CHAPTER 15

MICROWARE C AND ASSEMBLY LANGUAGE

Programming in C is usually far more productive than programming in assembly language, and produces code that is more readable, portable, and maintainable. However, assembly language is still used where speed and memory efficiency are important. For these reasons, most of OS-9 itself is written in assembly language, (whereas for portability OS-9000 is written in C). C programs wishing to make calls to the OS-9 operating system must use interface functions written in assembly language.

This section describes the interface between C and assembly language under OS-9, and shows how operating system components such as file managers and device drivers can be written in C.

15.1 MICROWARE C

The latest version of the Microware C compiler (version 3.2, supplied with OS-9 version 2.4) is an up-to-date and complete implementation of the C language, including bit fields, enumerated constants, structure assignment, and structure return. There are no limits on the length of symbol names, and the same element name may be used in different structure definitions. However, the additional features from the ANSI C standard are not incorporated. Microware has announced that an ANSI standard C compiler is in development - at the time of writing its release is imminent.

Microware C conforms to the standard laid down by Kernighan and Ritchie in their definitive book "The C Programming Language". A complete set of library functions are provided, giving both UNIX compatibility (as far as possible), and access to OS-9 specific functions. Both "buffered file" functions (such as **printf()** and **fread()**), and low level path I/O functions (such as **open()** and **read()**) are provided.

15.2 CIO AND MATH TRAP HANDLERS

The most commonly used I/O library functions are also provided in the **cio** trap handler. This saves on memory, disk space, and time loading programs from disk by making the program much smaller – the program makes trap calls to the I/O functions in **cio**, rather than having the code for the functions in the program module itself.

The decision whether to use **cio**, or to use library functions included in the program, is taken at compile (link) time. The '-i' option to the **cc** executive requests that library functions (from 'LIB/cio.l') be used that simply make trap calls to the **cio** trap handler. If this option is omitted, the program is linked with library functions that perform the I/O functions themselves (from 'LIB/clibn.l' or 'LIB/clib.l'). The functions will execute slightly faster than calling the trap handler, but the program will be significantly larger.

Similarly, math functions can be included in the program, or can make trap calls to the **math** trap handler module. If the '-x' option is specified to **cc**, the trap handler technique is used, by linking to the 'LIB/clib.l' library and generating "in-line" trap instructions in the program. Otherwise the 'LIB/math.l' library is used. In addition, if the target processor is indicated as a 68020/030/040 (using the '-k' option of **cc**), the compiler will use the additional integer arithmetic instructions of these members of the 68000 family, and if the '-k2F' option is used to indicate that the 68881/68882 floating point coprocessor is available (or the processor is a 68040), the compiler will use in-line floating point instructions.

15.3 THE REMOTE DIRECTIVE

All OS-9 programs use register indirect indexing (relative to the **a6** register) to access static storage memory. However, the 68000 and 68010 are limited to signed 16-bit constant offset indexing with address registers. The linker automatically offsets data space accesses by 32k bytes, allowing 64k bytes of storage to be accessed using constant offset indexing. To access more static storage than this the compiler must use a different form of addressing. The OS-9 C compiler offers three options to do this.

The first approach uses the **remote** directive, which is an additional keyword in the Microware C language, prefixing static storage definitions and declarations. For example:

```
    remote static int x;
```

This declares that **x** is likely to lie outside of the 64k limit, and therefore an alternative addressing mode must be used. The C compiler generates two instructions for each access to such variables, as it must load a data register with the long word offset, and then use register indirect addressing with index. The linker places all **remote** variables after all ordinary static storage, so it is usually sufficient to define only large arrays as **remote**. The **remote** keyword is specific to the OS-9 C compiler, and so its use is not portable.

The second approach is to use the option '-k0L' with the **cc** executive. This instructs the compiler to generate two instructions for each static storage access, as if all static variables had been defined as **remote**. This makes programs more portable (and simplifies the porting of existing programs to OS-9), but also makes the programs slower and larger than necessary.

The third approach is only available with the 68020/030/040 version of the C compiler. The '-k2L' option allows the user to specify that long word constant offset indexing be used for data space accesses - this is an additional addressing mode in the 68020/030/040. This causes all static storage references to use long word offsets, allowing the full use of the 32-bit addressing capability of the 68000 family. However, it does mean that accesses to the first 64k bytes of the program's static storage are using 32-bit offsets when 16-bit offsets would do, making the program longer and slower.

