C Compiler User's Guide

From nitros9

Contents

- 8.7.4 Additional Comments About Macros

The C Compiler System

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 the Bel 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 if 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 nonstandardization 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 corrected executed.

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). A copy of this book, which serves as the language reference manual, is included with each software package.

Although 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.

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 (#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 newline '\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.

Enhancements and Extensions

The "Direct" Storage Class

The 6809 microprocessor instruction 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 should be, and it need not concern the program, 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 keyword 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 classes 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 of 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 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.

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 an **ENDSECT** directive at the end to leave the state correct for following compiler generated code.

Control Character Escape Sequences

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

```
linefeed (LF): \l (lowercase 'ell')
```

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

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

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

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

Implementation-Dependent Characteristics

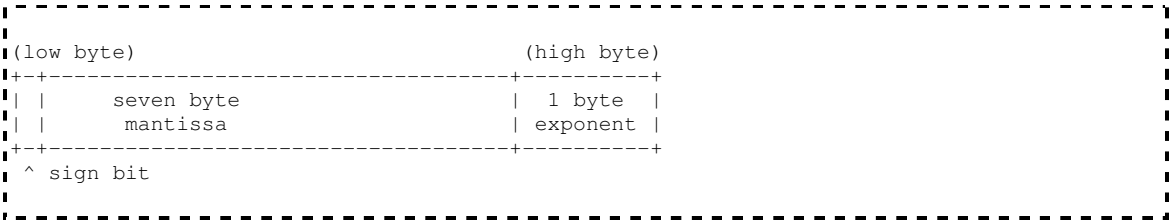
K&R frequently refer to characteristics of the C language whose exact operations depend on the architecture and instruction set of the computer actually used. This section contains specific information regarding this version of C for the 6809 processor.

Data Representation and Storage

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 **char** is converted to **int** 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:



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.

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 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 in a complex arithmetic expression, there is no savings. The "U" register is assigned to register variables.

IMPORTANT NOTE: Upper- and lowercase letters cannot be mixed as in Basic09. For example, Prog.c and prog.c are distinct names. Since the Color Computer is usually used in uppercase only, it is necessary to enter the following commands to use upper- and lowercase: **TMODE -UPC** and CLEAR<0>.

Access to Command Line Parameters

The standard C arguments **argc** and **argv** are available to **main** as described in K&R page 110. The startup 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.

System Calls and the Standard Library

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.

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 te 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.

Run-Time Arithmetic Error Handling

K&R leave the treatment of various arithmetic errors open, merely saying that it is machine dependent. This implementation deals 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 or 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, this will cause termination of the 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.

Achieving maximum Program Performance

Programming Considerations

Because the 6809 is an 8/16 bit microprocessor, the compiler can generate efficient code for 8 and 16 bit objects (**char**, **int**, etc.). However, code for 32 and 64 bit values (**long**, **float**, **double**) 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.

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 compilation.

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.

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 direction (CMDS).

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

OS-9 Level II Systems:

cc2 compiler executive program
c.prep macro pre-processor
c.comp compiler proper
c.asm relocating assembler
c.link linkage editor

In addition a file called **clib.l** contains the standard library, math functions, and system 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 drive /D1. The DEFS directory must also be on /D1.

If, when specifying **#include** files for the processor to read in, the programmer uses angle brackets, < and >, instead of parentheses, the file will be sought starting at the DEFS directory.

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 include files) should be free.

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

Running the Compiler

There 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 uses a single pass version. Both versions of the compiler work 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 {file}
```

One file at a time can be compiled, or a number of files may be compiled together. The compiler manages the compilation through up to four stages: pre-processor, compilation to assembler code, assembly to relocatable module, 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 show below. Any of the above file types may be mixed on the command line.

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 named 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 int he 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 called **prg1.r** in the current data directory, and **prg2.r** in /d0/fred.

```
CC -E=3 FNAME.C -F=PROG
```

compiles the file called **FNAME.C** into an executable object file named **PROG** and sets the module revision level to 3.

