninitialized program variables reside. These two areas are, in fact, cleared to all zeros by the program entry routine. The initialized data area is where the initialized variables of the program reside. There are two globally defined values which may be referred to: 'edata' and 'end', which are the addresses of one byte higher than the initialized data and one byte higher than the uninitialized data respectively. Note that these are not variables; the values are accessed in C by using the '&' operator as in:

```
high = &end;
low = &edata;
```

and in assembler:

```
leax   end,y
stx    high,y
```

The Y register points to the base of the data area and variables are addresses using Y-offset indexed instructions.

When the program starts running, the remaining memory is assigned to the "free" area. A program may call "ibrk()" to request additional working memory (initialized to zeros) from the free memory area. Alternatively, more memory can be dynamically obtained using the "sbrk()" which requests additional memory from the operating system and returns its lower bound. If this fails because OS-9 refuses to grant more memory for any reason "sbrk()" will return -1.

2.2.2. Compile Time Memory Allocation

If not instructed otherwise, the linker will automatically allocate 1k bytes more than the total size of the program's variables and strings. This size will normally be adequate to cover the parameter area,

stack requirements, and Standard Library file buffers. The allocation size may be altered when using the compiler by using the "-m" option on the command line. The memory requirements may be stated in pages, for example,

```
cc prg.c -m=2
```

which allocates 512 bytes extra, or in kilobytes, for example:

```
cc prg.c -m=10k
```

The linker will ignore the request if the size is less than 256 bytes.

The following rules can serve as a rough guide to estimate how much memory to specify:

1. The parameter area should be large enough for any anticipated command line string.
2. The stack should not be less than 128 bytes and should take into account the depth of function calling chains and any recursion.
3. All function arguments and local variables occupy stack space and each function entered needs 4 bytes more for the return address and temporary storage of the calling function's register variable.
4. Free memory is requested by the Standard Library I/O functions for buffers at the rate of 256 bytes per accessed file. The does not apply to the lower level service request I/O functions such as "open()", "read()" or "write()" not to "stderr" which is always un-buffered, but it does apply to both "stdin" and "stdout" (see the Standard Library documentation).

A good method for getting a feel for how much memory is needed by your program is to allow the linker to set the memory size to its usually conservative value. Then, if the program runs with a variety of input satisfactorily but memory is limited on the system, try reducing the allocation at the next compilation. If a stack overflow occurs or an "ibrk()" call returns -1, then try increasing the memory next time. You cannot damage the system by getting it wrong, but data may be lost if the program runs out of space at a crucial time. It pays to be in error on the generous side.

Chapter 3. C System Calls

This section of the C compiler manual is a guide to the system calls available from C programs.

It is *not* intended as a definitive description of OS-9 service requests as these are described in the OS-9 System Programmer's Manual. However, for most calls, enough information is available here to enable the programmer to write systems calls into programs without looking further.

The names used for the system calls are chosen so that programs transported from other machines or operating systems should compile and run with as little modification as possible. However, care should be taken as the parameters and returned values of some calls may not be compatible with those on other systems. Programmers that are already familiar with OS-9 names and values should take particular care. Some calls do not share the same names as the OS-9 assembly language equivalents. The assembly language equivalent call is shown, where there is one, on the relevant page of the C call description, and a cross-reference list is provided for those already familiar with OS-9 calls.

The normal error indication on return from a system call is a returned value of -1. The relevant error will be found in the predefined int "errno". Errno always contains the error from the last erroneous system call. Definitions for the errors for inclusion in the programs are in "<errno.h>".

In the "See Also" sections on the following pages, unless otherwise stated, the references are to other system calls.

Where "#include" files are shown, it is not mandatory to include them, but it might be convenient to use the manifest constants defined in them rather than integers; it certainly makes for more readable programs.

Name

Abort — stop the program and produce a core dump

Synopsis

```
abort();
```

Description

This call causes a memory image to be written out to the file "core" in the current directory, and then the program exits with a status of 1.

Name

Abs — Absolute value

Synopsis

```
abs(i);

int i;
```

Description

ABS returns absolute value of its integer operand.

Caveats

You get what the hardware gives on the largest negative number.

Name

Access — give file accessibility

Synopsis

```
access(fname, perm);

char *fname;
int perm;
```

Description

Access returns 0 if the access modes specified in "perm" are correct for the user to access "fname". -1 is returned if the file cannot be accessed.

The value for "perm" may be any legal OS-9 mode as used for "open()" or "creat()", it may be zero, which tests whether the file exists, or the path to it may be searched.

Caveats

NOTE that the "perm" value is *not* compatible with other systems.

Diagnostics

The appropiate error indication, if a value of -1 is returned, may be found in "errno".

Name

Chain — load and execute a new program

Synopsis

```
chain(modname, paramsize, paramptr, type, lang, datasize);

char *modname;
int paramsize;
char *paramptr;
int type;
int lang;
int datasize;
```

Assembler Equivalent

os9 F$CHAIN

Description

The action of F$CHAIN is described fully in the OS-9 documentation. Chain implements the service request as described with one important exception: chain will NEVER return to the caller. If there is an error, the process will abort and return to its parent process. It might be wise, therefore, for the programs to check the existence and access permissions of the module before calling chain. Permissions may be checked by using "modlink()" or "modload()" followed by an "munlink()".

"Modname" should point to the name of the desired module. "Paramsize is the length of the parameter string (which should normally be terminated with a "\n"), and "paramptr" points to the parameter string. "Type" is the module type as found in the module header (normally 1: program), and "lang" should match the language nibble in the module header (C programs have 1 for 6809 machine code here). "Datasize" may be zero, or it may contain the number of 256 byte pages to give to the new process as initial allocation of data memory.

Name

Chdir, Chxdir — change directory

Synopsis

```
chdir(dirname);

char *dirname;

chxdir(dirname);

char *dirname;
```

Assembler Equivalent

os9 I$CHGDIR

Description

These calls change the current data directory and the current execution directory, respectively, for the running task. "Dirname" is a pointer to a string that gives a pathname for a directory.

Diagnostics

Each call returns 0 after a successful call, or -1 if "dirname" is not a directory path name, or it is not searchable.

See Also

OS-9 shell commands "chd" and "chx".

Name

Chmod — change access permissions of a file

Synopsis

#include <modes.h>

```
chmod(fname, perm);

char *fname;
int perm;
```

Description

Chmod changes the permission bits associated with a file. "Fname" must be a pointer to a file name, and "perm" should contain the desired bit pattern,

The allowable bit patterns are defined in the include file as follows:

```
/* permissions */
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
```

```
#define  S_ISHARE  0x40     /* sharable */
#define  S_IFDIR   0x80     /* directory */
```

Only the owner or the super user may change the permissions of a file.

Diagnostics

A successful call returns 0. A -1 is returned if the caller is not entitled to change permissions of "fname" cannot be found.

See Also

OS-9 command "attr"

Name

Chown — change the ownership of a file

Synopsis

```
chown(fname, ownerid);

char *fname;
int ownerid;
```

Description

This call is available only to the super user. "Fname" is a pointer to a file name, and "ownerid" is the new user-id.

Diagnostics

Zero is returned from a successful call. -1 is returned from on error.

Name

Close — close a file

Synopsis

```
close(pn);

int pn;
```

Assembler Equivalent

os9 I$CLOSE

Description

Close takes a path number, "pn", as returned from system calls "open()", "creat()", or "dup()", and closes the associated file.

Termination of a task always closes all open files automatically, but it is necessary to close files where multiple files are opened by the task, and it is desired to re-use path numbers to avoid going over the system or process path number limit.

See Also

creat(), open(), dup()

Name

Crc — compute a cyclic redundancy count

Synopsis

```
crc(start, count, accum);

char *start;
int count;
char accum[3];
```

Assembler Equivalent

os9 F$CRC

Description

This call accumulates a crc into a three byte array at "accum" for "count" bytes starting at "start". All three bytes of "accum" should be initialized to 0xff before the first call to "crc()". However, repeated calls can be subsequently made to cover an entire module. If the result is to be used as an OS-9 module crc, it should have its bytes complemented before insertion at the end of the module.

Name

Creat — create a file

Synopsis

#include <modes.h>

```
creat(fname, perm);

char *fname;
int perm;
```

Assembler Equivalent

os9 I$CREATE

Description

Creat returns a path number to a new file available for writing, giving it the permissions specified in "perm" and making the task user the owner. If, however, "fname" is the name of an existing file, the file is truncated to zero length, and the ownership and permissions remain unchanged. NOTE, that unlike the OS-9 assembler service request, creat does not return an error if the file already exists. "Access()" should be used to establish the existence of a file if it is important that a file should not be overwritten.

It is unnecessary to specify writing permissions in "perm" in order to write to the file in the current task.

The permissions allowed are defined in the include file as follows:

```
#define  S_IPRM    0xff     /* mask for permission bits */
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
#define  S_ISHARE  0x40     /* sharable */
```

Directories may not be created with this call; use "mknod()" instead.

Diagnostics

This call returns -1 if there are too many files open. If the pathname cannot be searched, if permission to write is denied, or if the file exists and is a directory.

See Also

write(), close(), chmod()

Name

Defdrive — get default system drive

Synopsis

```
char *defdrive();
```

Description

A call to defdrive returns a pointer to a string containing the name of the default system drive. The method used is to consult the "Init" module for the default directory name. The name is copied to a static data area and a pointer to it is returned.

Diagnostics

-1 is returned if the "Init" module cannot be linked to.

Name

Dup — duplicate an open path number

Synopsis

```
dup(pn);

int pn;
```

Assembler Equivalent

os9 I$DUP

Description

Dup takes the path number, "pn", as returned from "open()" or "creat()" and returns another path number associated with the same file.

Diagnostics

A -1 is returned is the call fails because there are too many files open or the path nmber is invalid.

See Also

open(), creat(), close()

Name

Exit, _Exit — task termination

Synopsis

```
exit(status);

int status;

_exit(status);

int status;
```

Assembler Equivalent

os9 F$EXIT

Description

Exit is the normal means of terminating a task. Exit does any cleaning up operations required before terminating, such as flushing out any file buffers (see Standard i/o), but _exit does not.

A task finishing normally, that is returning from "main()", is equivalent to a call - "exit(0)".

The status passed to exit is available to the parent task if it is executing a "wait".

See Also

wait()

Name

Getpid — get the task id

Synopsis

```
getpid();
```

Assembler Equivalent

os9 F$ID

Description

A number unique to the current running task is often useful in creating names for temporary files. This call returns the task's system id (as returned to its parent by "os9fork").

Description

os9fork(), Standard Library function mktemp.

Name

Getstat — get file status

Synopsis

#include <sgstat.h>

/* code 0 */

```
getstat(code, filenum, buffer);
```

```
int code;
int filenum;
char *buffer;
/* codes 1 and 6 */

getstat(code, filenum);

int code;
int filenum;
/* code 2 */

getstat(code, filenum, size);

int code;
int filenum;
long *size;
/* code 5 */

getstat(code, filenum, pos);

int code;
int filenum;
long *pos;
```

Assembler Equivalent

os9 I$GETSTT

Description

A full description of getstat can be found in the OS-9 System Programmer's Manual.

"Code" must be the value of one of the standard codes for the getstat service request. "Filenum" must be the path number of an open file.

The form of the call depends on the value of "code".

Code 0:	"Buffer" must be the address of a 32 byte buffer into which the relevant status packet is copied. The header file has the definitions of the various file and device structures for use by the program.
Code 1:	Code 1 only applies to SCF devices and to test for data available. The return value is zero if there is data available. -1 is returned if there is no data.
Code 2:	"Size" should be the address of a long integer into which the current file size is placed. The return value of the function is -1 on error and 0 on success.
Code 5:	"Pos" should be the address of a long integer into which the current file position is placed. The return value of the function is -1 on error and 0 on success.
Code 6:	Returns -1 on EOF and error and 0 on success.

NOTE that when one of the previous calls returns -1, then actual error is returned in errno.

Name

Getuid — return user id

Synopsis

```
getuid();
```

Assembler Equivalent

os9 F$ID

Description

Getuid returns the real user id of the current task (as maintained in the password file).

Name

Intercept — set function for interrupt processing

Synopsis

```
intercept((* func));

int (* func) (int);
```

Assembler Equivalent

os9 F$ICPT

Description

Intercept instructs OS-9 to pass control to the function "func" when an interrupt (signal) is received by the current process.

If the interrupt processing function has an argument, it will contain the value of the signal received. On return from "func", the process resumes at the point in the program where it was interrupted by the signal. "Interrupt()" is an alternative to the use of "signal()" to process interrupts.

As an example, suppose we wish to ensure that a partially completed output file is deleted if an interrupt is received. The body of the program might include:

```
char *temp_file = "temp"; /* name of temporary file */
int pn=0;                 /* path number */
int intrupt();            /* predeclaration */

...

intercept(intrupt);       /* route interrupt processing */
pn = creat(temp_file,3);  /* make a new file */

...

write(pn,string,count);   /* write string to temp file */

...

close(pn);
pn=0;

...
```

The interrupt routine might be coded:

```
intrupt(sig);
{
     if (pn){ /* only done if pn refers to an open file */
          close(pn);
          unlink(temp_file); /* delete */
     }
exit(sig);
}
```

Caveats

"Intercept()" and "signal()" are mutually incompatible so that calls to both must not appear in the same program. The linker guards against this by giving an "entry name clash - _sigint" error if it is attempted.

See Also

signal()

Name

Kill — send an interrupt to a task

Synopsis

#include <signal.h>

```
kill(tid, interrupt);

int tid;
int interrupt;
```

Description

Kill sends the interrupt type "interrupt" to the task with id "tid".

Both tasks, sender and receiver, must have the same user id unless the user is the super user.

The include file contains definitions of the defined signals as follows:

```
       /* OS-9 signals */
#define        SIGKILL 0   /* system abort (cannot be caught or ignored)*/
#define        SIGWAKE 1   /* wake up */
#define        SIGQUIT 2   /* keyboard abort */
#define        SIGINT  3   /* keyboard interrupt */
```

Other user-defined signals may, of course, be sent.

Diagnostics

Kill returns 0 from a successful call and -1 if the task does not exist, the effective user ids do not match, or the user is not the system manager.

See Also

signal(), OS-9 shell command "kill"

Name

Lseek — position in file

Synopsis

```
lseek(pn, position, type);

int pn;
long position;
int type;
```

Assembler Equivalent

os9 I$SEEK

Description

The read or write pointer for the open file with the path number, "pn", is positioned by lseek to the specified place in the file. The "type" indicates from where "position" is to be measured: if 0, from the beginning of the file, if 1, from the current location, or if 2, from the end of the file.

Seeking to a location beyond the end of a file open for writing and then writing to it, creates a "hole" in the file which appears to be filled with zeros from the previous end to the position sought.

The returned value is the resulting position in the file unless there is an error, so to find out the current position use

lseek(pn,0l,1);

Caveats

The argument "position" *must* be a long integer. Constants should be explicitly made long by appending an "l", as above, and other types should be converted using a cast:

e.g. lseek(pn,(long)pos,1);

Notice also, that the return value from lseek is itself a long integer.

Diagnostics

-1 is returned if "pn" is a bad path number, or attempting to seek to a position before the beginning of a file.

See Also

creat(), open(), Standard Library function "fseek"

Name

Mknod — create a directory

Synopsis

#include <modes.h>

```
mknod(fname, desc);

char *fname;
int desc;
```

Assembler Equivalent

os9 I$MAKDIR

Description

This call may be used to create a new directory. "Fname" should point to a string containing the desired name of the directory. "Desc" is a descriptor specifying the desired mode (file type) and permissions of the new file.

The include file defines the possible values for "desc" as follows:

```
#define  S_IREAD   0x01     /* owner read */
#define  S_IWRITE  0x02     /* owner write */
#define  S_IEXEC   0x04     /* owner execute */
#define  S_IOREAD  0x08     /* public read */
#define  S_IOWRITE 0x10     /* public write */
#define  S_IOEXEC  0x20     /* public execute */
#define  S_ISHARE  0x40     /* sharable */
```

Diagnostics

Zero is returned if the directory has been successfully made; -1 if the file already exists.

See Also

OS-9 command "makdir"

Name

Modload — return a pointer to a module structure

Synopsis

#include <module.h>

```
mod_exec *modlink(modname, type, language);

char *modname;
int type;
int language;

mod_exec *modload(modname, type, language);

char *modname;
int type;
int language;
```

Assembler Equivalent

os9 F$LINK

os9 F$LOAD

Description

Each of these calls return a pointer to an OS-9 memory module.

Modlink will search the module directory for a module with the same name as "modname" and, if found, increment its link count.

Modload will open the file which has the path list specified by "filename" and loads modules from the file adding them to the module directory. The returned value is a pointer to the first module loaded.

Above, each is shown as returning a pointer to an executable module, but it will return a pointer to whatever type of module is found.

Diagnostics

-1 is returned on error.

See Also

munlink()

Name

Munlink — unlink a module

Synopsis

#include <module.h>

```
munlink(mod);

mod_exec *mod;
```

Assembler Equivalent

os9 F$UNLINK

Description

This call informs the system that the module pointed to by "mod" is no longer required by the current process. Its link count is decremented, and the module is removed from the module directory if the link count reaches zero.

See Also

modlink(), modload()

Name

_os9 — system call interface from C programs

Synopsis

#include <os9.h>

```
_os9(code, reg);

char code;
struct registers *reg;
```

Description

_os9 enables a programmer to access virtually any OS-9 system call directly from a C program without having to resort to assembly language routines.

`Code` is one of the codes that are defines in os9.h. os9.h contains codes for the F$ and I$ function/service requests, and it also contains getstt, setstt, and error codes.

The input registers(reg) for the system calls are accessed by the following structure that is defined in os9.h:

```
struct registers {
     char rg_cc,rg_a,rg_b,rg_dp;
     unsigned rg_x,rg_y,rg_u;
};
```

An example program that uses _os9 is presented on the following page.

Diagnostics

-1 is returned if the OS-9 call failed. 0 is returned on success.

Program Example

```
#include <os9.h>
#include <modes.h>

/* this program does an I$GETSTT call to get file size */
main(argc,argv)
int argc;
char **argv;
{
     struct registers reg;
     int path;

/* tell linker we need longs */
     pflinit();

/* low level open(file name is first command line param */
     path=open(*++argv,S_IREAD);

/* set up regs for call to OS-9 */
     reg.rg_a=path;
     reg.rg_b=SS_SIZE;

     if(_os9(I_GETSTT,&reg) == 0)
          printf("filesize = %lx\n", /* success */
          (long) (reg.rg_x << 16)+reg.rg_u);
     else printf("OS9 error #%d\n",reg.rg_b & 0xff); /*failed*/

     dumpregs(&reg);  /* take a look at the registers */
}

dumpregs(r)
register struct registers *r;
{
     printf("cc=%02x\n",r->rg_cc & 0xff);
     printf(" a=%02x\n",r->rg_a & 0xff);
     printf(" b=%02x\n",r->rg_b & 0xff);
     printf("dp=%02x\n",r->rg_dp & 0xff);
     printf(" x=%02x\n",r->rg_x);
     printf(" y=%02x\n",r->rg_u);
     printf(" u=%02x\n",r->rg_y);
}
```

Name

Open — open a file for read/write access

Synopsis

```
open(fname, mode);

char *fname;
int mode;
```

Assembler Equivalent

os9 I$OPEN

Description

This call opens an existing file for reading if "mode" is 1, writing if "mode" is 2, or reading and writing if "mode" is 3. NOTE that these values are OS-9 specific and not compatible with other systems. "Fname" should point to a string representing the pathname of the file.

Open returns an integer as "path number" which should be used by i/o system calls referring to the file.

The position where reads or writes start is at the beginning of the file.

Diagnostics

-1 is returned if the file does not exist, if the pathname cannot be searched, if too many files are already open, or if the file permissions deny the requested mode.

See Also

creat(), read(), write(), dup(), close()

Name

Os9fork — create a process

Synopsis

```
os9fork(modname, paramsize, paramptr, type, lang, datasize);

char *modname;
int paramsize;
char *paramptr;
int type;
int lang;
int datasize;
```

Assembler Equivalent

os9 F$FORK

Description

The action of F$FORK is desribed fully in the OS-9 System Programmer's Manual. Os9fork will create a process that will run concurrently with the calling process. When the forked process terminates, it will return to the calling process.

"Modname" should point to the name of the desired module. "Paramsize" is the length of the parameter string which should normally be terminated with a '\n', and "paramptr" points to the parameter string. "Type" is the module type as found in the header(normally 1: program), and "lang" should match the language nibble in the module header (C programs have 1 for 6809 machine code here). "Datasize"

may be zero, or it may contain the number of 256 byte pages to give to the new process as initial allocation of memory.

Diagnostics

-1 will be returned on error, or the ID number of the child process will be returned on success.