As most programs do not need more than 64k bytes of static storage (large buffers are instead dynamically allocated, and addressed via pointers), these techniques are generally not required.

The same problem applies to function calls in programs larger than 32k bytes, as the 68000 **bsr** instruction is limited to a signed 16-bit offset. The linker therefore uses a jump table in the data space for all references that exceed +/-32k bytes; it patches over the **bsr** instruction generated by the compiler with a **jsr** relative to the data space. The 68020/030/040 version of the C compiler allows the user instead to specify (using the '-k2CL' option) that long word offsets be used with the **bsr** instruction, as this is available on the 68020/030/040 processors. This may be advantageous for very large programs, but bear in mind that all function calls will then use long word offsets, creating a larger and perhaps slower program.

Similarly, the '-k0CL' option causes the C compiler to generate two instructions for every function call instead of a simple **bsr** instruction. This approach is compatible with the 68000 and 68010, which do not support long word **bsr** offsets.

Note that if any of the **cc** '-k2' options are used, causing the compiler to generate code using the additional instructions and addressing modes available on the 68020/030/040, the program cannot be run on a 68000 or 68010.

15.4 PROGRAM STARTUP

The C programmer expects execution to start with his **main()** function. However, because the kernel passes parameters to a newly forked program in the processor registers, it is necessary for an assembly language "core" to be called first, which then calls **main()**. This "core" is in the file 'C/SOURCE/cstart.a'. It is provided already assembled in the file 'LIB/cstart.r'. The **cc** executive automatically makes this the first Relocatable Object File (ROF) in the linker command line it generates. 'cstart.r' also is the only file in a C program that has a "non-null" ("root") **psect** directive. The **psect** assembler directive specifies the output module type, attributes, and revision number. The linker permits only one root **psect**, so files produced by the C compiler have null **psect** directives.

If **cc** is used to generate the linker command line, 'cstart.r' is automatically included. However, if you create your own linker command line you must manually include 'cstart.r' as the first ROF in the command line. Try using the '-bp' option of the **cc** executive to display the command lines that **cc** generates.

15.5 C WITH ASSEMBLY LANGUAGE

The problems in interfacing C to assembly language are to do with parameter passing and register usage. Provided that the C compiler's parameter passing and register usage schemes are understood, assembly language functions can be written that are called from C functions, or that call C functions.

The C compiler uses a simple and consistent parameter passing mechanism. The first two long words of parameters are passed to a function in the **d0** and **d1** registers. If the first parameter is a double precision floating point number it is passed in **d0** and **d1**. If the first parameter is not a double, but the second is, **d1** is not used. The remaining parameters are on the stack (above the return address). If a function returns a value, the returned value is in **d0** (and **d1** if the returned value is a **double**). If the returned value is a structure, its *address* is returned in **d0** – the structure itself is built in static storage private to the called function (as described below).

The only register usage convention that the assembly language programmer needs to know (and can rely on) is that the address of the program static storage is always held in the **a6** register.

Because the compiler's use of registers cannot be relied on (future releases of the compiler may behave differently), you should never embed assembly language within C functions. Instead, always use separate assembly language subroutines, called as functions from the C code. For the same reason, the assembly language subroutine should always preserve all of the processor registers other than **ccr**, **d0**, and **d1**. Indeed, even **d0** and **d1** should be preserved if they are not used to pass a parameter or return a value.

To increase the portability of your C code, the assembly language functions should be placed in a separate source file. Only this file will need modification if your C source code is ported to a different processor or compiler. Note that if an assembly language symbol is to be used from a different source file, it must be made public. The Microware assembler makes a symbol public if its name is immediately followed by a colon in the line defining the symbol.

15.6 REGISTER VARIABLES

The C language provides for automatic variables (temporary variables within functions) and parameters passed to functions to be declared as **register**. The C compiler will try to place these variables in processor registers, rather than in memory allocated from the stack. Because such variables do not have to be read from memory when their value is required, or written to memory when their value is changed, code fragments using such variables are much smaller and faster. This is particularly important if the variable is used frequently – for example, a pointer to **char** used for parsing a string, or an **int** used as the controlling variable in a **for** loop.

The C language specification states that the compiler will assign register variables to processor registers in the order in which the definitions appear. Once all the available processor registers have been used, further **register** definitions will create ordinary automatic variables on the stack. Therefore the order in which the register variables are defined is very important, especially if the code is intended to be portable – another processor may have fewer registers, and another compiler may make fewer registers available for register variables.