```
CC PROG.C -DIDENTIFIER=VALUE
```

compiles the program with a definition identifier being passed to the compiler. The definition being passed is used within the source to control compilation via **#ifdef/#ifndef** functions.

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*	Sets 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 invocation 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* Instructs 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 Outputs 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* 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** or **#ifndef** pre-processor directives. If the *identifier* is used as a macro for expansion by the pre-processor, 1 will the expanded value unless the form **-d *identifier=string*** is used in which case the expansion will be *string*.

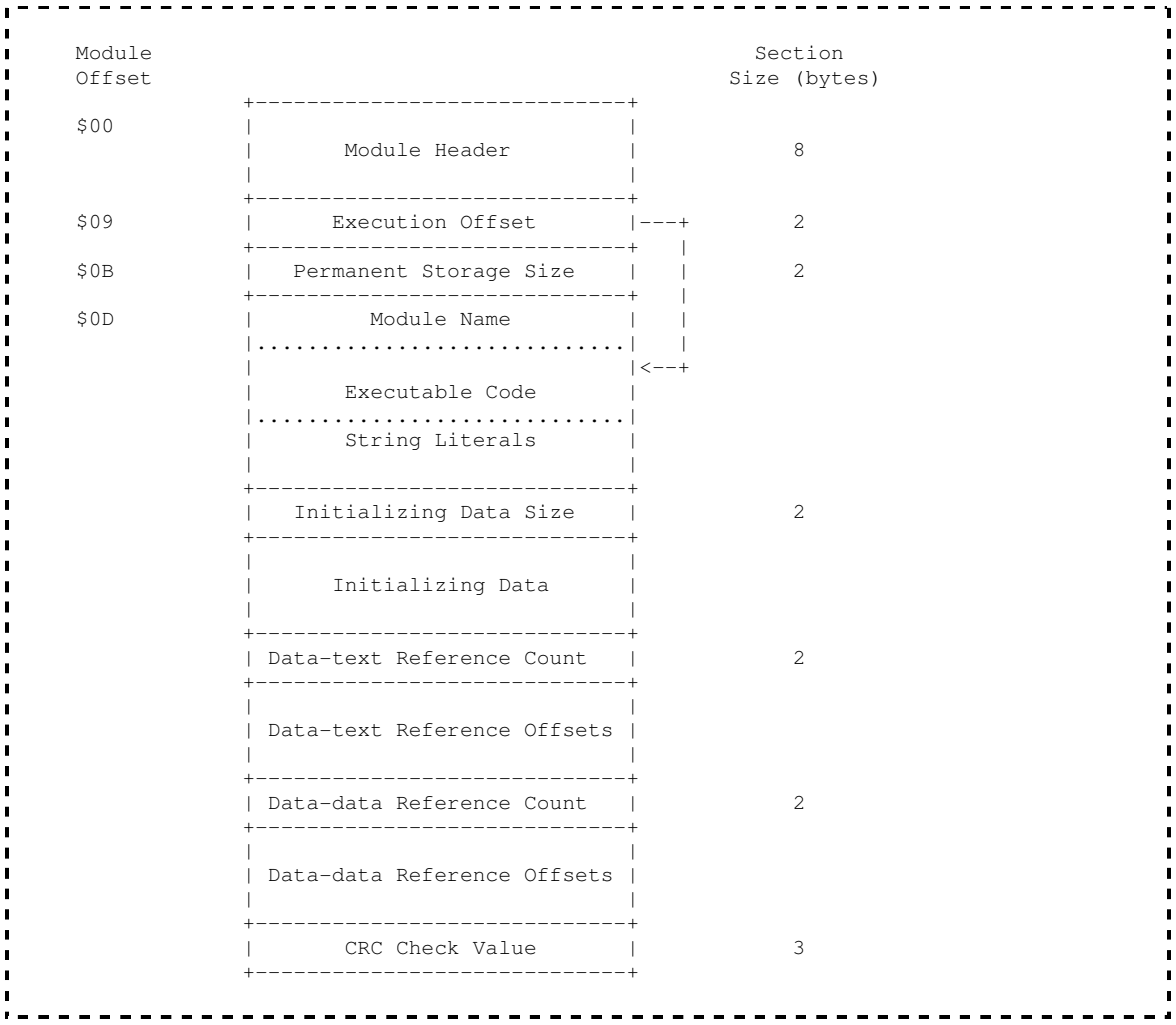
Command Line and Option Flag Examples

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	prog1.r, prg2.r, prg3.r, output
cc prg1.c prg2.a prg3.r	compile prg1.c, assemble prg2.a and combine all into an 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

Characteristics of Compiled Programs

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 the following pages.



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).

Execution Offset

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

Storage Size

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

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 overriden by the **-E=** option in cc.

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.

Executable Code

The machine code instructions of the program.

String Literals

Quoted strings 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 ncessary 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.

Initialization 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.

Data References

No absolute addresses are known at compile time under OS-9, so where there are pointer values in the initializing 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.

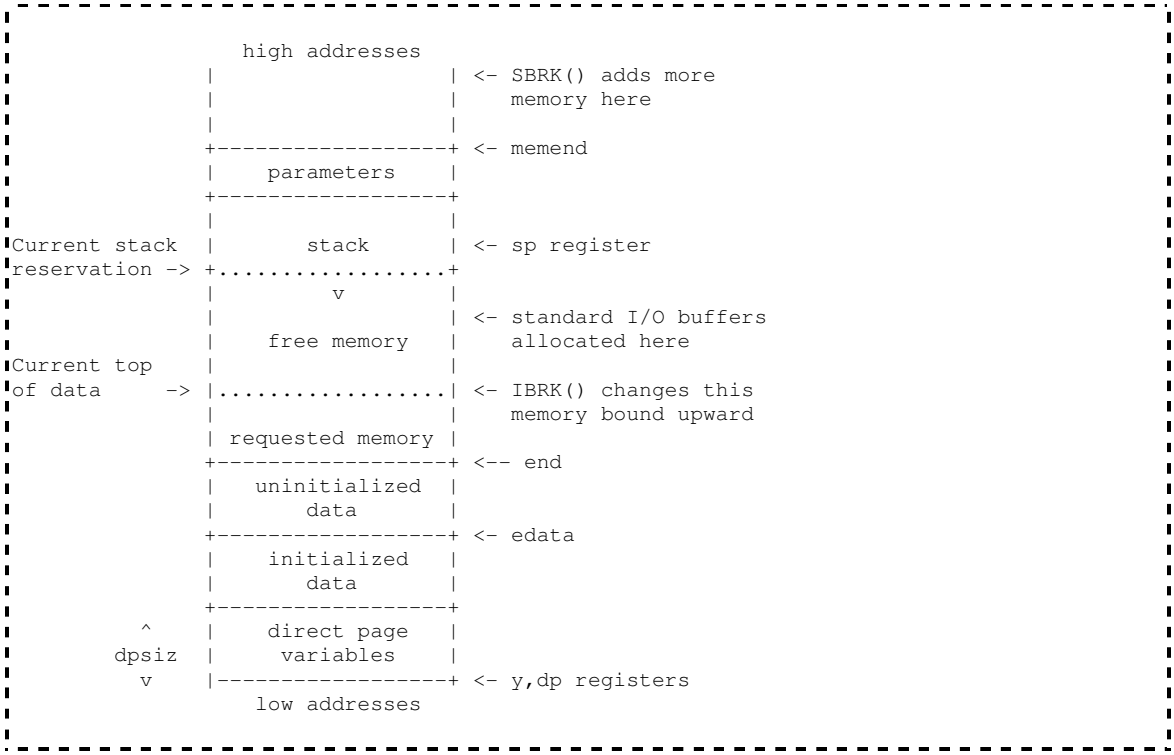
The startup routine will first copy all the entries in the initializing data section into their allotted places in the DATA section. then it will scan the data-text reference table for the offsets of values that need to have the addresses of the base of the TEXT section added to them. Among thee will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

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.

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:



The overall size of the 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 parameter 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 being this lower, 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 language 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 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 each reason **sbrk()** will return -1.

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. This does not apply to the lower level service request I/O functions such as **open()**, **read()**, or **write()** nor to **stderr** which is always unbuffered, but it does apply to both **stdin** and **stdout** (see the Standard Library documentation).

A good method for getting the feel of how much memory is needed by your program is to allow the linker to set the memory size to its usually conservative default 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 return -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.

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 system 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 program 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.

---

abort - Stop the program and produce a core dump

Usage

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.

abs - Absolute value

Usage

```
int 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.

access - Give file accessibility

Usage

```
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 appropriate error indication, if a value of -1 is returned, may be found in *errno*.

chain - Load and execute a new program

Usage

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

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 program to check the existence and access permissions of the module before calling chain. Permissions may be checked by using **modlink()** or **modload()** followed by **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.

chdir, chxdir - Change directory

Usage

```
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**.

chmod - Change access permissions of a file

Usage

```
#include <modes.h>
chmod(fname, perm)
char *fname;
```

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_EXEC    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 NULL (0). A -1 is returned if the caller is not entitled to change permissions or **fname** cannot be found.

See Also

OS-9 command **attr**

chown - Change the ownership of a file

Usage

```
chown(fname, ownerid)
char *fname;
```

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 on error.

close - Close a file

Usage

close(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 file automatically, but it is necessary to close files where multiple files are opened by the task, and it is desired to reuse path numbers to avoid going over the system or process path number limit.