Name

Pause — halt and wait for interrupt

Synopsis

```
pause();
```

Assembler Equivalent

os9 I$SLEEP (with a value of 0)

Description

Pause may be used to halt a task until an interrupt is received from "kill".

Pause always returns -1.

See Also

kill(), signal(), OS-9 shell command "kill"

Name

Prerr — print error message

Synopsis

```
prerr(filnum, errcode);

int filnum;
int errcode;
```

Assembler Equivalent

os9 F$PERR

Description

PRERR prints an error message on the output path as specified by "filnum" which must be the path number of an open file. The message depends on "errcode" which will normally be a standard OS-9 error code.

Name

Read, Readln — read from a file

Synopsis

```
read(pn, buffer, count);

int pn;
char *buffer;
```

```
int count;

readln(pn, buffer, count);

int pn;
char *buffer;
int count;
```

Assembler Equivalent

os9 I$READ

os9 I$READLN

Description

The path number, "pn" is an integer which is one of the standard path numbers 0, 1, or 2, or the path number should have been returned by a successful call to "open", "creat", or "dup". "Buffer" is a pointer to space with at least "count" bytes of memory into which read will put the data from the file.

It is guaranteed that at most "count" bytes will be read, but often less will be, either because, for `readln`, the file represents a terminal and input stops at the end of a line, or for both, end-of-file has been reached.

Readln causes "line-editing" such as echoin to take place and returns once the first "\n" is encountered in the input or the number of bytes requested has been read. Readln is the preferred call for reading from the user's terminal.

Read does not cause any such editing. See the OS-9 manual for a fuller description of the actions of these calls.

Diagnostics

Read and readln return the number of bytes actually read (0 at end-of-file) or -1 for physical i/o errors, a bad path number, or a ridicolous "count".

NOTE that end-of-file is not considered an error, and no error indication is returned. Zero is returned on EOF.

See Also

open(), creat(), dup()

Name

Sbrk, Ibrk — request additional working memory

Synopsis

```
char *sbrk(increase);

int increase;

char *ibrk(increase);

int increase;
```

Description

Sbrk requests an allocation from free memory and returns a pointer to its base.

"Sbrk()" requests the system to allocate "new" memory from outside the initial allocation.

Users should read the Memory Management section of this manual for a fuller explanation of the arrangement.

Ibrk requests memory from inside the initial memory allocation.

Diagnostics

Sbrk and ibrk return -1 if the requested amount of contiguous memory is unavailable.

Name

Setpr — set process priority

Synopsis

```
setpr(pid, priority);

int pid;
int priority;
```

Assembler Equivalent

os9 F$SPRIOR

Description

SETPR sets the process identified by "pid" (process id) to have a priority of "priority". The lowest level is 0 and the highest is 255.

Diagnostics

The call will return -1 if the process does not have the same user id as the caller.

Name

Setime, Getime — set and get system time

Synopsis

#include <time.h>

```
setime(buffer);

struct sgtbuf *buffer;

getime(buffer);

struct sgtbuf *buffer;
```

Assembler Equivalent

os9 F$STIME

os9 F$GTIME

Description

GETIME returns system time in buffer. SETIME sets system time from buffer.

Name

Setuid — set user id

Synopsis

```
setuid(uid);

int uid;
```

Assembler Equivalent

os9 F$SUSER

Description

This call may be used to set the user id for the current task. Setuid only works if the caller is the super user (user id 0).

Diagnostics

Zero is returned from a successful call, and -1 is returned on error.

See Also

getuid()

Name

Setstat — set file status

Synopsis

```
#include <sgstat.h>

/* code 0 */

setstat(code, filenum, buffer);

int code;
int filenum;
char *buffer;
/* code 2 */

setstat(code, filenum, size);

int code;
int filenum;
long size;
```

Assembler Equivalent

os9 F$SETSTT

Description

For a detailed explanation of this call, see the OS-9 System Programmer's Manual.

"Filenum" must be the path number of a currently open file. The only values for code at this time are 0 and 2. When "code" is 0, "buffer" should be the address of a 32 byte structure which is written

to the option section of the path descriptor of the file. The header file contains definitions of various structures maintained by OS-9 for use by the programmer. When code is 2, "size" should be a long integer specifying the new file size.

Name

Signal — catch or ignore interrupts

Synopsis

#include <signal.h> typedef int (*sighandler_t)(int);

```
sighandler_t signal(interrupt, address);

int interrupt;
sighandler_t address;
```

Description

This call is a comprehensive method of catching or ignoring signals sent to the current process. Notice that "kill()" does the sending of signals, and "signal()" does the catching.

Normally, a signal sent to a process causes it to terminate with the status of the signal. If, in advance of the anticipated signal, this system call is used, the program has the choice of ignoring the signal or designating a function to be executed when it is received. Different functions may be designated for different signals.

The values for "address" have the following meanings:

0	reset to the default i.e. abort when received.
1	ignore; this will apply until reset to another value.
Otherwise	taken to be the address of a C function which is to be executed on receipt of the signal.

If the latter case is chosen, when the signal is received by the process the "address" is reset to 0, the default, before the function is executed. This means that if the next signal received should be caught then another call to "signal()" should be made immediately. This is normally the first action taken by the "interrupt" function. The function may access the signal number which caused its execution by looking at its argument. On completion of this function the program resumes at the point at which is was "interrupted" by the signal.

An example should help to clarify all this. Suppose a program needs to create a temporary file which should be deleted before exiting. The body of the program might contain fragments like this:

```
pn  =  creat("temp",3);          /*  create  a temporary file */
signal(2,intrupt);               /*   ensure   tidying   up   */
signal(3,intrupt);
write(pn,string,count);
close(pn);                       /* finished writing */
unlink("temp");                  /* delete it */
exit(0);                         /* normal exit */
```

The call to "signal()" will ensure that if a keyboard or quit signal is received then the function "intrupt()" will be executed and this might be written:

```
intrupt(sig)
{
```

```
close(pn);                  /* close it if open */
unlink("temp");             /* delete it */
exit(sig);                  /* received signal er exit status */
}
```

In this case, as the function will be exiting before another signal is received, it is unnecessary to call "signal()" again to reset its pointer. Note that either the function "intrupt()" should appear in the source code before the call to "signal()", or it should be pre-declared.

The signals used by OS-9 are defined in the header file as follows:

```
/* OS-9 signals */
#define        SIGKILL 0  /* system abort (cannot be caught or ignored)*/
#define        SIGWAKE 1    /* wake up */
#define        SIGQUIT 2    /* keyboard abort */
#define        SIGINT  3    /* keyboard interrupt */

/* special addresses */
#define        SIG_DFL 0    /* reset to default */
#define        SIG_IGN 1    /* ignore */
```

Please note that there is another method of trapping signals, namely "intercept()" (q.v.). However, since "signal()" and "intercept()" are mutually incompatible, calls to both of them must not appear in the same program. The link-loader will preven the creation of an executable program in which both are called by aborting with an "entry name clash" error for "_sigint".

See Also

intercept(), OS-9 shell command "kill", kill()

Name

Stacksize, Freemem — obtain stack reservation size

Synopsis

```
stacksize();


freemem();
```

Description

For a description of the meaning and use of this call, the user is referred to the Memory Management section of this manual.

If the stack check code is in effect, a call to stacksize will return the maximum number of bytes of stack used at the time of the call. This call can be used to determine the stack size required by a program.

Freemem() will return the number of bytes of the stack that has not been used.

See Also

ibrk(), sbrk(), Global variable "memend" and value "end".

Name

Strass — byte by byte copy

Synopsis

```
_strass(s1, s2, count);

char *s1;
char *s2;
int count;
```

Description

Until such time as the compiler can deal with structure assignment, this function is useful for copying one structure to another.

"Count" bytes are copied from memory location at "s2" to memory as "s1" regardless of the contents.

Name

Tsleep — put process to sleep

Synopsis

```
tsleep(ticks);

int ticks;
```

Assembler Equivalent

os9 F$SLEEP

Description

Tsleep deactivates the calling process for a specified number of system "ticks" or indefinitely if "ticks" is zero. A tick is system dependent but is usually 100ms.

For a fuller description of this call, see the OS-9 System Programmer's Manual.

Name

Unlink — remove directory entry

Synopsis

```
unlink(fname);

char *fname;
```

Assembler Equivalent

os9 I$DELETE

Description

Unlink deletes the directory entry whose name is pointed to by "fname". If the entry was the last link to the file, the file itself is deleted and the disc space occupied made available for re-use. If, however the file is open, in any active task, the deletion of the actual file is delayed until the file is closed.

Diagnostics

Zero is returned from a successful call, -1 if the file does not exist, if its directory is write-protected, or cannot be searched, if the file is a non-empty directory or a device.

See Also

OS-9 command "kill"

Name

Wait — wait for task termination

Synopsis

```
wait(status);

int *status;

wait(0);

0;
```

Assembler Equivalent

os9 F$WAIT

Description

Wait is used to halt the current task until a child task has terminated.

The call returns the task id of the terminating task and places the status of that task in the integer pointed to by "status" unless "status" is 0. A wait must be executed for each child task spawned.

The status will contain the argument of the "exit" or "_exit" call in the child task of the signal number if it was interrupted. A normally terminating C program with no call to "exit" or "_exit" has an implied call of "exit(0)".

Caveats

NOTE that the status is the OS-9 status code and is not compatible with codes on other systems.

Diagnostics

-1 is returned if there is no child to be waited for.

See Also

os9fork(), signal(), exit(),_exit()

Name

Write, Writeln — write to a file or device

Synopsis

```
write(pn, buffer, count);

int pn;
char *buffer;
int count;

writeln(pn, buffer, count);

int pn;
```

```
char *buffer;
int count;
```

Assembler Equivalent

os9 I$WRITE

os9 I$WRITLN

Description

"Pn" must be a value returned by "open", "creat" or "dup" or should be a 0(stdin), 1(stdout), or 2(stderr).

"Buffer should point to an area of memory from which "count" bytes are to be written. Write returns the actual number of bytes written, and if this is different from "count", an error has occurred.

Writes in multiples of 256 bytes to file offset boundaries of 256 bytes are the most efficient.

Write causes no "line-editing" to occur on output. Writeln causes line-editing and only writes up to the first "\n" in the buffer if this is found before "count" is exhausted. For a full description of the actions of these calls the reader is referred to the OS-9 documentation.

Diagnostics

-1 is returned if "pn" is a bad path number, of "count" is ridiculous or on physical i/o error.

See Also

creat(), open()

Chapter 4. C Standard Library

The Standard Library contains functions which fall into two classes: high-level I/O and convenience.

The high-level I/O functions provide facilities that are normally considered part of the definition of other languages; for example, the FORMAT "statement" of Fortran. In addition, automatic buffering of I/O channels improves the speed of file access because fewer system calls are necessary.

The high-level I/O functions should not be confused with the low-level system calls with similar names. Nor should "file pointers" be confused with "path numbers". The standard library functions maintain a structure for each file open that holds status information and a pointer into the files buffer, A user program uses a pointer to this structure as the "identity" of the file (which is provided by "fopen()"), and passes it to the various I/O functions. The I/O functions will make the low-level system calls when necessary.

Using a file pointer in a system call, or a path number in a Standard Library call, is a common mistake among beginners to C and, if made, will be sure to *crash your program.*

The convenience functions include facilities for copying, comparing, and concatenating strings, converting numbers to strings, and doing the extra work in accessing system information such as the time.

In the page which follow, the functions available are described in terms of what they do and the parameters they expect. The "USAGE" section shows the name of the function and the type returned (if not int). The declaration of arguments are shown as they would be written in the function definition to indicate the types expected by the function. If it is necessary to include a file before the function can be used, it is shown in the "USAGE" section by "#include <filename>".

Most of the header files that are required to be included, must reside in the "DEFS" directory on the default system drive. If the file is included in the source program using angle bracket delimiters instead of the usual double quotes, the compiler will append this path name to the file name. For example, "#include <stdio.h>" is equivalent to "#include </d0/defs/stdio.h>", if "/d0" is the path name of the default system drive.

Please note that if the type of the value returned by a function is not INT, you should make a pre-declaration in your program before calling it. For example, if you wish to use "atof()", you should pre-declare by having "double atof();" somewhere in your program before a call to it. Some functions which have associated header files in the DEFS directory that should be included, will be pre-declared for you in the header. An example of this is "ftell()" which is pre-declared in "stdio.h". If you are in any doubt, read the header file.

Name

Atof, Atoi, Atol — ASCII to number conversions

Synopsis

```
double atof(ptr);

char *ptr;

long atol(ptr);

char *ptr;

int atoi(ptr);

char *ptr;
```

Description

Conversions of the string pointed to by "ptr" to the relevant number type are carried out by these functions. They cease to convert a number when the first unrecognized character is encountered.

Each skips leading spaces and tab characters. Atof() recognizes an optional sign followed by a digit string that could possibly contain a decimal point, then an optional "e" or "E", and optional sign and a digit string. Atol() and atoi() recognize an optional sign and a digit string.

Caveats

Overflow causes unpredictable results. There are no error indications.

Name

Fclose, Fflush — flush or close a file

Synopsis

#include <stdio.h>

fclose(*fp*);

FILE **fp*;

fflush(*fp*);

FILE **fp*;

Description

Fflush causes a buffer associated with the file pointer "fp" to be cleared by writing out to the file; of course, only if the file was opened for write or update. It is not normally necessary to call fflush, but it can be useful when, for example, normal output is to "stdout", and it is wished to send something to "stderr" which is unbuffered. If fflush were not used and "stdout" referred to the terminal, the "stderr" message will appear before large chunks of the "stdout" message even though the latter was written first.

Fclose call fflush to clear out the buffer associated with "fp", closes the file, and frees the buffer for use by another fopen call.

The exit() system call and normal termination of a program causes fclose to be called for each open file.

See Also

System call close(), fopen(), setbuf().

Diagnostics

EOF is returned if "fp" does not refer to an output file or there is an error writing to the file.

Name

Feof, Ferror, Clearerr, Fileno — return status information of files

Synopsis

#include <stdio.h>

feof(*fp*);

```
FILE *fp;

ferror(fp);

FILE *fp;

clearerr(fp);

FILE *fp;

fileno(fp);

FILE *fp;
```

Description

Feof returns non-zero if the file associated with "fp" has reached its end. Zero is returned on error.

Ferror returns non-zero if an error condition occurs on access to the file "fp"; zero is returned otherwise. The error condition persists, preventing further access to the file by other Standard Library functions, until the file is closed, or it is cleared by clearerr.

Clearerr resets the error condition on the file "fp". This does NOT "fix" the file or prevent the error from occurring again; it merely allows Standard Library functions at least to try.

Caveats

These functions are actually macros that are defined in "<stdio.h>" so their names cannot be redeclared.

See Also

System call open(), fopen().

Name

Findstr, Findnstr — string search

Synopsis

```
findstr(pos, string, pattern);

int pos;
char *string;
char *pattern;

findnstr(pos, string, pattern, size);

int pos;
char *string;
char *pattern;
int size;
```

Description

These functions search the string pointed to by "string" for the first instance of the pattern pointed to by "pattern" starting at position "pos" (where the first position is 1 not 0). The returned value is the position of the first matched character of the pattern in the string or zero if a match is not found.

Findstr stops searching the string when a null byte is found in "string".

Findnstr only stops searching at position "pos" + "len" so it may continue past null bytes.

Caveats

The current implementation does not use the most efficient algorithm for pattern matching so that use on very long strings is likely to be somewhat slower than it might be.

See Also

index(), rindex()

Name

Fopen — open a file and return a file pointer

Synopsis

#include <stdio.h>

```
FILE *fopen(filename, action);

char *filename;
char *action;

FILE *freopen(filename, action, streak);

char *filename;
char *action;
FILE *streak;

FILE *fdopen(filedes, action);

FILE *filedes;
char *action;
```

Description

Fopen returns a pointer to a file structure (file pointer) if the file name in the string pointed to by "filename" can be validly opened with the action in the string pointed to by "action".

The valid actions are:

“r”	open for reading
“w”	create for writing
“a”	append(write) at end of file, or create for writing
“r+”	open for update
“w+”	create for update
“a+”	create or open for update at end of file
“d”	directory read

Any action may have an “x” after the initial letter which indicates to “fopen()” that it should look in the current execution directory if a full path is not given, and the x also specifies that the file should have execute permission.

E.g. f = fopen(“fred”,“wx”);

Opening for write will perform a “creat()”. If a file with the same name exists when the file is opened for write, it will be truncated to zero length. Append means open for write and position to the end

of the file. Writes to the file via “putc()” etc. will extend the file. Only if the file does not already exist will it be created.

NOTE that the type of a file structure is pre-defined in “stdio.h” as FILE, so that a user program may decale or define a file pointer by, for example, FILE *f;

Three file pointers are available and can be considered open the moment the program runs:

stdin	the standard input - equivalent to path number 0
stdout	the standard output - equivalent to path number 1
stderr	the standard error output - equivalent to path number 2

All files are automatically buffered except stderr, unless a file is made unbuffered by a call to setbuf() (q.v.).

Freopen is usually used to attach stdin, stdout, and stderr to specified files. Freopen substitutes the file passed to it instead of the open stream. The original stream is closed. NOTE that the original stream will be closed even if the open does not succeed.

Fdopen associates a stream with a file descriptor. The streams type(r,w,a) must be the same as the mode of the open file.

Caveats

The “action” passed as an argument to fopen must be a pointer to a string, *not* a character. For example

fp = fopen(“fred”,“r”); is correct but
fp = fopen(“fred”,'r'); is not.

Diagnostics

Fopen returns NULL (0) if the call was unsuccessful.

See Also

System call open(), fclose()

Name

Fread, Fwrite — read/write binary data

Synopsis

#include <stdio.h>

```
fread(ptr, size, number, fp);

char *ptr;
int size;
int number;
FILE *fp;

fwrite(ptr, size, number, fp);

char *ptr;
int size;
int number;
```

```
FILE *fp;
```

Description

Fread reads from the file pointed to by "fp". "Number" is the number of items of size "size" that are to be read starting at "ptr". The best way to pass the argument "size" to fread is by using "sizeof". Fread returns the number of items actually read.

Fwrite writes to the file pointed to by "fp". "Number" is the number of items of size "size" reading the from memory starting at "ptr".

Diagnostics

Both functions return 0 (NULL) at the end of file or error.

See Also

System calls read(), write(). Fopen(), getc(), putc(), printf().

Name

Fseek, Rewind, Ftell — position in a file or report current position

Synopsis

#include <stdio.h>

```
fseek(fp, offset, place);

FILE *fp;
long offset;
int place;

rewind(fp);

FILE *fp;

long ftell(fp);

FILE *fp;
```

Description

Fseek repositions the next character position of a file for either read or write. The new position is a "offset" bytes from the beginning of the file if "place" is 0, the current position is 1, or the end if 2. Fseek sorts out the special problems of buffering.

NOTE that using "lseek()" on a buffered file will produce unpredictable results.

Rewind is equivalent to "fseek(fp,0L,0)".

Ftell returns the current position, measured in bytes, from the beginning of the file pointed to by "fp".

Diagnostics

Fseek returns -1 if the call is invalid.

See Also

System call lseek().

Name

Getc, Getchar, Getw — return next character to be read from a file

Synopsis

#include <stdio.h>

```
int getc(fp);

FILE *fp;

int getchar();


int getw(fp);

FILE *fp;
```

Description

Getc returns the next character from the file pointed to by "fp".

Getchar is equivalent to "getc(stdin)".

Getw returns the next two bytes from the file as an integer.

Under OS-9 there is a choice of service requests to use when reading from a file. "Read()" will get characters up to a specified number in "raw" mode i.e. no editing will take place on the input stream and the characters will appear to the program exactly as in the file. "Readln()", on the other hand, will honor the various mappings of characters associated with a Serial Character device such as a terminal and in any case will return to the caller as soon as a carriage return is seen on the input.

In the vast majority of cases, it is preferable to use "readln()" for accessing Serial Character devices and "read()" for any other file input. "Getc()" uses this strategy and, as all file input using the Standard Library functions is routed through "getc()", so do all the other input functions. The choice is made when the first call to "getc()" is made after the file has been opened. The system is consulted for the status of the file and a flag bit is set in the file structure accordingly. The choice may be forced by the programmer by setting the relevant bit before a call to "getc()". The flag bits are defined in "<stdio.h>" and "_SCF" and "_RBF" and the method is as follows: assuming that the file pointer for the file, as returned by "fopen()" is f,

```
f->_flag |= _SCF;
```

will force the use of "readln()" on input and

```
f->_flag |= _RBF;
```

will force the use of "read()". This trick may be played on the standard streams "stdin", "stdout" and "stderr" without the need for calling "fopen()" but before any input is requested from the stream.

Diagnostics

EOF(-1) is returned for end of file or error.