The Microware C compiler makes four data registers (**d4–d7**) and three address registers (**a2–a4**) available for register variables. The data registers are used for register variables of types **char**, **short**, **int**, and **long** (and the unsigned equivalents). The address registers are used for pointers of any type. If the '-k2F' option of the **cc** executive has been used to indicate that a floating point unit is available, the C compiler makes six FPU registers (**fp2–fp7**) available for register variables of type **double** or **float**.

15.7 CODING FOR SPEED

It is a widely stated axiom that a program spends most of its time executing only a very small percentage of its code in a frequently repeating loop. By carefully identifying such code fragments and making judicious use of register variables within them, the programmer can make his program run significantly faster.

The following example of a function to copy a string executes very much faster because register variables have been used. Try compiling this function with the '-a' option of the **cc** executive, and study the output with and without the use of the **register** keyword. Using the '-c' option as well will keep the C source code as comments in the assembly language file. Notice that the Microware C compiler makes use of special features of the 68000 instruction set, such as the post-increment addressing mode, to reduce the code length and increase the speed.

```
void strcpy(d,s)
register char *d,           /* string to copy to */
              *s;           /* string to copy from */
{
    do {
    } while ((*d++=*s++)!='\0');
}
```

Because the C compiler must not only conform to the C language specification, but must also make use of the 68000 instruction set as provided by Motorola, careful use of certain coding techniques can also significantly improve execution speed. For example, the two fragments shown below produce the same effect (copy a given number of bytes):

```
void copybytes(d,s,n)
register char *d,           /* where to copy to */
              *s;           /* where to copy from */
register int n;             /* number of bytes to copy */
```

```
{
#ifdef FIRST_METHOD
    while (n--)
        *d++=*s++;
#else
    if (n!=0) {
        do {
            *d++=*s++;
        } while (--n);
    }
#endif
}
```

However, the second method will execute much more rapidly. In the first method, to comply with the C language specification, the register variable **n** must be saved (the Microware compiler saves it to **d0**) and then decremented, and then the saved value must be tested, and a conditional branch taken on the result – four instructions, plus one for the instruction to copy the byte, making five in total for the loop. In the second method, a byte is copied, then the register variable is decremented, and a conditional branch taken on the result – a total of three instructions. Thus the second method will execute significantly faster. The extra instruction `if (n!=0)` is unimportant for the speed of the function, because it is executed only once.

If program execution speed is important, you should identify the small percentage of the program that is consuming most of the processor's time. If you are already familiar with the behaviour of the C compiler, you can then recode those fragments for optimum speed of execution. Otherwise, use the '–a' option of the **cc** executive to compile the C source code to assembly language, and study the behaviour of the C compiler in those critical code fragments. This should help you decide on appropriate modifications to the C source code.

If, after having optimized your C source code in this way, you feel that the output of the C compiler is still not producing the fastest code possible, you can write a separate assembly language function to be called instead of the C code. Of course, this should be a last resort, as it is less portable and less readable.

A corollary to the axiom that the program spends most of its time executing only a small part of the code, is that the error handling parts of the program are executed very infrequently – most of the time a program does not generate errors. Therefore it is reasonable to allow error handling code to be designed to be easy to write, and informative in its output, rather than trying to code it carefully for rapid execution. This applies to other parts of the program that will be used infrequently (perhaps only once), such as command line parameter parsing.

15.8 THE 'LINK' INSTRUCTION

The C compiler uses the 68000 **link** instruction to capture the stack pointer on entry to a function. This is required by the Microware C Source Level Debugger, to give access to the stack frame (parameters passed, and automatic variables) during execution of the function (for example at a breakpoint or when single stepping).

Note that because the Source Level Debugger has no information about the parameter structure or stack allocation for variables in a function written in assembly language, the **link** and **unlk** instructions are not strictly needed when writing a function in assembly language, although including them allows the **frame** command of **srcdbg** (and the **w** command of **debug**) to report which function called the assembly language function.