See Also

creat(), **open()**, **dup()**

crc - Compute a cyclic redundancy count

Usage

```
crc(start, count, accum)
char *start, 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 **acum** 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.

creat - Create a new file

Usage

```
#include <modes.h>
creat(fname, perm)
char *fname;
```

Assembler Equivalent

os9 I$Creat

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:

```
/* permissions */
#define S_IPRM    0xFF  /* mask for permission bits */
#define S_IREAD   0x01  /* owner read */
#define S_IWRITE  0x02  /* owner write */
#define S_EXEC    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 search, if permission to write is denied, or if the file exists and is a directory.

See Also

write(), **close()**, **chmod()**

defdrive - Get default system drive

Usage

```
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.

dup - Duplicate an open path number

Usage

dup(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 if the call fails because there are too many files open or the path number is invalid.

See Also

open(), **creat()**, **close()**

exit, _exit - Task termination

Usage

exit(status)
_exit(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 to **exit(0)**.

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

See Also

wait()

getpid - Get the task id

Usage

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()**).

See Also

os9fork(), **mktemp()**

getstat - Get file status

Usage

```
#include <sgstat.h>
getstat(code, filenum, buffer) /* code 0 */
char *buffer;
getstat(code, filenum) /* codes 1 and 6 */
getstat(code, filenum, size) /* code 2 */
long *size;
getstat(code, filenum, pos) /* code 5 */
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 size 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, the actual error is returned in **errno**.

getuid - Return user id

Usage

getuid()

Assembler Equivalent

os9 F$ID

Description

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

intercept - Set function for interrupt processing

Usage

intercept(func)
int (*func)(); /* i.e. "func" is a pointer to a function returning an int */

Assembler Equivalent

os9 F$Icpt

Description

intercept() instructs OS-9 to pass control to **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()

kill - Send an interrupt to a task

Usage

```
#include <signal.h>
kill(tid, 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 system manager.

See Also

signal, OS-9 shell command kill

lseek - Position a file

Usage

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

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:

```
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

open(), creat(), fseek()

mknod - Create a directory

Usage

```
#include <modes.h>
mknod(fname, desc)
char *fname;
```

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

modload, modlink - Return a pointer to a module structure

Usage

```
#include <module.h>
mod_exec *modlink(modname, type, language)
char *modname;
mod_exec *modload(filename, type, language)
char *filename;
```

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()

munlink - Unlink a module

Usage

```
#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()

_os9 - System call interface from C programs

Usage

```
#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 defined 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 = %1x\n", (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=%04x\n", r->rg_x);
        printf(" y=%04x\n", r->rg_y);
        printf(" u=%04x\n", r->rg_u);
    }
```

open - Open a file for read/write access

Usage

```
open(fname, mode)
char *fname;
```

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 are 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 read and 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()

os9fork - Create a process

Usage

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

Assembler Equivalent

```
os9 F$Fork
```

Description

The action of **F$Fork** is described 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.

pause - Halt and wait for interrupt

Usage

pause()

Assembler Equivalent

os9 F$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**

prerr - Print error message

Usage

prerr(filnum, 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.

read, readln - Read from a file

Usage

```
read(pn, buffer, count)
char *buffer;
readln(pn, buffer, count)
char *buffer;
```

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 echoing 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 ridiculous **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()

sbrk, ibrk - Request additional working memory

Usage

```
char *sbrk(increase)
char *ibrk(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.

setptr - Set process priority

Usage

setpr(pid, priority)

Assembler Equivalent

os9 F$$SPrior

Description

setpr sets the process identified by **pid** (process id) to have a priority of **priority**. The lowest priority 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.

setime, getime - Set and get system time

```
#include <time.h>
setime(buffer)
getime(buffer)
struct sgtbuf *buffer; /* defined in time.h */
```

Assembler Equivalent

```
os9 F$STime
os9 F$GTime
```

Description

getime returns system time in **buffer**.

setime sets system time from **buffer**.

setuid - Set user id

Usage

setuid(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()

setstat - Set file status

Usage

```
#include <sgtstat.h>
setstat(code, filnum, buffer) /* code 0 */
char *buffer
setstat(code, filenum, size) /* code 2 */
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 options 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.

signal - Catch or ignore interrupts

Usage

#include <signal.h>
(*signal(interrupt, address))()
(*address)();
Which means **signal** returns a pointer to a function, **address** is a pointer to a funtion.

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 it 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);           /* write to temporary file */
close(pn);                          /* finished writing */
unlink(pn);                         /* delete it */
exit(0);                            /* normal exit */
```

The call to **signal()** will ensure that if a keyboard or quit signal is received then the funtion **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 as 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 in the source code before the call to **signal()**, or it should be predeclared.

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()**. However, since **signal()** and **intercept()** are mutually incompatible, calls to both of these must not appear in the same program. The link-loader will prevent the creation of an executable program in which both are called by aborting with an "entry name clash" error for **_sigint**.

See Also

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

stacksize, freemem - Obtain stack reservation size

Usage

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(), freemem(), Global variable memend and value end

_strass - Byte by byte copy

Usage

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

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 at **s1** regardless of the contents.

tsleep - Put process to sleep

Usage

tsleep(ticks)

Description

tsleep deativates the calling process for the 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.

unlink - Remove directory entry

Usage

```
unlink(fname)
char *fname;
```

Assembler Equivalent

os9 i$Delete

Description

unlink deletes the diretory 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 disk 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.

Errors

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

See Also

link(), Os-9 command kill

wait - Wait for task termination

Usage

```
wait(status)
int *status;
wait(0)
```

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 or 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 i there is no child to be waited for.

See Also

fork(), signal(), exit(), _exit()

write, writeln - Write to a file or device

Usage

```
write(pn, buffer, count)
char *buffer;
writeln(pn, buffer, count)
char *buffer;
```

Assembler Equivalent

```
os9 I$Write
os9 I$WritLn
```

Description

pn must be a value returned by **open**, **creat**, or **dup** or should be 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, if **count** is ridiculous, or on physical I/O error.