See Also

Putc(), fread(), fopen(), gets(), ungetc()

Name

Gets, Fgets — input a string

Synopsis

#include <stdio.h>

```
char *gets(s);

char *s;

char *fgets(s, n, fp);

char *s;
int n;
FILE *fp;
```

Description

Fgets reads characters from the file "fp" and places them in the buffer pointed to by "s" up to a carriage return ('\n') but not more than "n" - 1 characters. A null character is appended to the end of the string.

Gets is similar to fgets, but gets is applied to "stdin" and no maximum is stipulated and '\n' is replaced by a null.

Both functions return their first arguments.

Caveats

The different treatment of the "\n" by these functions is retained here for portability reasons.

Diagnostics

Both functions return NULL on end-of-file or error.

See Also

puts(), getc(), scanf(), fread()

Name

Isalpha, Isupper, Islower, Isdigit, Isalnum, Isspace, Ispunct, Isprint, Iscntrl, Isascii — character classification

Synopsis

#include <ctype.h>

```
isalpha(c);

int c;
```

Description

These functions use table look-up to classify characters according to their ascii value. The header file defines them as macros which means that they are implemented as fast, inline code rather than subroutines.

Each results in non-zero for true or zero for false.

The correct value is guaranteed for all integer values in isascii, but the result is unpredictable in the others if the argument is outside the range -1 to 127.

The truth tested by each function is a follows:

isalpha	c is a letter
isdigit	c is a digit
isupper	c is an upper case letter
islower	c is a lower case letter
isalnum	c is a letter or a digit
isspace	c is a space, tab character, newline, carriage return or formfeed
iscntrl	c is a control character (0 to 32) or DEL (127)
ispunct	c is neither countrol nor alpha-numeric
isprint	c is printable (32 to 126)
isascii	c is in the range -1 to 127

Name

L3tol, Ltol3 — convert between long integers and 3-byte integers

Synopsis

```
l3tol(lp, cp, n);

long *lp;
char *cp;
int n;

ltol3(cp, lp, n);

char *cp;
long *lp;
int n;
```

Description

Certain system values, such as disc addresses, are maintained in three-byte form rather than four-byte; these functions enable arithmetic to be used on them.

L3tol converts a vector on "n" three-byte integers pointed to by "cp", into a vector of long integers starting at "lp".

Ltol3 does the opposite.

Name

Longjmp, Setjmp — jump to another function

Synopsis

#include <setjmp.h>

```
setjmp(env);
```

```
jmp_buf env;

longjmp(env, val);

jmp_buf env;
int val;
```

Description

These functions allow the return of program control directly to a higher level function. They are most useful when dealing with errors and interrupts encountered in a low level routine.

"Goto" in C has scope only in the function in which it is used; i.e. the label which is the object of a "goto" may only be in the same function. Control can only be transferred elsewhere by means of the function call, which, of course returns to the caller. In certain abnormal situations a programmer would prefer to be able to start some section of code again, but this would mean returning up a ladder of function calls with error indications all the way.

Setjmp is used to "mark" a point in the program where a subsequent longjmp can reach. It places in the buffer, defined in the header file, enough information for longjmp to restore the environment to that existing at the relevant call to setjmp.

Longjmp is called with the environment buffer as an argument and also, a value which can be used by the caller of setjmp as, perhaps, an error status.

To set the system up, a function will call setjmp to set up the buffer, and if the returned value is zero, the program will know that the call was the "first time through". If, however, the returned value is non-zero, it must be a longjmp returning from some deeper level of the program.

NOTE that the function calling setjmp must *not have returned* at the time of calling longjmp, and the environment buffer must be declared *globally*.

Name

Malloc, Free, Calloc — memory allocation

Synopsis

```
char *malloc(size);

unsigned size;

free(ptr);

char *ptr;

char *calloc(nel, elsize);

unsignednel;
unsignedelsize;
```

Description

Malloc returns a pointer to a block of at least "size" free bytes.

Free requires a pointer to a block that has been allocated by malloc; it frees the space to be allocated again.

Calloc allocates space for an array. Nel is the number of elements in the arrary, and elsize is the size of each element. Calloc initializes the space to zero.

Diagnostics

Malloc, free, and calloc return NULL(0) if no free memory can be found or if there was an error.

Name

Mktemp — create unique temporary file name

Synopsis

```
char *mktemp(name);

char *name;
```

Description

Mktemp may be used to ensure that the name of a temporary file is unique in the system and does not clash with any other file name.

"Name" must point to a string whose last five characters are "X"; the Xs will be replaced with the ascii representation of the task id.

For example, if "name" points to "foo.XXXXX", and the task id is 351, the returned value points at the same place, but it now holds "foo.351".

See Also

System call getpid()

Name

Printf, Fprintf, Sprintf — formatted output

Synopsis

#include <stdio.h>

```
printf(control, );

char *control;
...;

fprintf(fp, control, );

FILE *fp;
char *control;
...;

sprintf(string, control, );

char *string;
char *control;
...;
```

Description

Thse three functions are used to place numbers and strings on the output in formatted, human readable form.

Fprintf places its output on the file "fp", printf on the standard output, and sprintf in the buffer pointed to by "string". NOTE that it is the user's responsibility to ensure that this buffer is large enough.

The "control" string determines the format, type, and number of the following arguments expected by the function. If the control does not match the arguments correctly, the results are unpredictable.

The control may contain characters to be copied directly to the output and/or format specifications. Each format specification causes the function to take the next successive argument for output.

A format specification consists of a "%" character followed by (in this order):

- An optional minus sign ("-") that means left justification in the field.
- An optional string of digits indication the field width required. The field will be at least this wide and may be wider if the conversion requires it. The field will be padded on the left unless the above minus sign is present, in which case it will be padded on the right. The padding character is, by default, a space, but if the digit string starts with a zero ("0"), it will be "0".
- An optional dot (".") and a digit string, the precision, which for floating point arguments indicates the number of digits to follow the decimal point on conversion, and for strings, the maximum number of characters from the string argument are to be printed.
- An optional character "l" indicates that the following "d","x", or "o" is the specification of a long integer argument. NOTE that in order for the printing of long integers to take place, the source code must have in it somewhere the statement pflinit(), which causes routines to be linked from the library.
- A conversion character which shows the type of the argument and the desired conversion. The recognized conversion characters are:

d,o,x,X	The argument is an integer and the conversion is to decimal, octal, or hexadecimal, respectively. "X" prints hex and alpha in upper case.
u	The argument is an integer and the conversion is to an unsigned decimal in the range 0 to 65535.
f	The argument is a double, and the form of the conversion is "[-]nnn.nnn". Where the digits after the decimal point are specified as above. If not specified, the precision defaults to six digits. If the precision is 0, no decimal point or following digits are printed.
e,E	The argument is a double and the form of the conversion is "[-]n.nnne(+or-)nn"; one digit before the decimal point, and the precision controls the number following. "E" prints the "e" in upper case.
g,G	The argument is a double, and either the "f" format or the "e" format is chosen, whichever is the shortest. If the "G" format is used, the "e" is printed in upper case.

NOTE in each of the above double conversions, the last digit is rounded.

ALSO NOTE that in order for the printing of floats or doubles to take place, the source program *must* have the statement pffinit() somewhere.

c	The argument as a character.
s	The argument is a pointer to a string. Characters from the string are printed up to a null character, or untill the number of characters indicated by the precision have been printed. If the precision is 0 or missing, the characters are not counted.
%	No argument corresponding; "%" is printed.

See Also

Kernighan & Ritchie pages 145-147. putc(), scanf()

Name

Putc, Putchar, Putw — put character or word in a file

Synopsis

```
#include <stdio.h>

putc(ch, fp);

char ch;
FILE *fp;

putchar(ch);

char ch;

putw(n, fp);

int n;
FILE *fp;
```

Description

Putc add the character "ch" to the file "fp" at the current writing position and advances the position pointer.

Putchar is implemented as a macro (defined in the header file) and is equivalent to "putc(ch,stdout)".

Putw adds the (two byte) machine word "n" to the file "fp" in the manner of putc.

Output via putc is normally buffered except; (a) when the buffering is disabled by "setbuf()", and (b) the standard error output is always unbuffered.

Diagnostics

Putc and putchar return the character argument from a successful call, and EOF on end-of-file or error.

See Also

fopen(), fclose(), fflush(), getc(), puts(), printf(), fread()

Name

Puts, Fputs — put a string on a file

Synopsis

```
#include <stdio.h>

puts(s);

char *s;

fputs(s, fp);

char *s;
FILE *fp;
```

Description

Fputs copies the (null-terminated) string pointed to by "s" onto the file "fp".

Puts copies the string "s" onto the standard output and appends "\n".

The terminating null is not copied by either function.

Caveats

The inconsistency of the new-line being appended by puts and not by fputs is dictated by history and the desire for compatibility.

Name

Qsort — quick sort

Synopsis

```
qsort(base, n, size, (* compfunc));

char *base;
int n;
int size;
int (* compfunc) (void *, void *);
```

Description

Qsort implements the quick-sort algoritm for sortig an arbitrary array of items.

"Base" is the address of the array of "n" items of size "size". "Compfunc" is a pointer to a comparison routine supplied by the user. It will be called by qsort with two pointers to items in the array for comparison and should return an integer which is less than, equal to, or greater than 0 where, respectively, the first item is less than, equal to, or greater than the second.

Name

Scanf, Fscanf, Sscanf — input string interpretation

Synopsis

#include <stdio.h>

```
fscanf(fp, control, pointer...);

FILE *fp;
char *control;
char *pointer...;

scanf(control, pointer...);

char *control;
char *pointer...;

sscanf(string, control, pointer...);

char *string;
char *control;
char *pointer...;
```

Description

These functions perform the complement to "printf()" etc.

Fscanf performs conversions from the file "fp", scanf from the standard input, and sscanf from the string pointed to by "string".

Each function expects a control string containing conversion specifications, and zero or more pointers to objects into which the converted values are stored.

The control string may contain three types of fields:

a. Space, tab characters, or "\n" which match any of the three in the input.
b. Characters not among the above and not "%" which must match characters in the input.
c. A "%" followed by an optional "*" indicates suppression of assignment, an optional field width maximum and a conversion character indicating the type expected.

A conversion character controls the conversion to be applied to the next field and indicates the type of the corresponding pointer argument. A field consists of consecutive non-space characters and ends at either a character inappropiate for the conversion or when a specified field is exhausted. When one field is finished, white-space characters are passed over until the next field is found.

d	A decimal string is to be converted to an integer.
o	An octal string; the coresponding argument should point to an integer.
x	A hexadecimal string for conversion to an integer.
s	A string of non-space characters is expected and will be copied to the buffer pointed to by the corresponding argument and a null ("\0") appended. The user must ensure that the buffer is large enough. The input string is considered terminated by a space, tab of ("\n").
c	A character is expected and is copied into the byte pointed to by the argument. The white-space skipping is suppressed for this conversion. If a field width is given, the argument is assumed to point to a character array and the number of characters indicated is copied to it. NOTE to ensure that the next non-white-space character is read use "%1s" and that TWO bytes are pointed to by the argument.
e,f	A floating point representation is expected on the input and the argument must be a pointer to a float. Any of the usual ways of writing floating point numbers are recognized.
[	This denotes the start of a set of match characters; the inclusion or exclusion of which delimits the input field. The white-space skipping is suppressed. The corresponding argument should be a pointer to a character array. If the first character in the match string is not "^", characters are copied from the input as long as they can be found in the match string. If the first character is the "^", copying continues while characters cannot be found in the match string. The match string is delimited by a "]".
D,O,X	Similar to d,o,x above, but the corresponding argument is considered to point to a long integer.
E,F	Similar to e,f above, but the corresponding should point to a double.
%	A match for "%" is sought; no conversion takes place.

Each of the functions returns a count of the number of fields successfully matched and assigned.

Caveats

The returned count of matches/assigments does not include character matches and assigments suppressed by "*". The arguments must ALL be pointers. It is a common error to call scanf with the value of an item rather than a pointer to it.

Diagnostics

These functions return EOF on end of input or error and a count which is shorter than expected for unexpected or unmatched items.

See Also

Atoi(), atof(), getc(), printf() Kernighan and Ritchie pp 147-150

Name

Setbuf — fix file buffer

Synopsis

#include <stdio.h>

```
setbuf(fp, buffer);

FILE *fp;
char *buffer;
```

Description

When the first character is written to or read from a file after it has been opened by "fopen()", a buffer is obtained from the system if required and assigned to it. Setbuf may be used to forestall this by assigning a user buffer to the file.

Setbuf must be used after the file has been opened and before any I/O has taken place.

The buffer must be of sufficient size and a value for a manifest constant, BUFSIZ, is defined in the header file for use in declarations.

If the "buffer" argument is NULL (0), the file becomes unbuffered and characters are read or written singly.

NOTE that the standard error output is unbuffered and the standard output is buffered.

See Also

fopen(), getc(), putc()

Name

Sleep — stop execution for a time

Synopsis

```
sleep(seconds);

int seconds;
```

Description

The current task is stopped for the specified time.

If "seconds" is zero, the task will sleep for one tick.

Name

Strcat, Strncat, Strcmp, Strncmp, Strcpy, Strhcpy, Strncpy, Strlen, Index, Rindex — string functions

Synopsis

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

char *s1;
char *s2;

char *strncat(s1, s2, n);

char *s1;
char *s2;
int n;

int strcmp(s1, s2);

char *s1;
char *s2;

char *strhcpy(s1, s2);

char *s1;
char *s2;

int strncmp(s1, s2, n);

char *s1;
char *s2;
int n;

char *strcpy(s1, s2);

char *s1;
char *s2;

char *strncpy(s1, s2, n);

char *s1;
char *s2;
int n;

int strlen(s);

char *s;

char *index(s, ch);

char *s;
char ch;

char *rindex(s, ch);

char *s;
charch;
```

Description

All strings passed to these functions are assumed null-terminated.

Strcat appends a copy of the string pointed to by "s2" to the end of the string pointed to by "s1". Strncat copies at most "n" characters. Both return the first argument.

Strcmp compares strings "s1" and "s2" for lexicographic order and returns an integer less than, equal to or greater than 0 where, respectively, "s1" is less than, equal to or greater than "s2". Strncmp compares at most "n" characters.

Strcpy copies characters from "s2" to the space pointed to by "s1" up to and including the null byte. Strncpy copies exactly "n" characters. If the string "s2" is too short, the "s1" will be padded with null bytes to make up the difference. If "s2" is too long, "s1" may not be null-terminated. Both functions return the first argument.

Strhcpy copies string with sign bit terminator.

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

Index returns a pointer to the first occurrence of "ch" in "s" or NULL if not found.

Rindex returns a pointer to the last occurrence of "ch" in "s" or NULL if not found.

Caveats

Strcat and strcpy have no means of checking that the space provided is large enough. It is the user's responsibility to ensure that string space does not overflow.

See Also

findstr().

Name

System — shell command interpreter

Synopsis

```
system(string);

char *string;
```

Description

System passes its argument to "shell" which executes it as a command line. The task is suspended until the shell command is completed and system returns the shell's exit status. The maximum length of string is 80 characters. If a longer string is needed, use os9fork.

See Also

System calls os9fork(), wait().

Name

Toupper, Tolower — character translation

Synopsis

#include <ctype.h>

```
toupper(c);

int c;

tolower(c);

int c;
```

```
_toupper(c);

int c;

_tolower(c);

int c;
```

Description

The functions toupper and tolower have as their domain the integers in the range -1 to 255. Toupper converts lower-case to upper-case, and tolower converts upper-case to lower-case. All other arguments are returned unchanged.

The macros _toupper and _tolower do the same things as the corresponding functions, but they have restricted domains and they are faster. The argument to _toupper must be lower-case, and the argument to _tolower must be upper-case. Arguments that are outside each macros domain, such as passing a lower-case to _tolower, yield garbage results.

Name

Ungetc — put character back on input

Synopsis

#include <stdio.h>

```
ungetc(ch, fp);