The following example shows typical code produced by the C compiler. The C function:

```
set5(p)
int *p;
{
    int x;
    x=5;
    *p=x;
}
```

produces the following assembly language when compiled (I have added the comments!):

```
set5:       link    a5,#0           stack a5, put sp in a5, add 0 to sp
            movem.l d0/a0,-(sp)     save parameter and a0 register
            move.l  #-68,d0         ensure at least 68 bytes stack free
            bsr     _stkcheck
            subq.l  #4,sp           allocate automatic: x
            moveq.l #5,d0           set x = 5
            move.l  d0,(sp)
            movea.l 4(sp),a0        get parameter: p
            move.l  (sp),(a0)       copy x to *p
            addq.l  #4,sp           de-allocate x
            movem.l -4(a5),a0       retrieve a0 register
            unlk    a5              put a5 in sp, unstack a5
            rts
```

Note the use of the stack for automatic variables, and for temporary storage. The **_stkcheck()** function is called to determine that enough stack space is available for the function's needs. If not, the **_stkcheck()** function exits the program with a `*** stack overflow ***` message. The **_stkcheck()** function is part of 'cstart.a', and uses static storage variables initialized by 'cstart.a' to determine whether there is sufficient stack available. Because 'cstart.a' is only used when creating a program, the **_stkcheck()** function is not appropriate for the creation of operating system components such as device drivers and file managers – either stack checking must be disabled using the '–s' option of **cc**, or the programmer must supply an alternative **_stkcheck()** function.

15.9 A FUNCTION IN ASSEMBLY LANGUAGE

One common reason for writing a C–callable function in assembly language is to supplement the standard C libraries. Microware have not supplied library functions for the privileged (system state only) system calls. The example below shows a typical assembly language function to make the **F$IRQ** system call, with an example of this function being called from C:

```
{
    if ((f_irq(vector,priority,handler,port))==ERROR)
        printf("Error #%d\n",errno);
}
```

This call to the function **f_irq** is passing four parameters. The first parameter (the interrupt vector number) will be in the **d0** register, and the second parameter (the software polling priority) will be in the **d1** register. The remaining two parameters (the address of the interrupt handler function, and the address of the interface) will be on the stack. The third parameter will be just above the return address (at **4(a7)**), while the fourth parameter is above that (at **8(a7)**).

The following implementation of **f_irq** makes good use of the fact that the parameters have been ordered so that **d0** and **d1** already contain the correct parameters for the **F$IRQ** system call, and the remaining two parameters are in the correct order to be picked up by a single **movem** instruction. As always, the **a6** register contains the static storage address, which must be copied to the **a2** register as a parameter to the **F$IRQ** system call.

The function returns 0 if the **F$IRQ** call gave no error. Otherwise, the error code is extended to a long word and saved in the static storage field **errno**, and –1 is returned. Note that the returned value is in the **d0** register, as required by the C compiler. The function saves all the registers that it modifies (including **d1**, which will be modified if **F$IRQ** returns an error). This takes 16 bytes of space on the stack, so the third parameter passed to the function is at **20(a7)** – 16 bytes of temporary storage, plus 4 bytes of return address.

```
#asm
f_irq:        movem.l d1/a0/a2-a3,-(a7)   save registers
* d0 and d1 already contain the correct parameters
              movem.l 20(a7),a0/a3      get handler and port addresses
              movea.l a6,a2             copy static storage address
              os9     F$IRQ
              bcs.s   f_irq10           ..error
              moveq   #0,d0             no error - return 0
              bra.s   f_irq20
f_irq10       move.l  d1,errno(a6)      save OS-9 error code
              moveq   #-1,d0            indicate error
f_irq20       movem.l (a7)+,d1/a0/a2-a3   retrieve registers
              rts
#endasm
```

Note that the symbol **f_irq** is terminated by a colon. This causes the assembler to make the symbol public, so the function can be called from another source file. Of course, it is also good practice to provide a proper declaration for the function in your C source code. Note also that although the OS-9 error codes are 16-bit values, the kernel ensures error codes returned by system calls are 32-bit, with the high word set to zero.