See Also

creat(), open(), dup()

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 pages 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 predeclaration in your program before calling it. For example, if you wish to use **atotf()**, you should predeclare 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 predeclared for you in the header. An example of this is **ftell()** which is predeclared in **stdio.h**. If you are in any doubt, read the header file.

atof, atoi, atol — ASCII to number conversions

Usage

```
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", an 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.

fflush, fclose - Fluse or close a file

Usage

```
#include <stdio.h>
fflush(fp)
FILE *fp;
fclose(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 calls **fflush** to clear out the buffer associated with **fp**, closes the file, and frees the butter 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 on writing to the file.

feof, ferror, clearerr, fileno - Return status information of files

Usage

```
#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 or 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()

findstr, findnstr - String search

Usage

```
findstr(pos, string, pattern)
char *string, *pattern;
findnstr(pos, string, pattern, size)
char *string, *pattern;
```

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 fist 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 + size** so it may continue path 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()

fopen - Open a file and return a file pointer

Usage

```
#include <stdio.h>
FILE *fopen(filename, action)
char *filename, *action;
FILE *freopen(filename, action, stream)
char *filename, *action;
FILE *stream;
FILE *fdopen(filedesc, action)
int filedes;
char *action;
```

Description

fopen returns a pointer to a file structure (file pointer) if the file named 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 the 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 name 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 end of 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 predefined in <stdio.h> as FILE, so that a user program may declare 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 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()**.

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 unsucessful.

See Also

flose(), System call open()

fread, fwrite - Read/write binary data

Usage

```
#include <stdio.h>
fread(ptr, size, number, fp)
FILE *fp;
fwrite(ptr, size, number, fp)
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 be **fp**. **number** is the number of items of size **size** reading them from memory starting at **ptr**.

Diagnostics

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

See Also

fopen(), getc(), putc(), printf(), System calls read(), write()

fseek, rewind, ftell - Position in a file or report current position

Usage

```
#include <stdio.h>
fseek(fp, offset, place)
FILE &fp;
long offset;
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 at **offset** bytes from the beginning of the file if **place** is O, the current position if 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()

getc, getchar - Return next character to be read from a file

Usage

```
#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> as **_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()

gets, fgets - Input a string

Usage

```
#include <stdio.h>
char *gets(s)
char *s;
char *fgets(s, n, fp)
char *s;
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 the '\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()

isalpha, isupper, islower, isdigit, isalnum, isspace, ispunct, isprint, iscntrl, isascii - Character classification

Usage

```
#include <ctype.h>
isalpha(c)
etc.
```

Description

These functions use table lookup to classify characters according to their ASCII value. The header file defines them as macros which means that they are implemented as fast, in-line 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 as follows:

isalpha c is a letter
isdigit c is a digit
isupper c is an uppercase letter
islower c is a lowercase 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 control nor alphanumeric
isprint c is printable (32 to 126)
isascii c is in the range -1 to 127

l3tol, ltol3 - Convert between long integers and 3-byte integers

Usage

```
l3tol(lp, cp, n)
long *lp;
char *cp;
ltol3(cp, lp, n)
long *lp;
char *cp;
```

Description

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

l3tol converts a vector of **n** three-byte integers pointed to by **cp** into a vector of long integers starting at **lp**.

ltol3 does the opposite.

longjmp, setjmp - Jump to another function

Usage

```
#include <setjmp.h>
setjmp(env)
jum_buf env;
longjmp(env, val)
jmp_buf env;
```

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 sane function. Control can only be transferred elsewhere by means of the function call, which, of course, returns to the oaller. 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.

malloc, free, calloc - Memory allocation

Usage

```
char *malloc(size)
unsigned size;
free(ptr)
char *ptr;
char *calloc(nel, elsize)
unsigned nel, elsize;
```

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 array, and **elsize** is the size of each element. **calloc** initializes the space to zero.

Diagnostics

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

mktemp - Create unique temporary file name

Usage

```
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()

printf, fprintf, sprintf - Formatted output

Usage

```
#include <stdio.h>
printf(control [,arg0[, arg1...]])
char *control;
fprintf(fp, control [, arg0[, arg1...]])
FILE *fp;
char *control;
sprintf(string, control [, arg0[, arg1...]])
string [];
char *control;
```

Description

These 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 indicating 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 apace, 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 tho 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 **plfinit()**, 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 uppercase.
u	The argument is an integer and the conversion is to an unsigned decimal in the range O 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 uppercase.
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 uppercase.
c	The argument is a character.
s	The argument is a pointer to a string. Characters from the string are printed up to a null character, or until 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.

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.

See Also

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

putc, putchar, putw - Put character or word in a file

Usage

```
#include <stdio.h>
char putc(ch, fp)
char ch;
FILE *fp;
char putchar(ch)
char *ch;
putw(n, fp)
FILE *fp;
```

Description

putc adds 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 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()

puts, fputs - Put a string on a file

Usage

```
#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.

qsort - Quick sort

Usage

```
qsort(base, n, size, compfunc)
char *base;
int (*compfunc)(); /* which means a pointer to a function returning an int */
```

Description

qsort implements the quick-sort algorithm for sorting an arbitrary array of items.

base is the address of the array of **n** items of size **size**. **compfunc** is 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.

scanf, fscanf, sscanf - Input string interpretation

Usage

```
#include <stdio.h>
fscanf(fp, control[, pointer ...])
FILE *fp;
char *control;
scanf(control[, pointer ...])
char *control;
sscanf(string, control[, pointer ...])
char *string, *control;
```

Description

These functions perform the complement of **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) Spaces, 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 option "*" 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 inappropriate for the conversion or when a specified field width is exhausted. When one field is finished, white-space characters are passed over until the next field is found.

The following conversion characters are recognized:

d	A decimal string is to be converted to an integer.
o	An octal string; the corresponding 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, or "\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 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 matching 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 which 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 argument should point to a double.

% — a match for "%" is sought; no conversion takes place.

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

Caveats

The returned count of matches/assignments does not include character matches and assignments 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

atio(), atof(), getc(), printf(), Kernighan & Ritchie pp 147-150

setbuf - Fix file buffer

Usage

```
#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 un—buffered 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()

sleep - Stop execution for a time

Usage

```
sleep(seconds)
unsigned seconds;
```

Description

The current task is stopped for the specified time. If **seconds** is zero, the task will sleep for one tick.

strcat, strncat, strcmp, strncmp, strcpy, strhcpy, strncpy, strlen, index, rindex - String functions

Usage

```
char *strcat(s1, s2)
har *s1, *s2;
char *strncat(s1, s2, n)
char *s1, *s2;
strcmp(s1, s2)
char *s1, *s2;
strncmp(s1, s2, n)
char *s1, *s2;
char *strcpy(s1, s2)
char *s1, *s2;
char *strhcpy(s1, s2)
char *s1, *s2;
char *strncpy(s1, s2, n)
char *s1, *s2;
strlen(s)
char *s;
char *index(s, ch)
char *s, ch;
char *rindex(s, ch)
char *s, ch;
```

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 grater 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, **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 strings with the 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()

system - Shell command request

Usage

```
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()

toupper, tolower - Character translation

Usage

```
#include <ctype.h>
int toupper(c)
int c;
int tolower(c)
int c;
int _toupper(c)
int c;
int _tolower(c)
int c;
```

Description

The functions **toupper** and **tolower** have as their domain the integers in the range -1 through 255. **toupper** converts lowercase to uppercase and **tolower** converts uppercase to lowercase. 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 lowercase, and the argument to **_tolower** must be uppercase. Arguments that are outside each macro's domain, such as passing a lowercase to **_tolower**, yield garbage results.

ungetc - Put character back on input

Usage

```
#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 one character must have been read from the file before a call to **ungetc**.

fseek() erases any pushback.