char ch;
FILE *fp;
```

Description

This function alters the state of the input file buffer such that the next call of "getc()" returns "ch".

Only one character may be pushed back, and at least on character must have been read from the file before a call to ungetc.

“Fseek()” erases any puchback.

Diagnostics

Ungetc returns its character argument unless no puchback could occur, in which case EOF is returned.

See Also

getc(), fseek()

Appendix A. C Reference Manual

A.1. Introduction

This manual describes the C language on the DEC PDP-11[1], the DEC VAX-11, and the 6809[2]. Where differences exist, it concentrates on the VAX, but tries to point out implementation-dependent details. With few exceptions, these dependencies follow directly from the underlying properties of the hardware; the various compilers are generally quite compatible.

A.2. Lexical Conventions

There are six classes of tokens - identifiers, keywords, constants, strings, operators, and other separators. Blanks, tabs, newlines, and comments (collectively, “white space”) as described below are ignored except as they serve to separate tokens. Some white space is required to separate otherwise adjacent identifiers, keywords, and constants.

If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest string of characters which could possibly constitute a token.

A.2.1. Comments

The characters `/*` introduce a comment which terminates with the characters `*/`. Comments do not nest.

A.2.2. Identifiers (Names)

An identifier is a sequence of letters and digits. The first character must be a letter. The underscore (_) counts as a letter. Uppercase and lowercase letters are different. Although there is no limit on the length of a name, only initial characters are significant: at least eight characters of a non-external name, and perhaps fewer for external names. Moreover, some implementations may collapse case distinctions for external names. The external name sizes include:

PDP-11	7 characters, 2 cases
VAX-11	>100 characters, 2 cases
Motorola 6809	8 characters, 2 cases

A.2.3. Keywords

The following identifiers are reserved for use as keywords and may not be used otherwise:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry

[1] DEC PDP-11, and DEC VAX-11 are trademarks of Digital Equipment Corporation.
[2] 6809 is a trademark of Motorola.

unsigned	continue	register
auto	if	

Some implementations also reserve the words `direct`, `fortran` and `asm`

A.2.4. Constants

There are several kinds of constants. Each has a type; an introduction to types is given in Section A.4, “What's in a name?”. Hardware characteristics that affect sizes are summarized in “Hardware Characteristics” under Section A.2, “Lexical Conventions”.

A.2.4.1. Integer Constants

An integer constant consisting of a sequence of digits is taken to be octal if it begins with `0` (digit zero). An octal constant consists of the digits `0` through `7` only. A sequence of digits preceded by `0x` or `0X` (digit zero) is taken to be a hexadecimal integer. The hexadecimal digits include `a` or `A` through `f` or `F` with values 10 through 15. Otherwise, the integer constant is taken to be decimal. A decimal constant whose value exceeds the largest signed machine integer is taken to be `long`; an octal or hex constant which exceeds the largest unsigned machine integer is likewise taken to be `long`.

A.2.4.2. Explicit Long Constants

A decimal, octal, or hexadecimal integer constant immediately followed by `l` (letter ell) or `L` is a long constant. As discussed below, on some machines integer and long values may be considered identical.

A.2.4.3. Character Constants

A character constant is a character enclosed in single quotes, as in '`x`'. The value of a character constant is the numerical value of the character in the machine's character set.

Certain nongraphic characters, the single quote (`'`) and the backslash (`\`), may be represented according to the following table of escape sequences:

newline	NL (LF)	\n
horizontal tab	HT	\t
vertical tab	VT	\v
backspace	BS	\b
carriage return	CR	\r
form feed	FF	\f
backslash	\	\\
single quote	'	\'
bit pattern	*ddd*	*ddd*

The escape *ddd* consists of the backslash followed by 1, 2, or 3 octal digits which are taken to specify the value of the desired character. A special case of this construction is `\0` (not followed by a digit), which indicates the character `NUL`. If the character following a backslash is not one of those specified, the behavior is undefined. A new-line character is illegal in a character constant. The type of a character constant is `int`.

A.2.4.4. Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an `e` or `E`, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing. Either the decimal point or the `e` and the exponent (not both) may be missing. Every floating constant is taken to be double-precision.

A.2.5. Strings

A string is a sequence of characters surrounded by double quotes, as in `"..."`. A string has type "array of `char`" and storage class `static` (see Section A.4, "What's in a name?") and is initialized with the given characters. The compiler places a null byte (`\0`) at the end of each string so that programs which scan the string can find its end. In a string, the double quote character (`"`) must be preceded by a \; in addition, the same escapes as described for character constants may be used.

A \ and the immediately following newline are ignored. All strings, even when written identically, are distinct.

A.2.6. Hardware Characteristics

The following figure summarize certain hardware properties that vary from machine to machine.

Table A.1. DEC PDP-11 Hardware Characteristics

	DEC PDP-11 (ASCII)	**DEC VAX-11 (ASCII)**	**6809 (ASCII)**
char	8 bits	8 bits	8 bits
int	16	32	16
short	16	16	16
long	32	32	32
float	32	32	32
double	64	64	64
float range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$
double range	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$	$\pm 10^{\pm 38}$

A.3. Syntax Notation

Syntactic categories are indicated by *italic* type and literal words and characters in `bold` type. Alternative categories are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt," so that

{ $expression_{opt}$ }

indicates an optional expression enclosed in braces. The syntax is summarized in Section A.18, "Syntax Summary".

A.4. What's in a name?

C bases the interpretation of an identifier upon two attributes of the identifier: its *storage class* and its *type*. The storage class determines the location and lifetime of the storage associated with an identifier; the type determines the meaning of the values found in the identifier's storage.

There are four declarable storage classes: automatic, static, external, and register. Automatic variables are local to each invocation of a block (see Section A.9.2, "Compound Statement or Block") and are discarded upon exit from the block. Static variables are local to a block but retain their values upon reentry to a block even after control has left the block. External variables exist and retain their values throughout the execution of the entire program and may be used for communication between functions, even separately compiled functions. Register variables are (if possible) stored in the fast registers of the machine; like automatic variables, they are local to each block and disappear on exit from the block.

C supports several fundamental types of objects:

Objects declared as characters (`char`) are large enough to store any member of the implementation's character set, and if a genuine character from that character set is stored in a `char` variable, its value is equivalent to the integer code for that character. Other quantities may be stored into character variables, but the implementation is machine dependent.

Up to three sizes of integer, declared `short int`, `int`, and `long int`, are available. Longer integers provide no less storage than shorter ones, but the implementation may make either short integers or long integers, or both, equivalent to plain integers. “Plain” integers have the natural size suggested by the host machine architecture. The other sizes are provided to meet special needs.

Unsigned integers, declared `unsigned`, obey the laws of arithmetic modulo 2^n where *n* is the number of bits in the representation. (On the PDP-11, unsigned long quantities are not supported.)

Single-precision floating point (`float`) and double precision floating point (`double`) may be synonymous in some implementations.

Because objects of the foregoing types can usefully be interpreted as numbers, they will be referred to as *arithmetic* types. Types `char` and `int` of all sizes will collectively be called *integral* types. `float` and `double` types will collectively be called *floating* types.

Besides the fundamental arithmetic types, there is a conceptually infinite class of derived types constructed from the fundamental types in the following ways:

- *arrays* of objects of most types
- *functions* which return objects of a given type
- *pointers* to objects of a given type
- *structures* containing a sequence of objects of various types
- *unions* capable of containing any one of several objects of various types.

In general these methods of constructing objects can be applied recursively.

A.5. Objects and lvalues

An *object* is a manipulatable region of storage. An *lvalue* is an expression referring to an object. An obvious example of an lvalue expression is an identifier. There are operators which yield lvalues: for example, if `E` is an expression of pointer type, then `*E` is an lvalue expression referring to the object to which `E` points. The name “lvalue” comes from the assignment expression `E1 = E2` in which the left operand `E1` must be an lvalue expression. The discussion of each operator below indicates whether it expects lvalue operands and whether it yields an lvalue.

A.6. Conversions

A number of operators may, depending on their operands, cause conversion of the value of an operand from one type to another. This part explains the result to be expected from such conversions. The conversions demanded by most ordinary operators are summarized under “Arithmetic Conversions.” The summary will be supplemented as required by the discussion of each operator.

A.6.1. Characters and Integers

A character or a short integer may be used wherever an integer may be used. In all cases the value is converted to an integer. Conversion of a shorter integer to a longer preserves sign. Whether or not sign-extension occurs for characters is machine dependent, but it is guaranteed that a member of the

standard character set is non-negative. Of the machines treated here, only the PDP-11 and VAX-11 sign-extend. On these machines, `char` variables range in value from -128 to 127. The more explicit type `unsigned char` forces the values to range from 0 to 255.

On machines that treat characters as signed, the characters of the ASCII set are all non-negative. However, a character constant specified with an octal escape suffers sign extension and may appear negative; for example, `'\377'` has the value `-1`.

When a longer integer is converted to a shorter integer or to a `char`, it is truncated on the left. Excess bits are simply discarded.

A.6.2. Float and Double

All floating arithmetic in C is carried out in double precision. Whenever a `float` appears in an expression it is lengthened to `double` by zero padding its fraction. When a `double` must be converted to `float`, for example by an assignment, the `double` is rounded before truncation to `float` length. This result is undefined if it cannot be represented as a float. On the VAX, the compiler can be directed to use single precision for expressions containing only float and integer operands.

A.6.3. Floating and Integral

Conversions of floating values to integral type are rather machine dependent. In particular, the direction of truncation of negative numbers varies. The result is undefined if it will not fit in the space provided.

Conversions of integral values to floating type are well behaved. Some loss of accuracy occurs if the destination lacks sufficient bits.

A.6.4. Pointers and Integers

An expression of integral type may be added to or subtracted from a pointer; in such a case, the first is converted as specified in the discussion of the addition operator. Two pointers to objects of the same type may be subtracted; in this case, the result is converted to an integer as specified in the discussion of the subtraction operator.

A.6.5. Unsigned

Whenever an unsigned integer and a plain integer are combined, the plain integer is converted to unsigned and the result is unsigned. The value is the least unsigned integer congruent to the signed integer (modulo $2^{wordsize}$). In a 2's complement representation, this conversion is conceptual; and there is no actual change in the bit pattern.

When an unsigned `short` integer is converted to `long`, the value of the result is the same numerically as that of the unsigned integer. Thus the conversion amounts to padding with zeros on the left.

A.6.6. Arithmetic Conversions

A great many operators cause conversions and yield result types in a similar way. This pattern will be called the “usual arithmetic conversions.”

a. First, any operands of type `char` or `short` are converted to `int`, and any operands of type `unsigned char` or `unsigned short` are converted to `unsigned int`.

b. Then, if either operand is `double`, the other is converted to `double` and that is the type of the result.

c. Otherwise, if either operand is `unsigned long`, the other is converted to `unsigned long` and that is the type of the result.

d. Otherwise, if either operand is `long`, the other is converted to `long` and that is the type of the result.

e. Otherwise, if one operand is `long`, and the other is `unsigned int`, they are both converted to `unsigned long` and that is the type of the result.

f. Otherwise, if either operand is `unsigned`, the other is converted to `unsigned` and that is the type of the result.

g. Otherwise, both operands must be `int`, and that is the type of the result.

A.7. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section, highest precedence first. Thus, for example, the expressions referred to as the operands of `+` (see Section A.7.4, “Additive Operators”) are those expressions defined under Section A.7.1, “Primary Expressions”, Section A.7.2, “Unary Operators”, and Section A.7.3, “Multiplicative Operators”. Within each subpart, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein. The precedence and associativity of all the expression operators are summarized in the grammar of Section A.18, “Syntax Summary”.

Otherwise, the order of evaluation of expressions is undefined. In particular, the compiler considers itself free to compute subexpressions in the order it believes most efficient even if the subexpressions involve side effects. The order in which subexpression evaluation takes place is unspecified. Expressions involving a commutative and associative operator (`*`, `+`, `&`, `|`, `^`) may be rearranged arbitrarily even in the presence of parentheses; to force a particular order of evaluation, an explicit temporary must be used.

The handling of overflow and divide check in expression evaluation is undefined. Most existing implementations of C ignore integer overflows; treatment of division by 0 and all floating-point exceptions varies between machines and is usually adjustable by a library function.

A.7.1. Primary Expressions

Primary expressions involving `.`, `->`, subscripting, and function calls group left to right.

primary-expression:
identifier
constant
string
(expression)
primary-expression [expression]
primary-expression (expression-list$_{opt}$ *)*
primary-expression . identifier
primary-expression -> identifier

expression-list:
expression
expression-list , expression

An identifier is a primary expression provided it has been suitably declared as discussed below. Its type is specified by its declaration. If the type of the identifier is “array of ...”, then the value of the identifier expression is a pointer to the first object in the array; and the type of the expression is “pointer to ...”. Moreover, an array identifier is not an lvalue expression. Likewise, an identifier which is declared “function returning ...”, when used except in the function-name position of a call, is converted to “pointer to function returning ...”.

A constant is a primary expression. Its type may be `int`, `long`, or `double` depending on its form. Character constants have type `int` and floating constants have type `double`.

A string is a primary expression. Its type is originally "array of `char`", but following the same rule given above for identifiers, this is modified to "pointer to `char`" and the result is a pointer to the first character in the string. (There is an exception in certain initializers; see Section A.8.6, "Initialization".)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

A primary expression followed by an expression in square brackets is a primary expression. The intuitive meaning is that of a subscript. Usually, the primary expression has type "pointer to ...", the subscript expression is `int`, and the type of the result is "...". The expression `E1[E2]` is identical (by definition) to `*((E1)+E2))`. All the clues needed to understand this notation are contained in this subpart together with the discussions in Section A.7.2, "Unary Operators" and Section A.7.4, "Additive Operators" on identifiers, `*` and `+` respectively. The implications are summarized under "Arrays, Pointers, and Subscripting" under Section A.14, "Types Revisited".

A function call is a primary expression followed by parentheses containing a possibly empty, comma-separated list of expressions which constitute the actual arguments to the function. The primary expression must be of type "function returning ...," and the result of the function call is of type "...". As indicated below, a hitherto unseen identifier followed immediately by a left parenthesis is contextually declared to represent a function returning an integer; thus in the most common case, integer-valued functions need not be declared.

Any actual arguments of type `float` are converted to `double` before the call. Any of type `char` or `short` are converted to `int`. Array names are converted to pointers. No other conversions are performed automatically; in particular, the compiler does not compare the types of actual arguments with those of formal arguments. If conversion is needed, use a cast; see Section A.7.2, "Unary Operators" and Section A.8.7, "Type Names".

In preparing for the call to a function, a copy is made of each actual parameter. Thus, all argument passing in C is strictly by value. A function may change the values of its formal parameters, but these changes cannot affect the values of the actual parameters. It is possible to pass a pointer on the understanding that the function may change the value of the object to which the pointer points. An array name is a pointer expression. The order of evaluation of arguments is undefined by the language; take note that the various compilers differ. Recursive calls to any function are permitted.

A primary expression followed by a dot followed by an identifier is an expression. The first expression must be a structure or a union, and the identifier must name a member of the structure or union. The value is the named member of the structure or union, and it is an lvalue if the first expression is an lvalue.

A primary expression followed by an arrow (built from `-` and `>`) followed by an identifier is an expression. The first expression must be a pointer to a structure or a union and the identifier must name a member of that structure or union. The result is an lvalue referring to the named member of the structure or union to which the pointer expression points. Thus the expression `E1->MOS` is the same as `(*E1).MOS`. Structures and unions are discussed in Section A.8.5, "Structure and Union Declarations" under Section A.8, "Declarations".

A.7.2. Unary Operators

Expressions with unary operators group right to left.

unary-expression:
** expression*
& lvalue
- expression

! expression
~ expression
++ lvalue
--lvalue
lvalue ++
lvalue --
(type-name) expression
sizeof *expression*
sizeof *(type-name)*

The unary * operator means *indirection* ; the expression must be a pointer, and the result is an lvalue referring to the object to which the expression points. If the type of the expression is "pointer to ...," the type of the result is "...".

The result of the unary & operator is a pointer to the object referred to by the lvalue. If the type of the lvalue is "...", the type of the result is "pointer to ...".

The result of the unary - operator is the negative of its operand. The usual arithmetic conversions are performed. The negative of an unsigned quantity is computed by subtracting its value from 2^n where *n* is the number of bits in the corresponding signed type. There is no unary + operator.

The result of the logical negation operator ! is one if the value of its operand is zero, zero if the value of its operand is nonzero. The type of the result is `int`. It is applicable to any arithmetic type or to pointers.

The ~ operator yields the one's complement of its operand. The usual arithmetic conversions are performed. The type of the operand must be integral.

The object referred to by the lvalue operand of prefix ++ is incremented. The value is the new value of the operand but is not an lvalue. The expression `++x` is equivalent to `x=x+1`. See the discussions Section A.7.4, "Additive Operators" and Section A.7.14, "Assignment Operators" for information on conversions.

The lvalue operand of prefix -- is decremented analogously to the prefix ++ operator.

When postfix ++ is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is incremented in the same manner as for the prefix ++ operator. The type of the result is the same as the type of the lvalue expression.

When postfix -- is applied to an lvalue, the result is the value of the object referred to by the lvalue. After the result is noted, the object is decremented in the manner as for the prefix -- operator. The type of the result is the same as the type of the lvalue expression.

An expression preceded by the parenthesized name of a data type causes conversion of the value of the expression to the named type. This construction is called a *cast*. Type names are described in Section A.8.7, "Type Names".

The `sizeof` operator yields the size in bytes of its operand. (A *byte* is undefined by the language except in terms of the value of `sizeof`. However, in all existing implementations, a byte is the space required to hold a `char`.) When applied to an array, the result is the total number of bytes in the array. The size is determined from the declarations of the objects in the expression. This expression is semantically an `unsigned` constant and may be used anywhere a constant is required. Its major use is in communication with routines like storage allocators and I/O systems.

The `sizeof` operator may also be applied to a parenthesized type name. In that case it yields the size in bytes of an object of the indicated type.

The construction `sizeof(`*type*`)` is taken to be a unit, so the expression `sizeof(`*type*`)-2` is the same as `(sizeof(`*type*`))-2`.

A.7.3. Multiplicative Operators

The multiplicative operators `*`, `/`, and `%` group left to right. The usual arithmetic conversions are performed.

multiplicative expression:
*expression * expression*
expression / expression
expression % expression

The binary `*` operator indicates multiplication. The `*` operator is associative, and expressions with several multiplications at the same level may be rearranged by the compiler. The binary `/` operator indicates division.

The binary `%` operator yields the remainder from the division of the first expression by the second. The operands must be integral.

When positive integers are divided, truncation is toward 0; but the form of truncation is machine-dependent if either operand is negative. On all machines covered by this manual, the remainder has the same sign as the dividend. It is always true that `(a/b)*b + a%b` is equal to `a` (if `b` is not 0).

A.7.4. Additive Operators

The additive operators `+` and `-` group left to right. The usual arithmetic conversions are performed. There are some additional type possibilities for each operator.

additive-expression:
expression + expression
expression - expression

The result of the `+` operator is the sum of the operands. A pointer to an object in an array and a value of any integral type may be added. The latter is in all cases converted to an address offset by multiplying it by the length of the object to which the pointer points. The result is a pointer of the same type as the original pointer which points to another object in the same array, appropriately offset from the original object. Thus if `P` is a pointer to an object in an array, the expression `P+1` is a pointer to the next object in the array. No further type combinations are allowed for pointers.

The `+` operator is associative, and expressions with several additions at the same level may be rearranged by the compiler.

The result of the `-` operator is the difference of the operands. The usual arithmetic conversions are performed. Additionally, a value of any integral type may be subtracted from a pointer, and then the same conversions for addition apply.

If two pointers to objects of the same type are subtracted, the result is converted (by division by the length of the object) to an `int` representing the number of objects separating the pointed-to objects. This conversion will in general give unexpected results unless the pointers point to objects in the same array, since pointers, even to objects of the same type, do not necessarily differ by a multiple of the object length.

A.7.5. Shift Operators

The shift operators `<<` and `>>` group left to right. Both perform the usual arithmetic conversions on their operands, each of which must be integral. Then the right operand is converted to `int`; the type of the result is that of the left operand. The result is undefined if the right operand is negative or greater than or equal to the length of the object in bits. On the VAX a negative right operand is interpreted as reversing the direction of the shift.

shift-expression:
expression << *expression*
expression >> *expression*

The value of `E1<<E2` is `E1` (interpreted as a bit pattern) left-shifted `E2` bits. Vacated bits are 0 filled. The value of `E1>>E2` is `E1` right-shifted `E2` bit positions. The right shift is guaranteed to be logical (0 fill) if `E1` is `unsigned`; otherwise, it may be arithmetic.

A.7.6. Relational Operators

The relational operators group left to right.

relational-expression:
expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. The type of the result is `int`. The usual arithmetic conversions are performed. Two pointers may be compared; the result depends on the relative locations in the address space of the pointed-to objects. Pointer comparison is portable only when the pointers point to objects in the same array.

A.7.7. Equality Operators

equality-expression:
expression == *expression*
expression != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus `a<b == c<d` is 1 whenever `a<b` and `c<d` have the same truth value).

A pointer may be compared to an integer only if the integer is the constant 0. A pointer to which 0 has been assigned is guaranteed not to point to any object and will appear to be equal to 0. In conventional usage, such a pointer is considered to be null.

A.7.8. Bitwise AND Operator

and-expression:
expression & *expression*

The & operator is associative, and expressions involving & may be rearranged. The usual arithmetic conversions are performed. The result is the bitwise AND function of the operands. The operator applies only to integral operands.

A.7.9. Bitwise Exclusive OR Operator

exclusive-or-expression:
expression ^ *expression*

The ^ operator is associative, and expressions involving ^ may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise exclusive OR function of the operands. The operator applies only to integral operands.

A.7.10. Bitwise Inclusive OR Operator

inclusive-or-expression:
expression | *expression*

The | operator is associative, and expressions involving | may be rearranged. The usual arithmetic conversions are performed; the result is the bitwise inclusive OR function of its operands. The operator applies only to integral operands.

A.7.11. Logical AND Operator

logical-and-expression:
expression && *expression*

The && operator groups left to right. It returns 1 if both its operands evaluate to nonzero, 0 otherwise. Unlike &, && guarantees left to right evaluation; moreover, the second operand is not evaluated if the first operand is 0.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

A.7.12. Logical OR Operator

logical-or-expression:
expression || *expression*

The || operator groups left to right. It returns 1 if either of its operands evaluates to nonzero, 0 otherwise. Unlike |, || guarantees left to right evaluation; moreover, the second operand is not evaluated if the value of the first operand is nonzero.

The operands need not have the same type, but each must have one of the fundamental types or be a pointer. The result is always `int`.

A.7.13. Conditional Operator

conditional-expression:
expression ? expression : expression

Conditional expressions group right to left. The first expression is evaluated; and if it is nonzero, the result is the value of the second expression, otherwise that of third expression. If possible, the usual arithmetic conversions are performed to bring the second and third expressions to a common type. If both are structures or unions of the same type, the result has the type of the structure or union. If both pointers are of the same type, the result has the common type. Otherwise, one must be a pointer and the other the constant 0, and the result has the type of the pointer. Only one of the second and third expressions is evaluated.

A.7.14. Assignment Operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place. The two parts of a compound assignment operator are separate tokens.

assignment-expression:
lvalue = expression
lvalue += expression
lvalue -= expression
*lvalue *= expression*
lvalue /= expression
lvalue %= expression
lvalue >>= expression
lvalue <<= expression
lvalue &= expression
lvalue ^= expression
lvalue |= expression

In the simple assignment with `=`, the value of the expression replaces that of the object referred to by the lvalue. If both operands have arithmetic type, the right operand is converted to the type of the left preparatory to the assignment. Second, both operands may be structures or unions of the same type. Finally, if the left operand is a pointer, the right operand must in general be a pointer of the same type. However, the constant 0 may be assigned to a pointer; it is guaranteed that this value will produce a null pointer distinguishable from a pointer to any object.

The behavior of an expression of the form `E1` *op* `=` `E2` may be inferred by taking it as equivalent to `E1 = E1` *op* `(E2)`; however, `E1` is evaluated only once. In `+=` and `-=`, the left operand may be a pointer; in which case, the (integral) right operand is converted as explained in Section A.7.4, “Additive Operators”. All right operands and all nonpointer left operands must have arithmetic type.

A.7.15. Comma Operator

comma-expression:
expression , expression

A pair of expressions separated by a comma is evaluated left to right, and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand. This operator groups left to right. In contexts where comma is given a special meaning, e.g., in lists of actual arguments to functions (see Section A.7.1, “Primary Expressions”) and lists of initializers (see Section A.8.6, “Initialization”), the comma operator as described in this subpart can only appear in parentheses. For example,