Diagnostics

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

See Also

getc(), fseek()

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 parentheses) to see for more specific information.

already a local variable	Variable has already been declared at the current block level. (8.1, 9.2)
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. (10.1)
argument storage	Function arguments may only be declared as storage class register. (10.1)
bad character	A character not in the C character set (probably a control character) was encountered in the sour file. (2)
both must be integral	>> and << operands cannot be **float** or **double**. (7.5)
break error	The break statement is allowed only inside a while, do, for, or switch blocks. (9.8)
can't take address	& operator is not allowed on a register variable. Operand must otherwise be an lvalue. (7.2)
cannot cast	Type result of cast cannot be function or array. (7.2, 8.7)
cannot evaluate size	Could not determine size from declaration or initializer. (8.6, 14.3)
cannot initialize	Storage class or type does not allow variable to be initialized. (8.6)
compiler trouble	Compiler detected something it couldn't handle. Try compiling the program again. If this error still occurs, contact Radio Shack.
condition needed	While, do, for, switch, and if statements require a condition expression. (9.3)
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**. (8.6)
constant overflow	Input numeric constant was too large for the implied or explicit type. (2.6, [PUP—11])
constant required	Variables are not allowed for array dimensions or cases. (8.3, 8.7, 9.7)
continue error	The continue statement is allowed only inside a while, do, or for block. (9.9)

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. (11, 11.1 11.2)
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 caused by mismatched braces. (10.1)
function type error	A function cannot be declared as returning an array, function, struct, or union. (8.4, 10.1)
function unfinished	End—of—file encountered before the end of function definition. (10.1)
identifier missing	Identifier name required here but none was found.
illegal declaration	Declarations are allowed only at the beginning of a block. (9.2)
label required	Label name required on **goto** statement. (9.11)
label undefined	Goto to label not defined in the current function. (9.12)
lvalue required	Left side of assignment must be able to be “stored into”. Array names, functions, structs, etc. are not lvalues. (7.1)
multiple defaults	Only one default statement is allowed in a switch block. (9.7)
multiple definition	Identifier name was declared more than once in the sane block level (9.2, 11.1)
must be integral	Type of object required here must be type mt, char, or pointer.
name clash	Struct—union member and tag names must be mutually distinct. (8.5)
name in a cast	Identifier name found in a cast. Only types are allowed. (7.2, 8.7)
named twice	Names in a function parameter list may appear only once. (10.1)
no ‘if’ for ‘else’	Else statement found with no matching if. This is typically caused by extra or missing braces and/or semicolons. (9.3)
no switch statement	Case statements can only appear within a switch block. (9.7)
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. (7.1)
not an argument	Name does not appear in the function parameter list. (10.1)
operand expected	Unary operators require one operand, binary operators two. This is

	typically caused by misplaced parenthesis, casts or operators. (7.1)
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 they 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 cast if required. (7.1)
pointer or integer required	A pointer (of any type) or integer is required to the left of the "—>" operator. (7.1)
pointer required	Pointer operand required with unary * operator. (7.1)
primary expected	Primary expression required here. (7.1)
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. (7.13)
**** 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 may only be used within functions. (8.1)
struct member mismatch	Identical member names in two different structures must have the same type and offset in both. (8.5)
struct member required	Identifier used with * and —> operators must be a structure member name. (7.1)
struct syntax	Brace, comma, etc. is missing in a struct declaration. (8.5)
struct or union inappropriate	Struct or union cannot be used in this context.
syntax error	Expression, declaration, or statement is incorrectly formed.
third expression missing	? must be followed by a : with expression. This error may be caused by unmatched parenthesis or other errors in the expression. (7.13)
too long	Too many characters provided in a string initializing a character array. (8.6)
too many brackets	Unmatched or unexpected brackets encountered processing an initializer. (8.6)

too many elements	More data items supplied for aggregate level in initializer than members of the aggregate. (8.6)
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. (8.8)
undeclared variable	No declaration exists at any block level for this identifier.
undefined structure	Union or struct declaration refers to an undefined structure name. (8.5)
unions not allowed	Cannot initialize union members. (8.6)
unterminated character constant	Unmatched ' character delimiters. (2.14.3)
unterminated string	Unmatched " string delimiters. (2.5)
while expected	No while found for do statement. (9.5)

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.

```
ccl & cc2 (C executives):
     cc [options] file (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 o.link for
                    inclusion as the edition number of the linked
                    module.
          -o        Inhibits assembler 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 equiv 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.oom command file.

     CC2 only:
          -q       Quiet mode. Suppress echo of file names.

c.prep (C macro preprocessor)
     c.prep [options] <path>

     <path> is read as input. c.prep causes c.comp to generate a
     psect directive with the last element of the pathlist and _c as
     the psect name. If <path> is /d0/myprog.o, the 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 ccl/cc2.

c.comp (One-pass compiler)
     c.comp [options] [<tile>] [options]

     If <tile> 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.passl (Pass One of Two-pass Compiler)
c.pass2 (Pass Two of Two-pass Compiler)

    c.passl [options] [<file>) [options]
    c.pass2 [options) [<file>] [options]

    Command line and options are the same as o.comp. If the
    options given to c.passl are not given to c.pass2 also, c.pass2
    will not be able to read the c.passl output. Both c.passl and
    c.pass2 read stdin and write stdout normally.

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.asm (Assembler)

    c.asm <file> [options]

    C.asm reads <file> as assemble language input. Errors are
    written to stderr. Options are turned on or off by the
    inclusion of the option character preceded by a -.

    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).
         -dn      Set lines per page to n. (default 66).
         -wn      Set line width to n. (default 80).

c.link (Linker)

    c.link [options] <mainline> [<subi> {<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 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 by 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 output module name.
—l<path>	Use <path> as library file. A library tile 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 in the final output module, otherwise it is skipped. No mainline psects are allowed in a library file. Library files are searched in 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 if —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 adds 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 BASICO9. <ept> is the name of the function to be transferred to when BASICO9 executes the RUN command.
—t	Allows static data to appear in a BASICO9 callable module. It is assumed the C function called and the calling BASICO9 program have provided a sufficiently large static storage data area pointed to by the Y register.

Using and Linking to User Defined Libraries

A library consists of a group of C procedures or functions that have been separately compiled into Relocatable Object Files (ROF) and subsequently merged into one library file.

If, hypothetically, you had created a set of higher level mathematic functions, that you wanted to convert into a C library. First you would separately compile each one using the —R option. Then you would merge them all into one large library file. If you need to scan the library file for available functions you can use the example program RDUMP.C to inspect any C library file.

For example:

```
0S9:CC1 SIN.C COS.C TAN.C ARCOS.C -R
0S9:CCI ARCSIN.C ARCTAN.C EXP.C LOC.C —R
0S9:CC1 NLOG.C SQRT.C SQR.C CUBE.C —R

Then you would:
0S9:MERGE SIN.R COS.R TAM.R ARCOS.R >TEMP1
0S9:MERGE ARCSIN.R ARCTAN.R EXP.R LOG.R >TEMP2
0S9:MERGE NLOG.R SQRT.R SQR.R CUDE.R >TEMP3
0S9:MERGE TEMP1 TEMP2 TEMP3 >TRIG.L
```

Then to use the library simply use the -l=<pathlist> option in your command line when you compile your program.

When the linker is executed the pathlist specified will be searched to resolve any references made to the functions within the library. The linker searches all specified libraries in the order specified before searching the standard library. The linker will resolve all references on a first found basis. This means that the linker will use the first procedure or function whose name matches a reference to that name and will ignore any additional functions found that have the same name.

Procedures or functions within a library that use other functions within the same library should always appear first. For example, in the above example if the ARCSIN routine used the SIN routine, the SIN routine should be merged into the library file after the ARCSIN. Another way of putting this is that all references to other procedures within a library should be forward references.

Interfacing to BASIC09

The object code generated by the Nicroware C Compiler can be made callable from the BASICO9 “RUN” statement. Certain portions of a BASICO9 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 BASICO9 type **INTEGER** are identical; both are two byte two’s complement integers. C type **char** and BASICO9 type **BYTE** and **BOOLEAN** are also identical. Keep in mind that C will sign—extend characters for comparisons yielding the range —128 to 127.

BASICO9 strings are terminated by OxFF (255). C strings are terminated by 0x00 (0). If the BASICO9 string is of maximum length, the terminator is not present. Therefore, string length as well as terminator checks must be performed on BASICO9 strings when processing them with C functions.

The floating point format used by C and BASICO9 are not directly compatible. Since both use a binary floating point format it is possible to convert BASICO9 reals to C doubles and vice—versa. Multi—dimensional arrays are stored by BASICO9 in a different manner than C. Multi—dimensional arrays are stored by BASICO9 in a column—wise manner; C stores them row—wise. Consider the following example:

```
     BASICO9 matrix: DIM array(5,3):1NTEGER.
     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: mt array[5][3];
     The elements in consecutive memory locations (read left to right, line by line) are
stored as:
     (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 BASICO9 matrix elements in C, the subscripts must be transposed. To access element array (4,2) in BASICO9 use array[2][4) in C.

The details on interfacing BASICO9 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.

Example 1 — Simple Integer Arithmetic 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,s3,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; BASICO9 will not know anything about the name. Page 9—13 of the BASICO9 Reference manual describes how BASICO9 passes parameters to machine language modules. Since BASICO9 and C pass parameters in a similar fashion, it is easy to access BASICO9 values. The first parameter on the BASICO9 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. BASICO9 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 BASICO9 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 btl.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 BASICO9, 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 BASICO9 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 BASICO9 can link to and that the function **addints** is to be the entry point 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 BASICO9 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 BASICO9 RUN command to call the C procedure. The name given in the **-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 entry point offset.

Enter the following BASICO9 program:

```
PROCEDURE btest
    DIM i,j,k:INTEGER
    i = 1
    j = 132
    k = 1O33
    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!

Example 2 - More Complex Integer Arithmetic Case

The next example shows how static memory area 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,s3,arg3,s4)
? char *cmem;
? int *value,*argl,*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 named entcnt global to bt2.o. The parameter **cmem** and the line **char *cmem** 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 BAS1CO9 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 BASICO9. If the first parameter in the BASICO9 RUN statement is an array, and the **idy 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 BASICO9 callable C functions.

To compile this function, the following C compiler command line is used:

```
   cc bt2.c —rs
```

Where cc is ccl or cc2.

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 -t
```

The linker will respond with **BASICO9 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 BASICO9 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 adds 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!

Example 3 - Simple String Manipulation

This example shows how to access BASICO9 strings through C functions. For this example, write the C version of **substr**:

```
build bt3.c
? /* Find substring from BASICO9 string:
?         RUN findstr(A$,B$,fndpos)
?    returns in fndpos the position in A$ that B$ was found or
?    0 if not found. A$ and B$ Dust 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 j;
?     for(i=1;strlen-- > 0 && *str!=0xff; ++i)
?         if(smatch(str++,pat,patlen))
?             return j;
? }
?
? static smatch(str,pat,patlen)
? register char *str,*pat;
? int patlen;
? {
?     while(patlen-- > 0 && *pat != Oxtf)
?         if(*str++ != *pat++)
?             return O;
?     return 1;
? }
?
```

Compile the program:

```
oc bt3.c —rs
```

Where cc is col or cc2

And link it:

```
c.link bt3.r —o=findstr —b=findstr
```

The BASICO9 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 the program is run, it should print the position where the matched string was found in the source string.

Example 4 — Quicksort

The next example programs demonstrate how one might implement a quicksort written in C to sort some BASICO9 data.

C integer quicksort program:

```
#define swap(a,b) { int t; t=a; a=b; b=t; }

/* qsort to be called by BASICO9:
     dim d(100):INTEGER any size INTEGER array
    run oqsort(d,100) calling qsort.
*/

qsort(argcnt, iarray, iasize, icount, icsiz)
int, argcnt1      /* BASICO9 argument count */
     iarray[],    /* Pointer to BASICO9 integer array */
     iasize,      /* and it's size */
     *icount,     /* Pointer to BASICO9 (sort count) */
     icsiz;       /* Size of integer */
{
    sort(iarray,0,*icount); /* initial qsort partition */
}

/* standard quicksort algorithm from Horouitz—Sahni */
static sort(a,m,n)
register int *a,m,n;
{
    register i,j,x;
    if(m < n) (
        i = n;
        j = n + 1;
        x a[m];
        for(;;) {
            do j += 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 BASICO9 program is:

```
PROCEDURE sorter
DIM i,n,d(1000):INTEGER
n=1000
i=RND(-(PI))
FOR 1:1 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
DIN i:INTEGER
FOR i=n TO m
PRINT d(i); " ";
NEXT i
PRINT
END
```

C string quicksort program:

```
/* qsort to be oalled by BASICO9:
      dim cmemory:STRING[1O) This should be at least as large as
                            the linker says the data size should
                            be.
      dim d(100):INTEGER    Any size INTEGER array.
      run cqsort(cwemory,d,100) calling qsort. Note that the pro-
                            cedure name run is the linked OS—9
                            subroutine module. The module name
                            need not be the name of the C tuno—
                            tion.
*/

int maxatr; /* string maximum length */

static strbcmp(strl,str2) /* basic09 string compare */
register char •strl,*str2;
{
    int maxlen;

    for (maxlen = maxstr; *str1 == *str2; ++str1)
        if (maxlen-- > 0 || *str2++ == Oxff)
            return 0;
    return (*str1 - *str2);
}

cssort(argcnt,stor,storsiz,iarray,iasize,elemlen,elsiz,
             icount, icsiz)
int argcnt;            /* BASICO9 argument count */
char *stor;            /* Pointer to string (C data storage) */
char iarray[];         /* Pointer to BASICO9 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 BASICO9. 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 BASICO9
       RUN command. */

#asm
    ldy 6,s get addr for C data storage
#endasm

    /* Use the C library qsort function to do the sort. Our
       own BASICO9 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

    vaect
errno: rmb 2 C function system error number
_flacc: reb 8 C library float/long accumulator
    endsect
#endasm
```

The BASICO9 calling program: (words file contains strings to sort)

```
PROCEDURE ssorter
DIM a(200):STRING[20]
DIM cmemory:STRING[2O]
DIM i,n:INTEGER
DIM path:INTEGER
OPEN #path, "words": READ

n=1OO
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
END
```

The next example shows how to access BASICO9 reals from C functions:

```
flmult(cnt,cmemory,cmemisiz,realarg,realsize)
int cnt;            /* number of arguments */
char *cmemory;      /* pointer to some memory for C use */
double *realarg;    /* pointer a real */
{
#asm
    ldy 6,s get static memory address
#endasm

    double number;
    getbreal(&number,realarg); /* get the BASICO9 real */
    number *= 2.,; /* number times two */
    putbreal(realarg,&number); /* give back to BASICO9 */
}

/* getbreal(creal,breal)
    get a 5-byte real from BASICO9 format to C format */
getbreal(clreal,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 BASICO9 real
    ldx 0,s get address of B real

    cira clear the C double
    clrb
    std 0,u
    std 2,u
    std 4,u
    stb 6,u
    ldd 0,x
    beq g3 BASICO9 real is zero

    ldd 1,x get hi B mantissa
    anda #$7f clear place for sign
    std 0,u put hi C mantissa
    ldd 3,x get lo B mantissa
    andb #$fe mask off sign
    std 2,u put lo C mantissa
    lda 4,x get B sign byte
    isra shift out sign
    bcc g1
    lda 0,u get C sign byte
    ora #$80 turn on sign
    sta O,u put C sign byte
gl  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 BASICO9 */
putbreal(breal, creal)
double *breal,*creal;
{
    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 BASICO9 real
    ldx 0,s get address of B real

    lda 7,u get C exponent
    bne p0 not zero?
    clra clear the BASICO9
    clrb real
    std 0,x
    std 2,x
    sta 4,x
    bra p3 and exit

p0  ldd 0,u get hi C mantissa
    ora #$80 this bit always on for normalized real
    std 1,x put hi B mantissa
    ldd 2,u get lo C mantissa
    std 3,x put lo B mantissa
    incb round mantissa
    bne p1
    inc 3,x
    bne pl
    inc 2,x
    bne pl
    inc 1,x
p1  andb #$fe turn off sign
    stb 4,x put B sign byte
    lda O,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 BASICO9 */
_stkcheck:
_stkchec:
    rts

    vsect
flacc: rmb 8
errno: rmb 2
    endsect
#endasm
```

BASICO9 calling progras:

```
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
```

Example 5 - Matrix Elements

The last program is an example of accessing BASICO9 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 to 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
```

BASICO9 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
```

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 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 is 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 0S-9 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.

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.

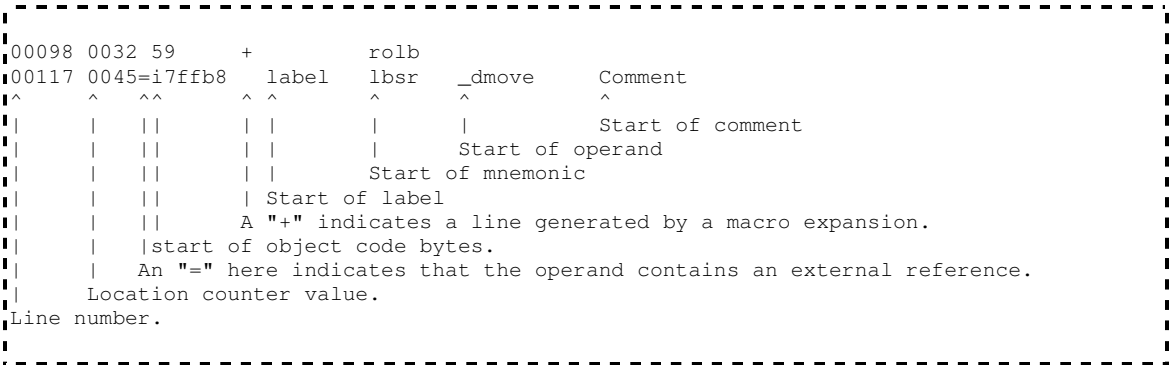
Label Field

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.

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.

Listing Format



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

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 appropriate 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 appropriate for the statement. Therefore, the mnemonics allowed within a section are restricted as follows:

These mnemonics are allowed inside or outside any section: nam, opt, ttl, pag, spa, use, fail, rept, endr, ifeq, ifne, iflt, ifle, ifge, ifgt, ifp1, 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.

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/lang usage byte. If this PSSECT 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 module, this should be set to zero.

</nowiki></pre>

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. There 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+Objet,Reent+1,Edit,0,progent
psect another_prog,0,0,0,0,0
```

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 appropriate uninitialized data section. The fcc, fdb, fcs, fcb and rzb (reserve zeroed bytes) directives place data into the appropriate 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.

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.

RZB Statement

Syntax: RZR <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.

COMPARISON BETWEEN ASSEMBLY PROGRAMS FOR THE MICROWARE INTERACTIVE ASSEMBLER AND THE RELOCATING MACRO ASSEMBLER

The following two program examples simply fork BASICO9. The purpose of the examples are to show some of the differences in the new relocating assembler. The differences are apparent.

```
* this program forks 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    $0D
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
```

MACRO INTERACTIVE ASSEMBLER SOURCE

```
          ifp1
          use    defsfile
          endc

          mod    siz,prnam,type,revs,start,size
prnam     fcs    /testshell/
type      set    prgrm+objct
revs      set    reent+1

          rmb    250
          rmb    200
name      fcs    /basic09/
prm       fcb    $0D
prmsize   equ    *-prm
size      equ    .
start     equ    *
          leax   name,pcr
          leau   prm,pcr
          ldy    #prmsize
          lda    #PRGRM+OBJCT
          clrb
          os9    F$Fork
          os9    F$Wait
          os9    F$Exit
          emod
siz       equ    *
```

Macros

Sometimes identical or similar sequences of instructions may be repeated in different places in a program. The problem is that if the sequence of instructions is long or must be used a number of times, writing it repeatedly can be tedious.