```
f(a, (t=3, t+2), c)
```

has three arguments, the second of which has the value 5.

A.8. Declarations

Declarations are used to specify the interpretation which C gives to each identifier; they do not necessarily reserve storage associated with the identifier. Declarations have the form

declaration:
decl-specifiers $declarator\text{-}list_{opt}$ *;*

The declarators in the declarator-list contain the identifiers being declared. The decl-specifiers consist of a sequence of type and storage class specifiers.

decl-specifiers:
type-specifier $decl\text{-}specifiers_{opt}$
sc-specifier $decl\text{-}specifiers_{opt}$

The list must be self-consistent in a way described below.

A.8.1. Storage Class Specifiers

The sc-specifiers are:

sc-specifier:
```
auto
static
extern
register
typedef
```

The `typedef` specifier does not reserve storage and is called a "storage class specifier" only for syntactic convenience. See Section A.8.8, "Typedef" for more information. The meanings of the various storage classes were discussed in Section A.4, "What's in a name?".

The `auto`, `static`, and `register` declarations also serve as definitions in that they cause an appropriate amount of storage to be reserved. In the `extern` case, there must be an external definition (see Section A.10, "External Definitions") for the given identifiers somewhere outside the function in which they are declared.

A `register` declaration is best thought of as an `auto` declaration, together with a hint to the compiler that the variables declared will be heavily used. Only the first few such declarations in each function are effective. Moreover, only variables of certain types will be stored in registers; on the PDP-11, they are `int` or pointer. One other restriction applies to register variables: the address-of operator & cannot be applied to them. Smaller, faster programs can be expected if register declarations are used appropriately, but future improvements in code generation may render them unnecessary.

At most, one sc-specifier may be given in a declaration. If the sc-specifier is missing from a declaration, it is taken to be `auto` inside a function, `extern` outside. Exception: functions are never automatic.

A.8.2. Type Specifiers

The type-specifiers are

type-specifier:
```
char
short
int
long
unsigned
float
double
```
struct-or-union-specifier
typedef-name

At most one of the words `long` or `short` may be specified in conjunction with `int`; the meaning is the same as if `int` were not mentioned. The word `long` may be specified in conjunction with `float`; the meaning is the same as `double`. The word `unsigned` may be specified alone, or in conjunction with `int` or any of its short or long varieties, or with `char`.

Otherwise, at most on type-specifier may be given in a declaration. In particular, adjectival use of `long`, `short`, or `unsigned` is not permitted with `typedef` names. If the type-specifier is missing from a declaration, it is taken to be `int`.

Specifiers for structures and unions are discussed in Section A.8.5, "Structure and Union Declarations". Declarations with `typedef` names are discussed in Section A.8.8, "Typedef".

A.8.3. Declarators

The declarator-list appearing in a declaration is a comma-separated sequence of declarators, each of which may have an initializer.

declarator-list:
init-declarator
init-declarator , declarator-list

init-declarator:
declarator initializer$_{opt}$

Initializers are discussed in Section A.8.6, “Initialization”. The specifiers in the declaration indicate the type and storage class of the objects to which the declarators refer. Declarators have the syntax:

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

The grouping is the same as in expressions.

A.8.4. Meaning of Declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type and storage class.

Each declarator contains exactly one identifier; it is this identifier that is declared. If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

A declarator in parentheses is identical to the unadorned declarator, but the binding of complex declarators may be altered by parentheses. See the examples below.

Now imagine a declaration

```
T D1
```

where `T` is a type-specifier (like `int`, etc.) and `D1` is a declarator. Suppose this declaration makes the identifier have type “... `T` ,” where the “...” is empty if `D1` is just a plain identifier (so that the type of `x` in “`int x`” is just `int`). Then if `D1` has the form

```
*D
```

the type of the contained identifier is “... pointer to `T` &.”

If `D1` has the form

```
D()
```

then the contained identifier has the type “... function returning `T`.” If `D1` has the form

`D[`*constant-expression*`]`

or

```
D[]
```

then the contained identifier has type "... array of `T`." In the first case, the constant expression is an expression whose value is determinable at compile time , whose type is `int`, and whose value is positive. (Constant expressions are defined precisely in Section A.15, "Constant Expressions") When several "array of" specifications are adjacent, a multidimensional array is created; the constant expressions which specify the bounds of the arrays may be missing only for the first member of the sequence. This elision is useful when the array is external and the actual definition, which allocates storage, is given elsewhere. The first constant expression may also be omitted when the declarator is followed by initialization. In this case the size is calculated from the number of initial elements supplied.

An array may be constructed from one of the basic types, from a pointer, from a structure or union, or from another array (to generate a multidimensional array).

Not all the possibilities allowed by the syntax above are actually permitted. The restrictions are as follows: functions may not return arrays or functions although they may return pointers; there are no arrays of functions although there may be arrays of pointers to functions. Likewise, a structure or union may not contain a function; but it may contain a pointer to a function.

As an example, the declaration

```
int i, *ip, f(), *fip(), (*pfi)();
```

declares an integer `i`, a pointer `ip` to an integer, a function `f` returning an integer, a function `fip` returning a pointer to an integer, and a pointer `pfi` to a function which returns an integer. It is especially useful to compare the last two. The binding of `*fip()` is `*(fip())`. The declaration suggests, and the same construction in an expression requires, the calling of a function `fip`. Using indirection through the (pointer) result to yield an integer. In the declarator `(*pfi)()`, the extra parentheses are necessary, as they are also in an expression, to indicate that indirection through a pointer to a function yields a function, which is then called; it returns an integer.

As another example,

```
float fa[17], *afp[17];
```

declares an array of `float` numbers and an array of pointers to `float` numbers. Finally,

```
static int x3d[3][5][7];
```

declares a static 3-dimensional array of integers, with rank 3×5×7. In complete detail, `x3d` is an array of three items; each item is an array of five arrays; each of the latter arrays is an array of seven integers. Any of the expressions `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` may reasonably appear in an expression. The first three have type "array" and the last has type `int`.

A.8.5. Structure and Union Declarations

A structure is an object consisting of a sequence of named members. Each member may have any type. A union is an object which may, at a given time, contain any one of several members. Structure and union specifiers have the same form.

struct-or-union-specifier:
struct-or-union { struct-decl-list }
struct-or-union identifier { struct-decl-list }

struct-or-union identifier

struct-or-union:
`struct`
`union`

The struct-decl-list is a sequence of declarations for the members of the structure or union:

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

In the usual case, a struct-declarator is just a declarator for a member of a structure or union. A structure member may also consist of a specified number of bits. Such a member is also called a *field ;* its length, a non-negative constant expression, is set off from the field name by a colon.

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

Within a structure, the objects declared have addresses which increase as the declarations are read left to right. Each nonfield member of a structure begins on an addressing boundary appropriate to its type; therefore, there may be unnamed holes in a structure. Field members are packed into machine integers; they do not straddle words. A field which does not fit into the space remaining in a word is put into the next word. No field may be wider than a word.

Fields are assigned right to left on the PDP-11 and VAX-11, left to right on the 3B 20.

A struct-declarator with no declarator, only a colon and a width, indicates an unnamed field useful for padding to conform to externally-imposed layouts. As a special case, a field with a width of 0 specifies alignment of the next field at an implementation dependant boundary.

The language does not restrict the types of things that are declared as fields, but implementations are not required to support any but integer fields. Moreover, even `int` fields may be considered to be unsigned. On the PDP-11, fields are not signed and have only integer values; on the VAX-11, fields declared with `int` are treated as containing a sign. For these reasons, it is strongly recommended that fields be declared as `unsigned`. In all implementations, there are no arrays of fields, and the address-of operator & may not be applied to them, so that there are no pointers to fields.

A union may be thought of as a structure all of whose members begin at offset 0 and whose size is sufficient to contain any of its members. At most, one of the members can be stored in a union at any time.

A structure or union specifier of the second form, that is, one of

`struct` *identifier { struct-decl-list }*
`union` *identifier { struct-decl-list }*

declares the identifier to be the *structure tag* (or union tag) of the structure specified by the list. A subsequent declaration may then use the third form of specifier, one of

`struct` *identifier*
`union` *identifier*

Structure tags allow definition of self-referential structures. Structure tags also permit the long part of the declaration to be given once and used several times. It is illegal to declare a structure or union which contains an instance of itself, but a structure or union may contain a pointer to an instance of itself.

The third form of a structure or union specifier may be used prior to a declaration which gives the complete specification of the structure or union in situations in which the size of the structure or union is unnecessary. The size is unnecessary in two situations: when a pointer to a structure or union is being declared and when a `typedef` name is declared to be a synonym for a structure or union. This, for example, allows the declaration of a pair of structures which contain pointers to each other.

The names of members and tags do not conflict with each other or with ordinary variables. A particular name may not be used twice in the same structure, but the same name may be used in several different structures in the same scope.

A simple but important example of a structure declaration is the following binary tree structure:

```
struct tnode
{
        char tword[20];
        int count;
        struct tnode *left;
        struct tnode *right;
};
```

which contains an array of 20 characters, an integer, and two pointers to similar structures. Once this declaration has been given, the declaration

```
struct tnode s, *sp;
```

declares `s` to be a structure of the given sort and `sp` to be a pointer to a structure of the given sort. With these declarations, the expression

```
sp->count
```

refers to the `count` field of the structure to which `sp` points;

```
s.left
```

refers to the left subtree pointer of the structure `s`; and

```
s.right->tword[0]
```

refers to the first character of the `tword` member of the right subtree of `s`.

A.8.6. Initialization

A declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and consists of an expression or a list of values nested in braces.

initializer:
= *expression*
= *{ initializer-list }*
= *{ initializer-list , }*

initializer-list:
expression
initializer-list , initializer-list
{ *initializer-list* }
{ *initializer-list* , }

All the expressions in an initializer for a static or external variable must be constant expressions, which are described in Section A.15, “Constant Expressions”, or expressions which reduce to the address of a previously declared variable, possibly offset by a constant expression. Automatic or register variables may be initialized by arbitrary expressions involving constants and previously declared variables and functions.

Static and external variables that are not initialized are guaranteed to start off as zero. Automatic and register variables that are not initialized are guaranteed to start off as garbage.

When an initializer applies to a *scalar* (a pointer or an object of arithmetic type), it consists of a single expression, perhaps in braces. The initial value of the object is taken from the expression; the same conversions as for assignment are performed.

When the declared variable is an *aggregate* (a structure or array), the initializer consists of a brace-enclosed, comma-separated list of initializers for the members of the aggregate written in increasing subscript or member order. If the aggregate contains subaggregates, this rule applies recursively to the members of the aggregate. If there are fewer initializers in the list than there are members of the aggregate, then the aggregate is padded with zeros. It is not permitted to initialize unions or automatic aggregates.

Braces may in some cases be omitted. If the initializer begins with a left brace, then the succeeding comma-separated list of initializers initializes the members of the aggregate; it is erroneous for there to be more initializers than members. If, however, the initializer does not begin with a left brace, then only enough elements from the list are taken to account for the members of the aggregate; any remaining members are left to initialize the next member of the aggregate of which the current aggregate is a part.

A final abbreviation allows a `char` array to be initialized by a string. In this case successive characters of the string initialize the members of the array.

For example,

```
int x[] = { 1, 3, 5 };
```

declares and initializes `x` as a one-dimensional array which has three members, since no size was specified and there are three initializers.

```
float y[4][3] =
{
        { 1, 3, 5 },
        { 2, 4, 6 },
        { 3, 5, 7 },
};
```

is a completely-bracketed initialization: 1, 3, and 5 initialize the first row of the array `y[0]`, namely `y[0][0]`, `y[0][1]`, and `y[0][2]`. Likewise, the next two lines initialize `y[1]` and `y[2]`. The

initializer ends early and therefore `y[3]` is initialized with 0. Precisely, the same effect could have been achieved by

```
float y[4][3] =
{
        1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

The initializer for `y` begins with a left brace but that for `y[0]` does not; therefore, three elements from the list are used. Likewise, the next three are taken successively for `y[1]` and `y[2]`. Also,

```
float y[4][3] =
{
        { 1 }, { 2 }, { 3 }, { 4 }
};
```

initializes the first column of `y` (regarded as a two-dimensional array) and leaves the rest 0.

Finally,

```
char msg[] = "Syntax error on line %s\n";
```

shows a character array whose members are initialized with a string.

A.8.7. Type Names

In two contexts (to specify type conversions explicitly by means of a cast and as an argument of `sizeof`), it is desired to supply the name of a data type. This is accomplished using a “type name”, which in essence is a declaration for an object of that type which omits the name of the object.

type-name:
    type-specifier abstract-declarator

abstract-declarator:
    empty
    (abstract-declarator)
    ** abstract-declarator*
    abstract-declarator ()
    abstract-declarator [*constant-expression*$_{opt}$]

To avoid ambiguity, in the construction

    (abstract-declarator)

the abstract-declarator is required to be nonempty. Under this restriction, it is possible to identify uniquely the location in the abstract-declarator where the identifier would appear if the construction were a declarator in a declaration. The named type is then the same as the type of the hypothetical identifier. For example,

```
int
int *
int *[3]
int (*)[3]
```

```
int *()
int (*)()
int (*[3])()
```

name respectively the types “integer,” “pointer to integer,” “array of three pointers to integers,” “pointer to an array of three integers,” “function returning pointer to integer,” “pointer to function returning an integer,” and “array of three pointers to functions returning an integer.”

A.8.8. Typedef

Declarations whose “storage class” is `typedef` do not define storage but instead define identifiers which can be used later as if they were type keywords naming fundamental or derived types.

typedef-name:
identifier

Within the scope of a declaration involving `typedef`, each identifier appearing as part of any declarator therein becomes syntactically equivalent to the type keyword naming the type associated with the identifier in the way described in Section A.8.4, “Meaning of Declarators”. For example, after

```
typedef int MILES, *KLICKSP;
typedef struct { double re, im; } complex;
```

the constructions

```
MILES distance;
extern KLICKSP metricp;
complex z, *zp;
```

are all legal declarations; the type of `distance` is `int`, that of `metricp` is “pointer to `int`, ” and that of `z` is the specified structure. The `zp` is a pointer to such a structure.

The `typedef` does not introduce brand-new types, only synonyms for types which could be specified in another way. Thus in the example above `distance` is considered to have exactly the same type as any other `int` object.

A.9. Statements

Except as indicated, statements are executed in sequence.

A.9.1. Expression Statement

Most statements are expression statements, which have the form

expression ;

Usually expression statements are assignments or function calls.

A.9.2. Compound Statement or Block

So that several statements can be used where one is expected, the compound statement (also, and equivalently, called “block”) is provided:

compound-statement:
{ declaration-list$_{opt}$ *statement-list*$_{opt}$ *}*

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

If any of the identifiers in the declaration-list were previously declared, the outer declaration is pushed down for the duration of the block, after which it resumes its force.

Any initializations of `auto` or `register` variables are performed each time the block is entered at the top. It is currently possible (but a bad practice) to transfer into a block; in that case the initializations are not performed. Initializations of `static` variables are performed only once when the program begins execution. Inside a block, `extern` declarations do not reserve storage so initialization is not permitted.

A.9.3. Conditional Statement

The two forms of the conditional statement are

`if` (*expression*) *statement*
`if` (*expression*) *statement* `else` *statement*

In both cases, the expression is evaluated; and if it is nonzero, the first substatement is executed. In the second case, the second substatement is executed if the expression is 0. The “else” ambiguity is resolved by connecting an `else` with the last encountered `else`-less `if`.

A.9.4. While Statement

The `while` statement has the form

`while` (*expression*) *statement*

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

A.9.5. Do Statement

The `do` statement has the form

`do` *statement* `while` (*expression*) ;

The substatement is executed repeatedly until the value of the expression becomes 0. The test takes place after each execution of the statement.

A.9.6. For Statement

The `for` statement has the form:

`for` *(exp-1$_{opt}$; exp-2$_{opt}$; exp-3$_{opt}$) statement*

Except for the behavior of `continue`, this statement is equivalent to

exp-1 ;
`while` (*exp-2)*
{
    statement
    exp-3 ;
}

Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0. The third expression often specifies an incrementing that is performed after each iteration.

Any or all of the expressions may be dropped. A missing *exp-2* makes the implied `while` clause equivalent to `while(1)`; other missing expressions are simply dropped from the expansion above.

A.9.7. Switch Statement

The `switch` statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form

`switch` (*expression*) *statement*

The usual arithmetic conversion is performed on the expression, but the result must be `int`. The statement is typically compound. Any statement within the statement may be labeled with one or more case prefixes as follows:

`case` *constant-expression* :

where the constant expression must be `int`. No two of the case constants in the same switch may have the same value. Constant expressions are precisely defined in Section A.15, “Constant Expressions”.

There may also be at most one statement prefix of the form

```
default :
```

When the `switch` statement is executed, its expression is evaluated and compared with each case constant. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression and if there is a `default`, prefix, control passes to the prefixed statement. If no case matches and if there is no `default`, then none of the statements in the switch is executed.

The prefixes `case` and `default` do not alter the flow of control, which continues unimpeded across such prefixes. To exit from a switch, see Section A.9.8, “Break Statement”.

Usually, the statement that is the subject of a switch is compound. Declarations may appear at the head of this statement, but initializations of automatic or register variables are ineffective.

A.9.8. Break Statement

The statement

```
break ;
```

causes termination of the smallest enclosing `while`, `do`, `for`, or `switch` statement; control passes to the statement following the terminated statement.

A.9.9. Continue Statement

The statement

```
continue ;
```

causes control to pass to the loop-continuation portion of the smallest enclosing `while`, `do`, or `for` statement; that is to the end of the loop. More precisely, in each of the statements

while (...) {	do {	for (...) {
statement ;	statement ;	statement ;
contin: ;	contin: ;	contin: ;
}	} while (...);	}

a `continue` is equivalent to `goto contin`. (Following the `contin:` is a null statement, see Section A.9.13, “Null Statement”.)

A.9.10. Return Statement

A function returns to its caller by means of the `return` statement which has one of the forms

```
return ;
return expression ;
```

In the first case, the returned value is undefined. In the second case, the value of the expression is returned to the caller of the function. If required, the expression is converted, as if by assignment, to the type of function in which it appears. Flowing off the end of a function is equivalent to a return with no returned value. The expression may be parenthesized.

A.9.11. Goto Statement

Control may be transferred unconditionally by means of the statement

`goto` *identifier* ;

The identifier must be a label (see Section A.9.12, “Labeled Statement”) located in the current function.

A.9.12. Labeled Statement

Any statement may be preceded by label prefixes of the form

identifier :

which serve to declare the identifier as a label. The only use of a label is as a target of a `goto`. The scope of a label is the current function, excluding any subblocks in which the same identifier has been redeclared. See Section A.11, “Scope Rules”

A.9.13. Null Statement

The null statement has the form

```
;
```

A null statement is useful to carry a label just before the `}` of a compound statement or to supply a null body to a looping statement such as `while`.

A.10. External Definitions

A C program consists of a sequence of external definitions. An external definition declares an identifier to have storage class `extern` (by default) or perhaps `static`, and a specified type. The type-specifier (see Section A.8.2, “Type Specifiers”) may also be empty, in which case the type is taken to be `int`. The scope of external definitions persists to the end of the file in which they are declared just as the effect of declarations persists to the end of a block. The syntax of external definitions is the same as that of all declarations except that only at this level may the code for functions be given.

A.10.1. External Function Definitions

Function definitions have the form

function-definition:
decl-specifiers$_{opt}$ *function-declarator function-body*

The only sc-specifiers allowed among the decl-specifiers are `extern` or `static`; see Section A.11.2, “Scope of Externals” for the distinction between them. A function declarator is similar to a declarator for a “function returning ...” except that it lists the formal parameters of the function being defined.

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

The function-body has the form

function-body:
declaration-list$_{opt}$ *compound-statement*

The identifiers in the parameter list, and only those identifiers, may be declared in the declaration list. Any identifiers whose type is not given are taken to be `int`. The only storage class which may be specified is `register`; if it is specified, the corresponding actual parameter will be copied, if possible, into a register at the outset of the function.

A simple example of a complete function definition is

```
int max(a, b, c)
        int a, b, c;
{
        int m;

        m = (a > b) ? a : b;
        return((m > c) ? m : c);
}
```

Here `int` is the type-specifier; `max(a, b, c)` is the function-declarator; `int a, b, c;` is the declaration-list for the formal parameters; `{ ... }` is the block giving the code for the statement.

The C program converts all `float` actual parameters to `double`, so formal parameters declared `float` have their declaration adjusted to read `double`. All `char` and `short` formal parameter declarations are similarly adjusted to read `int`. Also, since a reference to an array in any context (in particular as an actual parameter) is taken to mean a pointer to the first element of the array, declarations of formal parameters declared “array of ...” are adjusted to read “pointer to ....”

A.10.2. External Data Definitions

An external data definition has the form

data-definition:
    declaration

The storage class of such data may be `extern` (which is the default) or `static` but not `auto` or `register`.

A.11. Scope Rules

A C program need not all be compiled at the same time. The source text of the program may be kept in several files, and precompiled routines may be loaded from libraries. Communication among the functions of a program may be carried out both through explicit calls and through manipulation of external data.

Therefore, there are two kinds of scopes to consider: first, what may be called the *lexical scope* of an identifier, which is essentially the region of a program during which it may be used without drawing “undefined identifier” diagnostics; and second, the scope associated with external identifiers, which is characterized by the rule that references to the same external identifier are references to the same object.

A.11.1. Lexical Scope

The lexical scope of identifiers declared in external definitions persists from the definition through the end of the source file in which they appear. The lexical scope of identifiers which are formal parameters persists through the function with which they are associated. The lexical scope of identifiers declared at the head of a block persists until the end of the block. The lexical scope of labels is the whole of the function in which they appear.

In all cases, however, if an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of that identifier outside the block is suspended until the end of the block.

Remember also (see Section A.8.5, “Structure and Union Declarations”) that identifiers associated with ordinary variables, and those associated with structure and union members form two disjoint classes which do not conflict. Members and tags follow the same scope rules as other identifiers. `typedef` names are in the same class as ordinary identifiers. They may be redeclared in inner blocks, but an explicit type must be given in the inner declaration:

```
typedef float distance;
...
{
     auto int distance;
     ...
}
```

The `int` must be present in the second declaration, or it would be taken to be a declaration with no declarators and type `distance`.

A.11.2. Scope of Externals

If a function refers to an identifier declared to be `extern`, then somewhere among the files or libraries constituting the complete program there must be at least one external definition for the identifier. All functions in a given program which refer to the same external identifier refer to the same object, so care must be taken that the type and size specified in the definition are compatible with those specified by each function which references the data.

It is illegal to explicitly initialize any external identifier more than once in the set of files and libraries comprising a multi-file program. It is legal to have more than one data definition for any external non-function identifier; explicit use of `extern` does not change the meaning of an external declaration.

In restricted environments, the use of the `extern` storage class takes on an additional meaning. In these environments, the explicit appearance of the `extern` keyword in external data declarations of identities without initialization indicates that the storage for the identifiers is allocated elsewhere, either in this file or another file. It is required that there be exactly one definition of each external identifier (without `extern`) in the set of files and libraries comprising a mult-file program.

Identifiers declared `static` at the top level in external definitions are not visible in other files. Functions may be declared `static`.

A.12. Compiler Control Lines

The C compiler contains a preprocessor capable of macro substitution, conditional compilation, and inclusion of named files. Lines beginning with # communicate with this preprocessor. There may be any number of blanks and horizontal tabs between the # and the directive. These lines have syntax independent of the rest of the language; they may appear anywhere and have effect which lasts (independent of scope) until the end of the source program file.

A.12.1. Token Replacement

A compiler-control line of the form

`#define` *identifier token-string*$_{opt}$

causes the preprocessor to replace subsequent instances of the identifier with the given string of tokens. Semicolons in or at the end of the token-string are part of that string. A line of the form

`#define` *identifier(identifier, ...) token-string*$_{opt}$

where there is no space between the first identifier and the `(`, is a macro definition with arguments. There may be zero or more formal parameters. Subsequent instances of the first identifier followed by a `(`, a sequence of tokens delimited by commas, and a `)` are replaced by the token string in the definition. Each occurrence of an identifier mentioned in the formal parameter list of the definition is replaced by the corresponding token string from the call. The actual arguments in the call are token strings separated by commas; however, commas in quoted strings or protected by parentheses do not separate arguments. The number of formal and actual parameters must be the same. Strings and character constants in the token-string are scanned for formal parameters, but strings and character constants in the rest of the program are not scanned for defined identifiers to replacement.

In both forms the replacement string is rescanned for more defined identifiers. In both forms a long definition may be continued on another line by writing \ at the end of the line to be continued.

This facility is most valuable for definition of “manifest constants,” as in

```
#define TABSIZE 100
```

```
int table[TABSIZE];
```

A control line of the form

#undef *identifier*

causes the identifier's preprocessor definition (if any) to be forgotten.

If a #defined identifier is the subject of a subsequent #define with no intervening #undef, then the two token-strings are compared textually. If the two token-strings are not identical (all white space is considered as equivalent), then the identifier is considered to be redefined.

A.12.2. File Inclusion

A compiler control line of the form

#include *"filename"*

causes the replacement of that line by the entire contents of the file *filename*. The named file is searched for first in the directory of the file containing the #include, and then in a sequence of specified or standard places. Alternatively, a control line of the form

#include <*filename*>

searches only the specified or standard places and not the directory of the #include. (How the places are specified is not part of the language.)

#includes may be nested.

A.12.3. Conditional Compilation

A compiler control line of the form

#if *constant-expression*

checks whether the constant expression evaluates to nonzero. (Constant expressions are discussed in Section A.15, “Constant Expressions”. A control line of the form

#ifdef *identifier*

checks whether the identifier is currently defined in the preprocessor; i.e., whether it has been the subject of a #define control line. It is equivalent to #ifdef(*identifier*). A control line of the form

#ifndef *identifier*

checks whether the identifier is currently undefined in the preprocessor. It is equivalent to

#if !defined(*identifier*).

All three forms are followed by an arbitrary number of lines, possibly containing a control line

#else

and then by a control line

```
#endif
```

If the checked condition is true, then any lines between `#else` and `#endif` are ignored. If the checked condition is false, then any lines between the test and a `#else` or, lacking a `#else`, the `#endif` are ignored.

These constructions may be nested.

A.12.4. Line Control

For the benefit of other preprocessors which generate C programs, a line of the form

`#line` *constant identifier*

causes the compiler to believe, for purposes of error diagnostics, that the line number of the next source line is given by the constant and the current input file is named by the identifier. If the identifier is absent, the remembered file name does not change.

A.13. Implicit Declarations

It is not always necessary to specify both the storage class and the type of identifiers in a declaration. The storage class is supplied by the context in external definitions and in declarations of formal parameters and structure members. In a declaration inside a function, if a storage class but no type is given, the identifier is assumed to be `int`; if a type but no storage class is indicated, the identifier is assumed to be `auto`. An exception to the latter rule is made for functions because `auto` functions do not exist. If the type of an identifier is "function returning ...," it is implicitly declared to be `extern`.

In an expression, an identifier followed by `(` and not already declared is contextually declared to be "function returning `int`."

A.14. Types Revisited

This part summarizes the operations which can be performed on objects of certain types.

A.14.1. Structures and Unions

Structures and unions may be assigned, passed as arguments to functions, and returned by functions. Other plausible operators, such as equality comparison and structure casts, are not implemented.

In a reference to a structure or union member, the name on the right of the `->` or the `.` must specify a member of the aggregate named or pointed to by the expression on the left. In general, a member of a union may not be inspected unless the value of the union has been assigned using that same member. However, one special guarantee is made by the language in order to simplify the use of unions: if a union contains several structures that share a common initial sequence and if the union currently contains one of these structures, it is permitted to inspect the common initial part of any of the contained structures.

A.14.2. Functions

There are only two things that can be done with a function `m`, call it or take its address. If the name of a function appears in an expression not in the function-name position of a call, a pointer to the function is generated. Thus, to pass one function to another, one might say

```
int f();
```

```
...
g(f);
```

Then the definition of `g` might read

```
g(funcp)
        int (*funcp)();
{
        ...
        (*funcp)();
        ...
}
```

Notice that `f` must be declared explicitly in the calling routine since its appearance in `g(f)` was not followed by `(`.

A.14.3. Arrays, Pointers, and Subscripting

Every time an identifier of array type appears in an expression, it is converted into a pointer to the first member of the array. Because of this conversion, arrays are not lvalues. By definition, the subscript operator `[]` is interpreted in such a way that `E1[E2]` is identical to `*((E1)+E2))`. Because of the conversion rules which apply to `+`, if `E1` is an array and `E2` an integer, then `E1[E2]` refers to the `E2-th` member of `E1`. Therefore, despite its asymmetric appearance, subscripting is a commutative operation.

A consistent rule is followed in the case of multidimensional arrays. If `E` is an *n*-dimensional array of rank i×j×...×k, then `E` appearing in an expression is converted to a pointer to an (n-1)-dimensional array with rank j×...×k. If the `*` operator, either explicitly or implicitly as a result of subscripting, is applied to this pointer, the result is the pointed-to (n-1)-dimensional array, which itself is immediately converted into a pointer.

For example, consider

```
int x[3][5];
```

Here `x` is a 3×5 array of integers. When `x` appears in an expression, it is converted to a pointer to (the first of three) 5-membered arrays of integers. In the expression `x[i]`, which is equivalent to `*(x +i)`, `x` is first converted to a pointer as described; then `i` is converted to the type of `x`, which involves multiplying `i` by the length the object to which the pointer points, namely 5-integer objects. The results are added and indirection applied to yield an array (of five integers) which in turn is converted to a pointer to the first of the integers. If there is another subscript, the same argument applies again; this time the result is an integer.

Arrays in C are stored row-wise (last subscript varies fastest) and the first subscript in the declaration helps determine the amount of storage consumed by an array. Arrays play no other part in subscript calculations.

A.14.4. Explicit Pointer Conversions

Certain conversions involving pointers are permitted but have implementation-dependent aspects. They are all specified by means of an explicit type-conversion operator, see Section A.7.2, “Unary Operators” and Section A.8.7, “Type Names”.

A pointer may be converted to any of the integral types large enough to hold it. Whether an `int` or `long` is required is machine dependent. The mapping function is also machine dependent but is intended to be unsurprising to those who know the addressing structure of the machine. Details for some particular machines are given below.

An object of integral type may be explicitly converted to a pointer. The mapping always carries an integer converted from a pointer back to the same pointer but is otherwise machine dependent.

A pointer to one type may be converted to a pointer to another type. The resulting pointer may cause addressing exceptions upon use if the subject pointer does not refer to an object suitably aligned in storage. It is guaranteed that a pointer to an object of a given size may be converted to a pointer to an object of a smaller size and back again without change.

For example, a storage-allocation routine might accept a size (in bytes) of an object to allocate, and return a `char` pointer; it might be used in this way.

```
extern char *alloc();
double *dp;

dp = (double *) alloc(sizeof(double));
*dp = 22.0 / 7.0;
```

The `alloc` must ensure (in a machine-dependent way) that its return value is suitable for conversion to a pointer to `double`; then the *use* of the function is portable.

The pointer representation on the PDP-11 corresponds to a 16-bit integer and measures bytes. The `char`'s have no alignment requirements; everything else must have an even address.

On the VAX-11, pointers are 32 bits long and measure bytes. Elementary objects are aligned on a boundary equal to their length, except that `double` quantities need be aligned only on even 4-byte boundaries. Aggregates are aligned on the strictest boundary required by any of their constituents.

The 3B 20 computer has 24-bit pointers placed into 32-bit quantities. Most objects are aligned on 4-byte boundaries. `short`s are aligned in all cases on 2-byte boundaries. Arrays of characters, all structures, `int`s, `long`s, `float`s, and `double`s are aligned on 4-byte boundries; but structure members may be packed tighter.

A.15. Constant Expressions

In several places C requires expressions which evaluate to a constant: after `case`, as array bounds, and in initializers. In the first two cases, the expression can involve only integer constants, character constants, and `sizeof` expressions, possibly connected by the binary operators

```
+ - * / % & | ^ << >> == != < > <= >= && ||
```

or by the unary operators

```
-  ~
```

or by the ternary operator

```
?:
```

Parentheses can be used for grouping but not for function calls.

More latitude is permitted for initializers; besides constant expressions as discussed above, one can also use floating constants and arbitrary casts and can also apply the unary & operator to external or static objects and to external or static arrays subscripted with a constant expression. The unary & can also be applied implicitly by appearance of unsubscripted arrays and functions. The basic rule is that initializers must evaluate either to a constant or to the address of a previously declared external or static object plus or minus a constant.

A.16. Portability Considerations

Certain parts of C are inherently machine dependent. The following list of potential trouble spots is not meant to be all-inclusive but to point out the main ones.

Purely hardware issues like word size and the properties of floating point arithmetic and integer division have proven in practice to be not much of a problem. Other facets of the hardware are reflected in differing implementations. Some of these, particularly sign extension (converting a negative character into a negative integer) and the order in which bytes are placed in a word, are nuisances that must be carefully watched. Most of the others are only minor problems.

The number of `register` variables that can actually be placed in registers varies from machine to machine as does the set of valid types. Nonetheless, the compilers all do things properly for their own machine; excess or invalid `register` declarations are ignored.

Some difficulties arise only when dubious coding practices are used. It is exceedingly unwise to write programs that depend on any of these properties.

The order of evaluation of function arguments is not specified by the language. The order in which side effects take place is also unspecified.

Since character constants are really objects of type `int`, multicharacter character constants may be permitted. The specific implementation is very machine dependent because the order in which characters are assigned to a word varies from one machine to another.

Fields are assigned to words and characters to integers right to left on some machines and left to right on other machines. These differences are invisible to isolated programs that do not indulge in type punning (e.g., by converting an `int` pointer to a `char` pointer and inspecting the pointed-to storage) but must be accounted for when conforming to externally-imposed storage layouts.

The language accepted by the various compilers differs in minor details. Most notably, the current PDP-11 compiler will not initialize structures containing bitfields, and does not accept a few assignment operators in certain contexts where the value of the assignment is used.

A.17. Anachronisms

Since C is an evolving language, certain obsolete constructions may be found in older programs. Although most versions of the compiler support such anachronisms, ultimately they will disappear, leaving only a portability problem behind.

Earlier versions of C used the form *=op* instead of *op=* for assignment operators. This leads to ambiguities, typified by:

```
x=-1
```

which actually decrements x since the = and the - are adjacent, but which might easily be intended to assign -1 to x.

The syntax of initializers has changed: previously, the equals sign that introduces and initializer was not present, so instead of

```
int x    = 1;
```

one used

```
int x    1;
```

The change was made because the initialization

```
int f  (1+2)
```

resembles a function declaration closely enough to confuse the compilers.

A.18. Syntax Summary

This summary of C syntax is intended more for aiding comprehension than as an exact statement of the language.

A.18.1. Expressions

The basic expressions are:

expression:
primary
** expression*
&lvalue
- expression
! expression
~ expression
++ lvalue
--lvalue
lvalue ++
lvalue --
`sizeof` *expression*
`sizeof (`*type-name*`)`
(type-name) expression
expression binop expression
expression ? expression : expression
lvalue asgnop expression
expression , expression

primary:
identifier
constant
string
(expression)
primary (expression-list$_{opt}$)
primary [expression]
primary . identifier
primary - identifier

lvalue:
identifier
primary [expression]
lvalue . identifier
primary - identifier
** expression*
(lvalue)

The primary-expression operators

() [] . -<

have highest priority and group left to right. The unary operators

* & - ! ~ ++ -- `sizeof` *(type-name)*

have priority below the primary operators but higher than any binary operator and group right to left. Binary operators group left to right; they have priority decreasing as indicated below.

binop:
* / %
+ -
>> <<
< > <= >=
== !=
&
^
|
&&
||

The conditional operator groups right to left.

Assignment operators all have the same priority and all group right to left.

asgnop:
= += -= *= /= %= >>= <<= &= ^= |=

The comma operator has the lowest priority and groups left to right.

A.18.2. Declarations

declaration:
decl-specifiers init-declarator-list$_{opt}$ *;*

decl-specifiers:
type-specifier decl-specifiers$_{opt}$
sc-specifier decl-specifiers$_{opt}$

sc-specifier:
```
auto
static
extern
register
typedef
```

type-specifier:
```
char
short
int
long
unsigned
float
```

`double`
struct-or-union-specifier
typedef-name

init-declarator-list:
init-declarator
init-declarator , init-declarator-list

init-declarator:
declarator initializer$_{opt}$

declarator:
identifier
(declarator)
** declarator*
declarator ()
declarator [constant-expression$_{opt}$ *]*

struct-or-union-specifier:
`struct` *{ struct-decl-list }*
`struct` *identifier { struct-decl-list }*
`struct` *identifier*
`union` `{` *struct-decl-list }*
`union` *identifier { struct-decl-list }*
`union` *identifier*

struct-decl-list:
struct-declaration
struct-declaration struct-decl-list

struct-declaration:
type-specifier struct-declarator-list ;

struct-declarator-list:
struct-declarator
struct-declarator , struct-declarator-list

struct-declarator:
declarator
declarator : constant-expression
: constant-expression

initializer:
= expression
= { initializer-list }
= { initializer-list , }

initializer-list:
expression
initializer-list , initializer-list

{ initializer-list }
{ initializer-list , }

type-name:
type-specifier abstract-declarator

abstract-declarator:
empty
(abstract-declarator)
** abstract-declarator*
abstract-declarator ()
abstract-declarator [constant-expression$_{opt}$]

typedef-name:
identifier

A.18.3. Statements

compound-statement:
{ declaration-list$_{opt}$ statement-list$_{opt}$ }

declaration-list:
declaration
declaration declaration-list

statement-list:
statement
statement statement-list

statement:
compound-statement
expression ;
`if` *(expression) statement*
`if` *(expression) statement* `else` *statement*
`while` *(expression) statement*
`do` *statement* `while` *(expression) ;*
`for` *(exp$_{opt}$';' exp$_{opt}$';' exp$_{opt}$') statement*
`switch` *(expression) statement*
`case` *constant-expression : statement*
`default` *: statement*
`break ;`
`continue ;`
`return ;`
`return` *expression ;*
`goto` *identifier ;*
identifier : statement
;

A.18.4. External definitions

program:

external-definition
external-definition program

external-definition:
function-definition
data-definition

function-definition:
decl-specifier$_{opt}$ *function-declarator function-body*

function-declarator:
declarator (parameter-list$_{opt}$ *)*

parameter-list:
identifier
identifier , parameter-list

function-body:
declaration-list$_{opt}$ *compound-statement*

data-definition:
`extern` *declaration* `;`
`static` *declaration* `;`

A.18.5. Preprocessor

`#define` *identifier token-string*$_{opt}$
`#define` *identifier*`(`*identifier*`,...)`*token-string*$_{opt}$
`#undef` *identifier*
`#include "`*filename*`"`
`#include <`*filename*`>`
`#if` *constant-expression*
`#ifdef` *identifier*
`#ifndef` *identifier*
`#else`
`#endif`
`#line` *constant identifier*

Appendix B. Compiler Generated Error Messages

Below is a list of the error messages that the C compiler generates, and, if applicable, probable causes and the K & R Appendix A section number (in parenthesis) to see for more specific information.

already a local variable	Variable has already been declared at the current block level. (Section A.8.1, “Storage Class Specifiers”, Section A.9.2, “Compound Statement or Block”)
argument : <text>	Error from preprocessor. Self-explanatory. Most common cause of this error is not being able to find an include file.
argument error	Function argument declared as type struct, union or function. Pointers to such types, however are allowed. (Section A.10.1, “External Function Definitions”)
argument storage	Function arguments may only be declared as storage class register. (Section A.10.1, “External Function Definitions”)
bad character	A character not in the C character set (probably a control char) was encountered in the source file. (2)
both must be integral	>> and << operands cannot be FLOAT or DOUBLE. (Section A.7.5, “Shift Operators”)
break error	The break statement is allowed only inside a while, do, for or switch block. (Section A.9.8, “Break Statement”)
can't take address	& operator not allowed in a register variable. Operand must otherwise be an lvalue. (Section A.7.2, “Unary Operators”)
cannot cast	Type result of cast cannot be FUNCTION or ARRAY. (Section A.7.2, “Unary Operators”, Section A.8.7, “Type Names”)
cannot evaluate size	Could not determine size from declaration or initializer. (Section A.8.6, “Initialization”, Section A.14.3, “Arrays, Pointers, and Subscripting”)
cannot initialize	Storage class or type does not allow variable to be initialized. (Section A.8.6, “Initialization”)
compiler trouble	Compiler detected something it couldn't handle. Try compiling the program again. If this error still occurs, contact Microware.
condition needed	While, do, for, switch and if statements require a condition expression. (Section A.9.3, “Conditional Statement”)
constant expression required	Initializer expressions for static or external variables cannot reference variables. They may, however, refer to the address of a previously declared variable. This installation allows no initializer expressions unless all operands are of type INT or CHAR (Section A.8.6, “Initialization”)
constant overflow	Input numeric constant was too large for the implied or explicit type. (Section A.2.6, “Hardware Characteristics”, [PDP-11])

constant required	Variables are not allowed for array dimensions or cases. (Section A.8.3, “Declarators”, Section A.8.7, “Type Names”, Section A.9.7, “Switch Statement”)
continue error	The continue statement is allowed only inside a while, do, or for block. (Section A.9.9, “Continue Statement”)
declaration mismatch	This declaration conflicts with a previous one. This is typically caused by declaring a function to return a non-integer type after a reference has been made to the function. Depending on the line structure of the declaration block, this error may be reported on the line following the erroneous declaration. (Section A.11, “Scope Rules”, Section A.11.1, “Lexical Scope”, Section A.11.2, “Scope of Externals”)
divide by zero	Divide by zero occurred when evaluating a constant expression.
? expected	? is any character that was expected to appear here. Missing semicolons or braces cause this error.
expression missing	An expression is required here.
function header missing	Statement or expression encountered outside a function. Typically causes by mismatched braces. (Section A.10.1, “External Function Definitions”)
function type error	A function cannot be declared as returning an array, function, struct, or union. (Section A.8.4, “Meaning of Declarators”, Section A.10.1, “External Function Definitions”)
function unfinished	End-of-file encountered before the end of function definition. (Section A.10.1, “External Function Definitions”)
identifier missing	Identifier name required here but none was found.
illegal declaration	Declarations are allowed only at the beginning of a block. (Section A.9.2, “Compound Statement or Block”)
label required	Label name required on goto statement. (Section A.9.1, “Expression Statement”1)
label undefined	Goto to label not defined in the current function. (Section A.9.12, “Labeled Statement”)
lvalue required	Left side of assignment must be able to be "stored into". Array names, functions, structs, etc. are not lvalues. (Section A.7.1, “Primary Expressions”)
multiple defaults	Only one default statement is allowed in a switch block. (Section A.9.7, “Switch Statement”)
multiple definition	Identifier name was declared more than once in the same block level (Section A.9.2, “Compound Statement or Block”, Section A.11.1, “Lexical Scope”)
must be integral	Type of object required here must be type int, char or pointer.
name clash	Struct-union member and tag names must be mutually distinct. (Section A.8.5, “Structure and Union Declarations”)
name in cast	Identifier name found in a cast. Only types are allowed. (Section A.7.2, “Unary Operators”, Section A.8.7, “Type Names”)