A macro is a definition of an instruction sequence that can be used numerous places within a program. The macro is given a name which is used similarly to any other instruction mnemonic. Whenever RMA encounters the name of a macro in the instruction field, it outputs all the instructions given in the macro definition. In effect, macros allow the programmer to create “new” machine language instructions.

For example, suppose a program frequently must perform 16 bit left shifts of the D register. The two instruction sequence can be defined as a macro, for example:

```
dasl macro
     aslb
     rola
     endm
```

The **macro** and **endm** directives specify the beginning and the end of the macro definition, respectively. The label of the **macro** directive specifies the name of the macro, **dasl** in this example. Now the "new" instruction can be used in the program:

```
       ldd 12,s get operand
       dasl     double it
       std 12,s save operand
```

In the example above, when RMA encountered the **dasl** macro, it actually outputted code for **aslb** and **rola**. Normally, only the macro name is listed as above, but an RMA option can be used to cause all instructions of the "macro expansion" to be listed.

Macros should not be confused with subroutines although they are similar in some ways. Macros repetitively duplicate an "in line" code sequence every time they are used and allow some alteration of the instruction operands. Subroutines appear exactly once, never change, and are called using special instructions (**BSR**, **JSR**, and **RTS**). In those cases where they can be used interchangeably, macros usually produce longer but slightly faster programs, and subroutines produce shorter and slightly slower programs. Short macros (up to 6 bytes or so) will almost always be faster and shorter than subroutines because of the overhead of the **BSR** and **RTS** instructions needed.

Macro Structure

A macro definition consists of three sections:

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

```
<name>  MACRO     /* macro header */
          .
          .
        body      /* macro body */
          .
          .
        ENDM      /* macro terminator */
```

The macro name must be defined by the label given in the **MACRO** statement. The name can be any legal assembler label. It is possible to redefine the 6809 instructions (**LDA**, **CLR**, etc.) themselves by defining macros having identical names. Caution: redefinition of assembler directives such as **RMB** can have unpredictable consequences.

The body of the macro can contain any number of legal RMA instruction or directive statements including references to previously defined macros. The last statement of a macro definition must be **ENDM**.

The text of macro definitions are stored on a temporary file that is created and maintained by RMA. This file has a large (1K byte) buffer to minimize disk accesses. Therefore, programs that use more than 1K of macro storage space should be arranged so that short, frequently used macros are defined first so they are kept in the memory buffer instead of disk space.

Macro calls may be nested, that is, the body of a macro definition may contain a call to another macro. For example:

```
times4 MACRO
       dasl
       dasl
       ENDM
```

The macro above consists of the **dasl** macro used twice. The definition of a new macro within another is not permitted. Macro calls may be nested up to eight deep.

Macro Arguments

Arguments permit variations in the expansion of a macro. Arguments can be used to specify operands, register names, constants, variables, etc., in each occurrence of a macro.

A macro can have up to nine formal arguments in the operand fields. Each argument consists of a backslash character and the sequence number of the formal argument, e.g, \1, \2 ... \9. When the macro is expanded, each formal argument is replaced by the corresponding text string "actual argument" given in the macro call. Arguments can be used in any part of the operand field not in the instruction or label fields. Formal arguments can be used in any order and any number of times.

For example, the macro below performs the typical instruction sequence to create an OS—9 file:

```
create MACRO
       leax   \1,pcr  get addr of file name string
       lda    \2      set path number
       ldb    #\3     set file access modes
       os9    I$CREATE
       ENDM
```

This macro uses three arguments: "\1" for the file name string address; "\2" for the path number; and "\3" for the file access mode code. When **create** is referenced, each argument is replaced by the corresponding string given in the macro call, for example:

```
       create outname,2,$1E
```

The macro call above will be expanded to the code sequence:

```
       leax   outname,pcr
       lda    #2
       ldb    #$1E
       os9    I$CREATE
```

If an argument string includes special characters such as backslashes or commas, the string must be enclosed in double quotes. For example, this macro reference has two arguments:

```
       double count,"2,s"
```

An argument may be declared null by omitting all or some arguments in the macro call to make the corresponding argument an empty string so no substitution occurs when it is referenced.

There are two special argument operators that can be useful in constructing more complex macros. They are:

\L*n* Returns the length of the actual argument *n*, in bytes.

\# Returns the number of actual arguments passed in a given macro call.

These special operators are most commonly used in conjunction with RMA's conditional assembly facilities to test the validity of arguments used in a macro call, or to change the way a macro works according to the actual arguments used. When macros are performing error checking they can report errors using the **FAIL** directive. Here is an example using the **create** macro given on the previous page but expanded for error checking:

```
create  MACRO
        ifne  \# - 3 must have exactly 3 args
        FAIL create: must have three arguments
        endc
        ifgt \L1 - 29 file name can be 1 — 29 chars
        FAIL create: file name too long
        endc
        leax \1,pcr get addr of file name string
        ida  #\2 set path number
        ldb  #\3 set file access modes
        os9  I$CREATE
        ENDM
```

Macro Automatic Internal 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 the 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 "@".

Here is an example of macro that uses internal labels:

```
testovr MACRO
        cmpd  #\1    compare to arg
        bls   \@A    bra if in range
        orcc  #1     set carry bit
        bra   \@B    and skip next instr.
\@A     andcc #$FE   clear carry
\@B     equ    *     continue...
        ENDM
```

Suppose the first macro call is:

```
        testovr $80
```

The expansion will be:

```
        cmpd  #$80   compare to arg
        bls   @001A  bra if in range
        orcc  #1     set carry bit
        bra   @001B  and skip next instr.
@001A   andcc #$FE   clear carry
@001B   equ   *      continue...
```

If the second macro call is:

```
        testovr #240
```

The expansion will be:

```
        cmpd  #240   compare to arg
        bls   @002A  bra if in range
        orcc  #1     set carry bit
        bra   @002B  and skip next instr.
@002A   andcc #$FE   clear carry
@002B   equ   *      continue...
```

Additional Comments About Macros

Macros can be an important and useful programming tool that can significantly extend RMA's capabilities. In addition to creating instruction sequences, they can also be used to create complex constant tables and data structures.

Macros can also be dangerous in the sense that if they are used indiscriminately and unnecessarily they can impair the readability of a program and make it difficult for programmers other than the original author to understand the program logic. Therefore, when macros are used they should be carefully documented.