named twice	Names in a function parameter list may appear only once. (Section A.10.1, “External Function Definitions”)
no 'if' for 'else'	Else statement found with no matching if. This is typically caused by extra or missing braces and/or semicolons. (Section A.9.3, “Conditional Statement”)
no switch statement	Case statements can only appear within a switch block. (Section A.9.7, “Switch Statement”)
not a function	Primary in expression is not type "function returning...". If this is really a function call, the function name was declared differently elsewhere. (Section A.7.1, “Primary Expressions”)
not an argument	Name does not appear in the function parameter list. (Section A.10.1, “External Function Definitions”)
operand expected	Unary operators require one operand, binary operators two. This is typically caused by misplaced parenthesis, casts or operators. (Section A.7.1, “Primary Expressions”)
out of memory	Compiler dynamic memory overflow. The compiler requires dynamic memory for symbol table entries, block level declarations and code generation. Three major factors affect this memory usage. Permanent declarations (those appearing on the outer block level (used in include files)) must be reserved from the dynamic memory for the duration of the compilation of the file. Each { causes the compiler to perform a block-level recursion which may involve "pushing down" previous declarations which consume memory. Auto class initializers require saving expression trees until past the declarations which may be very memory-expensive if may exist. Avoiding excessive declarations, both permanent and inside compound statement blocks conserve memory. If this error occurs on an auto initializer, try initializing the value in the code body.
pointer mismatch	Pointers refer to different types. Use a case if required. (Section A.7.1, “Primary Expressions”)
pointer or integer required	A pointer (of any type) or integer is required to the left of the '->' operator. (Section A.7.1, “Primary Expressions”)
pointer required	Pointer operand required with unary * operator. (Section A.7.1, “Primary Expressions”)
primary expected	Primary expression required here. (Section A.7.1, “Primary Expressions”)
should be NULL	Second and third expression of ?: conditional operator cannot be pointers to different types. If both are pointers, they must be of the same type or one of the two must be null. (Section A.7.13, “Conditional Operator”)
**** STACK OVERFLOW ****	Compiler stack has overflowed. Most likely cause is very deep lock-level nesting or hundreds of switch cases.
storage error	Reg and auto storage classes mey only be used within functions. (Section A.8.1, “Storage Class Specifiers”)
struct member mismatch	Identical member names in two different structures must have the same type and offset in both. (Section A.8.5, “Structure and Union Declarations”)

struct member required	Identifier used with . and -> operators must be a structure member name. (Section A.7.1, “Primary Expressions”)
struct syntax	Brace, comma, etc. is missing in a struct declaration. (Section A.8.5, “Structure and Union Declarations”)
struct or union inappropiate	Struct or union cannot be used in the context.
syntax error	Expression, declaration or statement is incorrectly formed.
third expression missing	? must be followed by a : with expression. This error may be causes by unmatched parenthesis or other errors in the expression. (Section A.7.13, “Conditional Operator”)
too long	Too many characters provided in a string initializing a character array. (Section A.8.6, “Initialization”)
too many brackets	Unmatched or unexpected brackets encountered processiong an initializer. (Section A.8.6, “Initialization”)
too many elements	More data items supplied for aggregate level in initializer than members of the aggregate. (Section A.8.6, “Initialization”)
type error	Compiler type matching error. Should never happen.
type mismatch	Types and/or operators in expression do not correspond. (6)
typedef - not a variable	Typedef type name cannot be used in this manner. (Section A.8.8, “Typedef”)
undeclared variable	no declaration exists at any block level for this identifier.
undefined structure	Union or struct declaration refers to an undefined structure name. (Section A.8.5, “Structure and Union Declarations”)
unions not allowed	Cannot initialize union members. (Section A.8.6, “Initialization”)
unterminated character constant	Unmatched ' character delimiters. (Section A.2.4.3, “Character Constants”)
unterminated string	Unmatched " string delimiters. (Section A.2.5, “Strings”)
while expected	No while found for do statement. (Section A.9.5, “Do Statement”)

Appendix C. Compiler Phase Command Lines

This appendix describes the command lines and options for the individual compiler phases. Each phase of the compiler may be executed separately. The following information describes the options available to each phase.

C.1. cc1 & cc2 (C executives)

`cc` [options] *`file`*... [options]

Recognized file suffixes:

.c	C source file
.a	Assembly language source file
.r	Relocatable module format file

Recognized options: (UPPER and lower case is equiv.)

`-a`	Suppress assembly. Leave output in ".a" file.
`-e`=n	Edition number (n) is supplied to c.prep for inclusion in module psect and/or to c.link for inclusion as the edition number of the linked module.
`-o`	Inhibits assembly code optimizer pass.
`-p`	Invoke compiler function profiler.
`-r`	Suppress link step. Leave output in ".r" file.
`-m=`*`size`*	Size in pages (in kbytes if followed by a K) of additional memory the linker should allocate to object module.
`-l=`*`path`*	Library file for linker to search before the standard library.
`-f=`*`path`*	Override other output naming. Module name (in object module) is the last name in the pathlist. -f is not allowed with -a or -r.
`-c`	Output comments in assembly language code.
`-s`	Suppress generation of stack-checking code.
`-d`*`NAME`*	Is equivalent to #define *`NAME`* 1 in the preprocessor. -d*`NAME`*=*`STRING`* is equivalent to #define *`NAME STRING`*.
`-n=`*`name`*	output module name. *`name`* is used to override the -f default output name.

CC1 only:

`-x`	Create, but do not execute c.com command file.

CC2 only:

`-q`	Quiet mode. Suppress echo of file names.

C.2. c.prep (C macro preprocessor)

`c.prep` [options] *`path`* [options]

`path` is read as input. C.prep causes c.comp to generate psect directive with last element of pathlist and _c as the psect name. If *`path`* is /d0/myprog.c, psect name is myprog_c. Output is always to stdout.

Recognized options:

`-l`	Cause c.comp to copy source lines to assembly output as comments.
`-E=`*`n`*	
`-e=`*`n`*	Use *`n`* as psect edition number.
`-D`*`NAME`*	Same as described above for cc1/cc2.

C.3. c.comp (One-pass compiler)

`c.comp` [options] [*`file`*] [options]

If *`file`* is not present, c.comp will read stdin. Input text need not be c.prep output, but no preprocessor directives are recognized (#include, #define, macros etc.). Output assembly code is normally to stdout. Error message output is always written to stdout.

Recognized options:

`-s`	Suppress generation of stack checking code.
`-p`	Generate profile code.
`-o=`*`path`*	Write assembly output to *`path`*.

C.4. c.pass (Pass One/Two of Two-pass Compiler)

`c.pass1` [options] [*`file`*] [options]

`c.pass2` [options] [*`file`*] [options]

Command line and options are the same as c.comp. If the options given to c.pass1 are not given to c.pass2 also, c.pass2 will not be able to read the c.pass1 output. Both c.pass1 and c.pass2 read stdin and write stdout normally.

C.5. c.opt (Assembly code optimizer)

`c.opt` [*`inpath`*] [*`outpath`*]

C.opt reads stdin and writes stdout. *`inpath`* must be present if *`outpath`* is given. Since c.opt rearranges and changes code, comments and assembler directives may be rearranged.

C.6. c.asm (Assembler)

`c.asm` *`file`* [options]

C.asm reads *`file`* as assembly language input. Errors are written to stderr. Options are turned on with one '-' and negated with '--'. To turn listing on use -l. To turn listing off use --l. To turn conditionals off use --c.

Recognized options:

`-o=`*`path`*	Write relocatable output to path. Must be a disk file.

`-l`	Write listing to stdout. (default off)
`-c`	List conditional assembly lines. (default on)
`-f`	Formfeed for top of form. (default off)
`-g`	List all code bytes generated. (default off)
`-x`	Suppress macro expansion listing. (default on)
`-e`	Print errors. (default on)
`-s`	Print symbol table. (default off)
`-d`*n*	Set lines per page to *n*. (default 66).
`-w`*n*	Set line width to *n*. (default 80).

C.7. c.link (Linker)

`c.link` [options] *`mainline subn`*... [options]

C.link turns c.asm output into executable form. All input files must contain relocatable object format (ROF) files. *`mainline`* specifies the base module from which to resolve external references. A mainline module is indicated by setting the type/lang value in the psect directive to non-zero. No other ROF can contain a mainline psect. The mainline and all subroutine files will appear in the final linked object module whether actually referenced or not.

For the C Compiler, cstart.r is the mainline module. It is the mainline module's job to perform the initialization of data and the relocation of any data-text and data-data references within the initialized data using the information in the object module supplied be c.link.

Recognized options:

`-o=`*`path`*	Linker object output file. Must be a disk file. The last element in *`path`* is used as the module name unless overridden by -n.
`-n=`*`name`*	Use *`name`* as object file name.
`-l=`*`path`*	Use *`path`* as library file. A library file consists of one or more merged assembly ROF files. Each psect in the file is checked to see if it resolves any unresolved references. If so, the module is included on the final output module, otherwise it is skipped. No mainline psects are allowed in a library file. Library files are searched on the order given on the command line.
`-E=`*`n`*	
`-e=`*`n`*	*`n`* is used for the edition number in the final output module. 1 is used is -e is not present.
`-M=`*`size`*	*`size`* indicates the number of pages (kbytes if size is followed by a K) of additional memory, c.link will allocate to the data area of the final object module. If no additional memory is given, c.link add up the total data stack requirements found in the psect of the modules in the input modules.
`-m`	Prints linkage map indicating base addresses of the psects in the final object module.
`-s`	Prints final addresses assigned to symbols in the final object module.
`-b=`*`ept`*	Link C functions to be callable by BASIC09. *`ept`* is the name of the function to be transferred to when BASIC09 executes the RUN command.

`-t`	Allows static data to appear in a BASIC09 callable module. It is assumed the C function called and the calling BASIC09 program have provided a sufficiently large static storage data area pointed to by the Y register.

Appendix D. Interfacing to Basic09

The object code generated by the Microware C Compiler can be made callable from the BASIC09 "RUN" statement. Certain portions of a BASIC09 program written in C can have a dramatic effect on execution speed. To effectively utilize this feature, one must be familiar with both C and BASIC09 internal data representation and procedure calling protocol.

C type "int" and BASIC09 type "INTEGER" are identical; both are two byte two's complement integers. C type "char" and BASIC09 type "BYTE" and "BOOLEAN" are also identical. Keep in mind that C will sign-extend characters for comparisons yielding the range -128 to 127.

BASIC09 strings are terminated by 0xff (255). C strings are terminated by 0x00 (0). If the BASIC09 string is of maximum length, the terminator is not present. Therefore, string length as well as terminator checks must be performed on BASIC09 strings when processing them with C functions.

The floating point format used by C and BASIC09 are not directly compatible. Since both use a binary floating point format it is possible to convert BASIC09 reals to C doubles and vice-versa.

Multi-dimensional arrays are stored by BASIC09 in a different manner than C. Multi-dimensional arrays are stored by BASIC09 in a column-wise manner; C stores them row-wise. Consider the following example:

BASIC09 matrix: `DIM array(5,3):INTEGER`

The elements in consecutive memory locations (read left to right, line by line) are stored as:

```
(1,1),(2,1),(3,1),(4,1),(5,1)
(1,2),(2,2),(3,2),(4,2),(5,2)
(1,3),(2,3),(3,3),(4,3),(5,3)
```

C matrix: `int array[5][3];`

```
(1,1),(1,2),(1,3)
(2,1),(2,2),(2,3)
(3,1),(3,2),(3,3)
(4,1),(4,2),(4,3)
(5,1),(5,2),(5,3)
```

Therefore to access BASIC09 matrix elements in C, the subscripts must be transposed. To access element array(4,2) in BASIC09 use array[2][4] in C.

The details on interfacing BASIC09 to C are best described by example. The remainder of this appendix is a mini tutorial demonstrating the process starting with simple examples and working up to more complex ones.

D.1. Example 1 - Simple Integer Aritmetic Case

This first example illustrates a simple case. Write a C function to add an integer value to three integer variables.

```
build bt1.c
? addints(cnt,value,s1,arg1,s2,arg2,s2,arg3,s4)
? int *value,*arg1,*arg2,*arg3;
? {
?     *arg1 += *value;
?     *arg2 += *value;
?     *arg3 += *value;
```

```
? }
?
```

That's the C function. The name of the function is "addints". The name is information for C and c.link; BASIC09 will not know anything about the name. Page 9-13 of the BASIC09 Reference manual describes how BASIC09 passes parameters to machine language modules. Since BASIC09 and C pass parameters in a similar fashion, it is easy to access BASIC09 values. The first parameter on the BASIC09 stack is a two-byte count of the number of following parameter pairs. Each pair consists of an address and size of value. For most C functions, the parameter count and pair size is not used. The address, however, is the useful piece of information. The address is declared in the C function to always be a "pointer to..." type. BASIC09 always passes addresses to procedures, even for constant values. The arguments `cnt`, `s1`, `s2`, `s3` and `s4` are just place holders to indicate the presence of the parameter count and argument sizes on the stack. These can be used to check validity of the passed arguments if desired.

The line "`int *value,*arg1,*arg2,*arg3`" declares the parameters (in this case all "pointers to int"), so the compiler will generate the correct code to access the BASIC09 values. The remaining lines increment each arg by the passed value. Notice that a simple arithmetic operation is performed here (addition), so C will not have to call a library function to do the operation.

To compile this function, the following C compiler command line is used:

```
cc2 bt1.c -rs
```

Cc2 uses the Level-Two compiler. Replace **cc2** with **cc1** if you are using the Level-One compiler. The `-r` option causes the compiler to leave `bt1.r` as output, ready to be linked. The `-s` option suppresses the call to the stack-checking function. Since we will be making a module for BASIC09, `cstart.r` will not be used. Therefore, no initialized data, static data, or stack checking is allowed. More on this later.

The `bt1.r` file must now be converted to a loadable module that BASIC09 can link to by using a special linking technique as follows:

```
c.link bt1.r -b=addints -o=addints
```

This command tells the linker to read `bt1.r` as input. The option "`-b=addints`" tells the linker to make the output file a module that BASIC09 can link to and that the function "addints" is to be the entrypoint in the module. You may give many input files to **c.link** in this mode. It resolves references in the normal fashion. The name given to the "`-b=`" option indicates which of the functions is to be entered directly by the BASIC09 RUN command. The option "`-o=addints`" says what the name of the output file is to be, in this case "addints". This name should be the name used in the BASIC09 RUN command to call the C procedure. The name given in "`-o=`" option is the name of the procedure to RUN. The "`-b=`" option is merely information to the linker so it can fill in the correct module entrypoint offset.

Enter the following BASIC09 program:

```
PROCEDURE btest
DIM i,j,k:INTEGER
i=1
j=132
k=-1033
RUN addints(4,i,j,k)
PRINT i,j,k
END
```

When this procedure is RUN, it should print:

```
5       136     -1029
```

indicating that our C function worked!

D.2. Example 2 - More Complex Integer Aritmetic Case

The next example shows how static memeory can be used. Take the C function from the previous example and modify it to add the number of times it has been entered to the increment:

```
build bt2.c
? static int entcnt;
?
? addints(cnt,cmem,cmemsiz,value,s1,arg1,s2,arg2,s2,arg3,s4)
? char *cmem;
? int *value,*arg1,*arg2,*arg3;
? {
? #asm
?  ldy 6,s base of static area
? #endasm
?     int j = *value + entcnt++;
?
?     *arg1 += j;
?     *arg2 += j;
?     *arg3 += j;
? }
?
```

This example differs from the first in a number of ways. The line "`static int entcnt`" defines an integer value name `entcnt` global to `bt2.c`. The parameter *`cmem`* and the line "`char *cmen`" indicate a character array. The array will be used in the C function for global/static storage. C accesses non-auto and non-register variables indexed off the Y register. `cstart.r` normally takes care of setting this up. Since `cstart.r` will not be used for this BASIC09-callable function, we have to take measures to make sure the Y register points to a valid and sufficiently large area of memory. The line "`ldy 6,s`" is assembly language code embedded in C source that loads the Y register with the first parameter passed by BASIC09. If the first parameter in the BASIC09 RUN statement is an array, and the "`ldy 6,s`" is placed *immediately* after the "{" opening the function body, the offset will always be "`6,s`". Note the line beginning "`int j = ...`". This line uses an initializer which, in this case, is allowed because `j` is of class "auto". No classes but "auto" and "register" can be initialized in BASIC09-callable C functions.

To compile this function, the following C compiler command line is used:

```
cc2 bt2.c -rs
```

Again, the `-r` option leaves `bt2.r` as output and the `-s` option suppresses stack checking.

Normally, the linker considers it to be an error if the "`-b=`" option appears and the final linked module requires a data memory allocation. In our case here, we require a data memory allocation and we will provide the code to make sure everything is set up correctly. The "`-t`" linker option causes the linker to print the total data memory requirement so we can allow for it rather than complaining about it. Our linker command line is:

```
c.link bt2.r -o=addints -b=addints -r
```

The linker will respond with "BASIC09 static data size is 2 bytes". We must make sure `cmem` points to at least 2 bytes of memory. The memory should be zeroed to conform to C specifications.

Enter the following BASIC09 program:

```
PROCEDURE btest
DIM i,j,k,n;INTEGER
DIM cmem(10):INTEGER
FOR i=1 TO 10
     cmem(i)=0
NEXT i
FOR n=1 TO 5
     i=1
     j=132
     k=-1033
     RUN addints(cmem,4,i,j,k)
     PRINT i,j,k
NEXT n
END
```

This program is similar to the previous example. Our area for data memory is a 10-integer array (20 bytes) which is way more than the 2 bytes for this example. It is better to err on the generous side. Cmem is an integer array for convenience in initializing it to zero (per C data memory specifications). When the program is run, it calls addints 5 times with the same data values. Because addints add the number of times it was called to the value, the i,j,k values should be 4+number of times called. When run, the program prints:

```
     5       136     -1029
     6       137     -1028
     7       138     -1027
     8       139     -1026
     9       140     -1025
```

Works again!

D.3. Example 3 - Simple String Manipulation

This example shows how to access BASIC09 strings through C functions. For this example, write the C version of SUBSTR.

```
build bt3.c
? /* Find substring from BASIC09 string:
?         RUN findstr(A$,B$,findpos)
?    returns in fndpos the position in A$ that B$ was found or
?    0 if not found.  A$ and B$ must be strings, fndpos must be
?    INTEGER.
? */
? findstr(cnt,string,strcnt,srchstr,srchcnt,result);
? char *string,*srchstr;
? int strcnt, srchcnt, *result;
? {
?     *result = finder(string,strcnt,srchstr,srchcnt);
? }
?
? static finder(str,strlen,pat,patlen)
? char *str,*pat;
? int strlen,patlen;
? {
?     int i;
?     for(i=1;strlen-- > 0 && *str!=0xff; ++i)
?         if(smatch(str++,pat,patlen))
```

```
?             return i;
? }
?
? static smatch(str,pat,patlen)
? register char *str,*pat;
? int patlen;
? {
?     while(patlen-- > 0 && *pat != 0xff)
?         if(*str++ != *pat++)
?             return 0;
?     return 1;
? }
?
```

Compile this program:

```
cc2 bt3.c -rs
```

And link it:

```
c.link bt3.r -o=findstr -b=findstr
```

The BASIC09 test program is:

```
PROCEDURE btest
DIM a,b:STRING[20]
DIM matchpos:INTEGER
LOOP
INPUT "String ",a
INPUT "Match  ",b
RUN findstr(a,b,matchpos)
PRINT "Matched at position ",matchpos
ENDLOOP
```

When this program is run, it should print the position where the matched string was found in the source string.

D.4. Example 4 - Quicksort

The next example programs demonstrate how one might implement a quicksort written in C to sort some BASIC09 data.

C integer quicksort program:

```
#define swap(a,b) { int t; t=a; a=b; b=t; }

/* qsort to be called by BASIC09:
     dim d(100):INTEGER  any size INTEGER array
     run cqsort(d,100)   calling qsort.
*/

qsort(argcnt,iarray,iasize,icount,icsiz)
int  argcnt,     /* BASIC09 argument count */
     iarrary[],  /* Pointer to BASIC09 integer array */
     iasize,     /* and it's size */
     *icount,    /* Pointer to BASIC09 (sort count) */
     icsiz;      /* Size of integer */
```

```
{
     sort(iarray,0,*icount);  /* initial qsort partition */
}

/* standard quicksort algorithm from Horowitz-Sahni */
static sort(a,m,n)
register int *a,m,n;
{
     register i,j,x;

     if(m < n) {
          i = m;
   j = n + 1;
   x = a[m];
   for(;;) {
        do i += 1; while(a[i] < x);  /* left partition */
        do j -= 1; while(a[j] > x);  /* right partition */
        if(i < j)
             swap(a[i],a[j])          /* swap */
      else break;
   }
   swap(a[m],a[j]);
   sort(a,m,j-1);                     /* sort left */
   sort(a,j+1,n);                     /* sort right */
     }
}
```

The BASIC09 program is:

```
PROCEDURE sorter
DIM i,n,d(1000):INTEGER
n=1000
i=RND(-(PI))
FOR i=1 to n
d(i):=INT(RND(1000))
NEXT i
PRINT "Before:"
RUN prin(1,n,d)
RUN qsortb(d,n)
PRINT "After:"
RUN prin(1,n,d)
END

PROCEDURE prin
PARAM n,m,d(1000):INTEGER
DIM i:INTEGER
FOR i=n TO m
PRINT d(i); " ";
NEXT i
PRINT
END
```

C string quicksort program:

```
/* qsort to be called by BASIC09:
     dim cmemory:STRING[10] This should be at least as large as
                            the linker says the data size should
```

```
                              be.
     dim d(100):INTERGER      Any size INTEGER array.

     run cqsort(cmemory,d,100) calling qsort. Note that the pro-
                              cedure name run in the linked OS-9
                              subroutine module. The module name
                              need not be the name of the C func-
                              tion.
*/

int maxstr;    /* string maximum length */

static strbcmp(str1,str2)            /* basic09 string compare */
register char *str1,*str2;
{
     int maxlen;

     for (maxlen = maxstr; *str1 == *str2 ;++str1)
          if (maxlen-- >0 || *str2++ == 0xff)
               return 0;
     return (*str1 - *str2);
}

cssort(argcnt,stor,storsiz,iaarray,iasize,elemlen,elsiz,
            icount,icsiz)
int argcnt;          /* BASIC09 argument count */
char *stor;         /* Pointer to string (C data storage) */
char iarray[];      /* Pointer to BASIC09 integer array */
int  iasize,          /* and it's size */
     *elemlen,        /* Pointer integer value (string length) */
     elsiz,           /* Size of integer */
     *icount,         /* Pointer to integer (sort count) */
     icsiz;           /* Size of integer */
{
/* The following assembly code loads Y with the first
   arg provided by BASIC09.  This code MUST be the first code
   in the function after the declarations.  This code assumes the
   address of the data area is the first parameter in the BASIC09
   RUN command. */
#asm
 ldy 6,s get addr for C storage
#endasm

/* Use the C library qsort function to do the sort. Our
   own BASIC09 string compare function will compare the strings.
*/

     qsort(iarray,*icount,maxstr=*elemlen,strbcmp);
}

/* define stuff cstart.r normally defines */
#asm
_stkcheck:
 rts dummy stack check function

 vsect
errno: rmb 2 C function system error number
_flacc: rmb 8 C library float/long accumulator
```

```
 endsect
#endasm
```

The BASIC09 calling programs: (words file contains strings to sort)

```
PROCEDURE ssorter
DIM a(200):STRING[20]
DIM cmemory:STRING[20]
DIM i,n:INTEGER
DIM path:INTEGER
OPEN #path,"words":READ

n=100
FOR i=1 to n
INPUT #path,a(i)
NEXT i
CLOSE #path
RUN prin(a,n)
RUN cssort(cmemory,a,20,n)
RUN prin(a,n)
END

PROCEDURE prin
PARAM a(100):STRING[20]; n:INTEGER
DIM i:INTEGER
FOR i=1 TO n
PRINT i; " "; a(i)
NEXT i
PRINT i
END
```

D.5. Example 5 - Floating Point

The next example shows how to access BASIC09 reals from C functions:

```
flmult(cnt,cmemory,cmemsiz,realarg,realsize)
int cnt;            /* number of arguments */
char *cmemory;      /* pointer to some memory for C use */
double *realarg;    /* pointer to real */
{
#asm
 ldy 6,s get static memory address
#endasm

     double number;

     getbreal(&number,realarg);      /* get the BASIC09 real */
     number *= 2.;                   /* number times two*/
     putbreal(realarg,&number);      /* give back to BASIC09 */

}

/* getreal(creal,breal)
     get a 5-byte real from BASIC09 format to C format */

getbreal(creal,breal)
double *creal,*breal;
```

```
{
     register char *cr,*br;   /* setup some char pointers */

     cr = creal;
     br = breal;
#asm
*  At this point U reg contains address of C double
*                  0,s contains address of BASIC09 real
 ldx 0,s get address of B real

 clra clear the C double
 clrb
 std 0,u
 std 2,u
 std 4,u
 stb 6,u
 ldd 0,x
 beq g3 BASIC09 real is zero

 ldd 1,x get hi B mantissa
 and a #$7f clear place for sign
 std 0,u put hi C matissa
 ldd 3,x get lo B mantissa
 andb #$fe mask off sign
 std 2,u put lo C mantissa
 lda 4,x get B sign byte
 lsra shift out sign
 bcc g1
 lda 0,u get C sign byte
 ora #$80 turn on sign
 sta 0,u put C sign byte
g1 lda 0,x get B exponent
 suba #128 excess 128
 sta 7,u put C exponent
g3 clra clear carry
#endasm

}

/* putbreal(breal,creal)
     put C format double into a 5-byte real from BASIC09 */

putbreal(breal,creal)
double *breal,*creal;
{
     register char *cr,*br;   /* setup some pointers */

     cr = creal;
     br = breal;
#asm
*  At this point U reg contains address of C double
*                  0,s contains address of BASIC09 real
 ldx 0,s get address of B real

 lda 7,u get C exponent
 bne p0 not zero?
 clra clear the BASIC09
 clrb real
```

```
 std 0,x
 std 2,x
 std 4,x
 bra p3 and exit

p0 ldd 0,u get hi C mantissa
 ora #$80 this bit always on for normalized real
 std 1,x put hi B mantissa
 ldd 2,u get lo C mantissa
 std 3,x put lo B mantissa
 incb round mantissa
 bne p1
 inc 3,x
 bne p1
 inc 2,x
 bne p1
 inc 1,x
p1 andb #$fe turn off sign
 stb 4,x put B sign byte
 lda 0,u get C sign byte
 lsla shift out sign
 bcc p2 bra if positive
 orb #$01 turn on sign
 stb 4,x put B sign byte
p2 lda 7,u get C exponent
 adda #128 less 128
 sta 0,x put B exponent
p3 clra clear carry
#endasm
}


/* replace cstart.r definitions for BASIC09 */
#asm
_stkcheck:
_stkchec:
 rts

 vsect
_flacc: rmb 8
errno: rmb 2
 endsect
#endasm
```

BASIC09 calling program:

```
PROCEDURE btest
DIM a:REAL
DIM i:INTEGER
DIM cmemory:STRING[32]
a=1.
FOR i=1 TO 10
  RUN flmult(cmemory,a)
  PRINT a
NEXT i
END
```

D.6. Example 6 - Matrix Elements

The last program is an example of accessing BASIC09 matrix elements. The C program:

```
matmult(cnt,cmemory,cmemsiz,matxaddr,matxsize,scalar,scalsize)
char *cmemory;       /* pointer to some memory for C use */
int matxaddr[5][3];  /* pointer a double dim integer array */
int *scalar;         /* pointer to integer */
{
#asm
 ldy 6,s get static memory address
#endasm

     int i,j;

     for(i=0; i<5; ++i)
          for(j=1; j<3; ++j)
               matxaddr[j][i]  *=  *scalar;  /* multiply by value */
}
#asm
_stkcheck:
_stkchec:
 rts

 vsect
_flacc: rmb 8
errno: rmb 2
 endsect
#endasm
```

BASIC09 calling program:

```
PROCEDURE btest
DIM im(5,3):INTEGER
DIM i,j:INTEGER
DIM cmem:STRING[32]
FOR i=1 TO 5
    FOR j=1 TO 3
        READ im(i,j)
    NEXT j
NEXT i
DATA 11,13,7,3,4,0,5,7,2,8,15,0,0,14,4
FOR i=1 TO 5
    PRINT im(i,1),im(i,2),im(i,3)
NEXT i
PRINT
RUN matmult(cmem,im,64)
FOR i=1 TO 5
    PRINT im(i,1),im(i,2),im(i,3)
NEXT i
END
```

Appendix E. Relocating Macro Assembler Reference

This appendix gives a summary of the operation of the "Relocating Macro Assembler" (named c.asm as distributed with the C Compiler). This appendix and the example assembly source files supplied with the C compiler should provide the basic information on how to use the "Relocating Macro Assembler" to create relocatable-object format files (ROF). It is further assumed that you are familiar with the 6809 instruction set and mnemonics. See the Microware Relocating Assembler Manual for a more detailed description. The main function of this appendix is to enable the reader to understand the output produced by c.asm. The Relocating Macro Assembler allows programs to be compiled separately and then linked together, and it also allows macros to be defined within programs.

Differences between the Relocating Macro Assembler (RMA) and the Microware Interactive Assembler (MIA):

RMA does not have an interactive mode. Only a disk file is allowed as input.

RMA output is an ROF file. The ROF file must be processed by the linker to produce an executable OS9 memory module. The layout of the ROF file is described later.

RMA has a number of new directives to control the placement of code and data in the executable module. Since RMA does not produce memory modules, the MIA directives "mod" and "emod" are not present. Instead, new directives PSECT and VSECT control the allocation of code and data areas by the linker.

RMA has no equivalent to the MIA "setdp" directive. Data (and DP) allocation is handled by the linker.

E.1. Symbolic Names

A symbolic name is valid if it consists of from one to nine uppercase or lowercase characters, decimal digits or the characters "$", "_", "." or "@". RMA does not fold lowercase letters to uppercase. The names "Hi.you" and "HI.YOU" are distinct names.

E.2. Label field

If a symbolic name in the label field of a source statement is followed by a “:” (colon), the name will be known *globally* (by all modules linked together). If no colon appears, the name will be known only in the PSECT in which it was defined. PSECT will be described later.

E.3. Undefined names

If a symbolic name is used in an expression and hasn't been defined, RMA assumes the name is external to the PSECT. RMA will record information about the reference so the linker can adjust the operand accordingly. External names cannot appear in operand expressions for assembler directives.

E.4. Listing format

```
00098 0032 59        +         rolb
00117 0045=17ffb8      label   lbsr  _dmove     Comment
^     ^   ^^         ^ ^       ^     ^          ^
|     |   ||         | |       |     |          Start of comment
```

```
|     |   ||         | |       |     Start of operand
|     |   ||         | |       Start of mnemonic
|     |   ||         | Start of label
|     |   ||         A "+" indicates a line generated by a macro
|     |   ||         expansion.
|     |   |Start of object code bytes.
|     |   An "=" here indicates that the operand contains an
|     |   external reference.
|     Location counter value
Line number.
```

E.5. Section Location Counters

Each section contains the following location counters:

PSECT instruction location counter

VSECT initialized direct page location counter
non-initialized direct page location counter
initialized data location counter
non-initialized data location counter

CSECT base offset counter

E.6. Section Directives

RMA contains 3 section directives. PSECT indicates to the linker the beginning of a relocatable-object-format file (ROF) and initializes the instruction and data location counters and assembles code into the ROF object code area. VSECT causes RMA to change to the data location counters and place any generated code into the appropiate ROF data area. CSECT initializes a base value for assigning offsets to symbols. The end of these sections is indicated by the ENDSECT directive.

The source statements placed in a particular section cause the linker to perform a function appropiate for the statement. Therefore, the mnemonics allowed within a section are restricted as follows:

- The mnemonics are allowed inside or outside any section: nam, opt, ttl, pag, spc, use, fail, rept, endr, ifeq, ifne, iflt, ifle, ifge, ifgt, ifpl, endc, else, equ, set, macro, endm, csect, and endsect.
- Within a CSECT: rmb
- Within a PSECT: any 6809 instruction mnemonic, fcc, fdb,fcs, fcb, rzb, vsect, endsect, os9 and end.
- Within a VSECT: rmb, fcc, fdb, fcs, fcb, rzb and endsect.

E.6.1. PSECT Directive

The main difference between PSECT and MOD is that MOD sets up information for OS-9 and PSECT sets up information for the linker (c.link in the C compiler).

PSECT {name,typelang,attrrev,edition,stacksize,entrypoint}

name	Up to 20 bytes (any printable character except space or comma) for a name to be used by the linker to identify this PSECT. This name need not be distinct from all other PSECTs linked together, but it helps to identify PSECTs the linker has a problem with if the names are different.

typelang	byte expression for the executable module type/language byte. If this PSECT is not a "mainline" (a module that has been designed to be forked to) module this byte must be zero.
attrrev	byte expression for executable module attribute/revision byte.
edition	byte expression for executable module edition byte.
stacksize	word expression estimating the amount of stack storage required by this psect. The linker totals this value in all PSECTs to appear in the executable module and adds this value to any data storage requirement for the entire program.
entrypoint	word expression entrypoint offset for this PSECT. If the PSECT is not a mainline odule, this should be set to zero.

PSECT must have either no operand list or an operand list containing a name and five expressions. If no operand list is provided, the PSECT name defaults to "program" and all other expressions to zero. The can only be one PSECT per assembly language file.

The PSECT directive initializes all counter orgs and marks the start of the program module. No VSECT data reservations or object code may appear before or after the PSECT/ENDSECT block.

Example:

```
psect myprog,Prgrm+Objct,Reent+1,Edit,0,progent
psect another_prog,0,0,0,0,0
```

E.6.2. VSECT Directive

VSECT {DP}

The VSECT directive causes RMA to change to the data location counters. If DP appears after VSECT, the direct page counters are used, otherwise the non-direct page data is used. The RMB directive within this section reserves the specified number of bytes in the appropiate uninitialized data section. The fcc, fdb, fcs, fcb and rzb (reserve zeroed bytes) directives place data into the appropiate initialized data section. If an operand for fdb or fcb contains an external reference, this information is placed in the external reference part of the ROF to be adjusted at link or execution time. ENDSECT marks the end of the VSECT block. Any number of VSECT blocks can appear within a PSECT. Note, however, that the data location counters maintain their values between one VSECT block and the next. Since the linker handles the actual data allocation, there is no facility provided to adjust the data location counters.

E.6.3. CSECT Directive

CSECT {expression}

The CSECT directive provides a means for assigning consecutive offsets to labels without resorting to EQUs. If the expression is present, the CSECT base counter is set to that value, otherwise it is set to zero.

E.6.4. RZB statement

RZB <expression>

The reserve zeroed bytes pseudo-instruction generates sequences of zero bytes in the code or initialized data sections, the number of which is specified by the expression.

E.7. Comparison Between Assembly Programs for the Microware Interactive Assembler and the Relocating Macro Assembler

The following two program examples simply fork a BASIC09. The purpose of the examples are to show some of the differences in the new relocating assembler. The differences are apparent.

```
* this program forks a basic09
          ifp1
          use ..../defs/os9defs.a
          endc

PRGRM     equ $10
OBJCT     equ $01

stk       equ 200
          psect rmatest,$11,$81,0,stk,entry

name      fcs /basic09/
prm       fcb $D
prmsize   equ *-prm

entry     leax name,pcr
          leau prm,pcr
          ldy #prmsize
          lda #PRGRM+OBJCT
          clrb
          os9 F$FORK
          os9 F$WAIT
          os9 F$EXIT
          endsect
```

E.7.1. Macro Interactive Assembler Source

```
          ifp1
          use defsfile
          endc

          mod siz,prnam,type,revs,start,size
prnam     fcs /testshell/
type      set prgm+objct
revs      set reent+1

          rmb 250
          rmb 200
name      fcs /basic09/
prm       fcb $D
prmsize   equ *-prm
size      equ .
start     equ *
          leax name,pcr
          leau prm,pcr
          ldy #prmsize
          lda #PRGRM+OBJCT
```

```
          clrb
          os9 F$FORK
          os9 F$WAIT
          os9 F$EXIT
          emod
siz       equ
```

E.8. Introduction to Macros

In programming applications it is frequently necessary to use a repeated sequence or pattern of instructions in many different places in a program. For example, suppose a group of program statements creates a file a number of times throughout the program. The code might look like the following statements:

```
leax name,pcr
lda $02
ldb $03
os9 I$CREATE
```

The sequence must be replicated each time that a new file is created. A macro assembler eliminates the need for coding duplicate statement patterns by allowing the programmer to define macro instructions that are equivalent to longer code sequences.

When a macro is called, it is the same as calling a subrouting to perform a defined function. A macro produces in-line code that is inserted into the normal flow of the program beginning at the location of the macro call. The statements that may be generated by a macro are generally unrestricted, and the statements may contain substitutable arguments.

E.9. Operations

E.9.1. Macro Definition

A macro definition consists of three sections:

```
<Label> MACRO    /* macro header */
        .        /* <Label> is the name of the macro */
        .
        body     /* macro body */
        .
        .
        ENDM     /* macro terminator */
```

1. The macro header - assigns a name to the macro
2. The body - contains the macro statements
3. The terminator - indicates the end of the macro

A macro can have up to nine arguments (\1 to \9) in the operand fields. The arguments are used to refer to symbols, registers, etc.

The following macro below could represent the file creation pattern:

```
CREATE MACRO
       leax \1,pcr
       lda $\2
       ldb $\3
```

```
        os9 I$CREATE
        ENDM
```

Calls can be made to create files with different names, access modes, and attributes as follows:

```
CREATE name2,02,03
CREATE name3,01,02
```

The above macro calls will produce the following in-line code:

```
        leax name2, pcr
        lda $02
        ldb $03
        os9 I$CREATE

        leax name3, pcr
        lda $01
        ldb $02
        os9 I$CREATE
```

If an argument has multiple parts, for example if d1,d2 is to be passed to the macro called frud, it must be passed in double quotes. For example:

```
frud "0,s","2,s"
```

If frud looks like the following macro:

```
frud MACRO
\@   leau \1
     ldd  \2
     beq  \@
     ENDM
```

The previous call to frud would expand the macro as follows:

```
@xxx  leau 0,s
      ldd  2,s
      beq  @xxx
```

Where "\@" is a label, and "xxx" would be replaced by a three digit number.

An argument may be declared null by leaving it blank in the macro call. For example, if the macro instruction was "ldd \1ZZ\2", then the call to the macro with arguments AA,BB would expand the instruction to "ldd AAZZBB", and the call with argument ,BB will expand it to "ldd ZZBB".

E.9.2. Nested Macro Calls

Macro calls may be nested, that is, the body of a macro definition may contain a call to another macro. For example, the macro prepw could be defined as follows:

```
prepw MACRO
      lda \1
      getw
      ENDM
```

Getw is a macro call. The code to getw is substituted in-line at expansion time. However, the definition of a new macro within another is not permitted. Macro calls may be nested up to eight deep.

E.9.3. Labels

Sometimes it is necessary to use labels within a macro. Labels are specified by “\@”. Each time the macro is called, a unique label will be generated to avoid multiple definition errors. Within the expanded code “\@” will take on the form “@xxx”, where xxx will be a decimal number between 000 to 999.

More than one label may be specified in a macro by the addition of an extra character(s). For example, if two different labels are required in a macro, they can be specified by "\@A" and "\@B". In the first expansion of te macro, the labels would be "@001A" and "@001B", and in the second expansion they would be "@002A" and "@002B". The extra characters may be appended before the "\" or after the "@".

E.9.4. Additional Pseudo-Instructions

\n	will return the number of arguments passed to the macro.
\L<num>	will return the length of the ith argument that is specified by <num>.
FAIL	Causes an error to be generated.
REPT <num>	will repeat an instruction or group of instructions <num> times. ENDR terminates REPT.

Colophon

This book is scanned from an OS-9 C-compiler + manual for the Dragon 64. The book was published by Dragon Data Ltd., and distributed by H. C. Andersen Computer A/S in Copenhagen.

As sold, the manual was in the form of a spiral-bound A5 book with a grey cover. The typeface was a monotype serif font as was common on typewriters in those days.

The manual has many references to Appendix A (The C reference) of the Kernighan & Ritchie C Programming Language book. Since there has been an evolution of the C language since 1983 and the original edition of K & R is very difficult to find, I have downloaded the C reference from Dennis Ritchie's website and included it as appendix A in this manual.