15.10 STRUCTURE RETURN

A structure can be of any length. Therefore it is not possible to use processor registers to return a structure. Instead, if a function is defined as returning a structure, the compiler reserves a private block of static storage for the returned structure, and returns a pointer to the structure. The following example illustrates parameter passing with structure return. It shows a C program, and the output of the C compiler in assembly language. This example also shows the use of register variables by the C compiler.

```
#include <stdio.h>
typedef struct {             /* declare a structure type */
    int x,y,z;
} int3;

int3 setints(); /* declare a function returning an item of that type */

main()
{
    int3 n;                  /* define an automatic of that type */
    n=setints(4,5,6);        /* set it from the value returned */
    printf("%d %d %d\n",n.x,n.y,n.z);   /* display the results */
}
```

```
/* Define the function returning an item of the structure type: */
int3 setints(i,j,k)
register int i,j,k;
{
    int3 m;                 /* define an automatic of that type */
    m.x=i;                  /* fill it with the given values */
    m.y=j;
    m.z=k;
    return(m);              /* return the structure */
}
```

The following assembly language output was obtained using the command:

```
$ cc -qixa test1.c
```

The '-a' option instructs the **cc** executive to save the assembly language output of the C compiler in a file 'test1.a', and not to proceed to assemble and link the program. Of course, the C compiler does not generate comments for its assembly language output – the comments are my addition!

```
            psect   test1_c,0,0,0,0,0   "null" psect starts code file
            nam     test1_c             nam and ttl are listing directives
            ttl     main
* Note that symbols not declared with 'static' are public:
main:       link    a5,#0               save a5, set stack frame ptr
            movem.l #_1!1,-(sp)         save registers d0-d1/a0
            move.l  #_3,d0              check 88 bytes of stack free
            bsr     _stkcheck
            lea     -12(sp),sp          allocate stack for 'n'
            pea     6.w                 third parameter is 6
            moveq.l #5,d1               second parameter is 5
            moveq.l #4,d0               first parameter is 4
            bsr     setints             call function 'setints'
            addq.l  #4,sp               ditch third parameter
            movea.l d0,a0             copy returned value - ptr to structure
            move.l  (a0),(sp)           copy the structure to 'n'
            move.l  4(a0),4(sp)         three long words
            move.l  8(a0),8(sp)
            move.l  8(sp),-(sp)         fourth parameter is 'n.z'
            move.l  8(sp),-(sp)         third parameter is 'n.y'
            move.l  8(sp),d1            second parameter is 'n.x'
            lea     _5(pc),a0           point at format string
            move.l  a0,d0               it is the first parameter
            bsr     printf              call function 'printf'
            addq.l  #8,sp               ditch parameters on stack
            lea     12(sp),sp           de-allocate stack used for 'n'
_4          movem.l -8(a5),#_1          retrieve registers d1/a0
            unlk    a5                  restore a5 and stack ptr
            rts                         return to 'cstart'
_3          equ     0xffffffa8
_1          equ     0x00000102
_2          equ     0x00000014
```

```
* Function 'setints'. Note that the 'link' instruction saves a5 on the
* stack, so after 6 other registers have been saved, the third
* parameter is at 32(a7). Notice also how the compiler makes use of
* the d4, d5, and d6 registers for 'register' variables:
setints:     link    a5,#0              save a5, set stack frame ptr
             movem.l #_6!3,-(sp)        save registers d0-d1/a0-a2/a4
             move.l  d0,d4              copy first parameter to register
             move.l  d1,d5              copy second parameter to register
             move.l  0+_7(sp),d6        copy third parameter to register
             move.l  #_8,d0             check 76 bytes of stack free
             bsr     _stkcheck
             vsect                      start local static storage
             align                      force word alignment
_10          ds.b12                     reserve storage for structure
*                                       return
             ends                       end local static storage
             lea     -12(sp),sp         allocate stack for 'm'
             move.l  d4,(sp)            'm.x' = first parameter
             move.l  d5,4(sp)           'm.y' = second parameter
             move.l  d6,8(sp)           'm.z' = third parameter
             move.l  (sp),_10(a6)       copy 'm' to local static storage
             move.l  4(sp),_10+4(a6)        three long words
             move.l  8(sp),_10+8(a6)
             lea     _10(a6),a0         point at the local structure
             move.l  a0,d0              it is the returned value
             lea     12(sp),sp          de-allocate stack used for 'm'
             bra     _9
             nop
_9           movem.l -16(a5),#_6        retrieve registers a0-a2/a4
             unlk    a5                 restore a5 and stack ptr
             rts                        return to 'main'
_8           equ     0xffffffb4
_6           equ     0x00000170
_7           equ     0x00000020
_5           dc.b"%d %d %d",$d,$0       the 'printf' format string
             ends                       end of code
```

15.11 CALLING C FROM ASSEMBLY LANGUAGE

The OS-9 kernel is written in assembly language, and device drivers and file managers have traditionally also been written in assembly language. Therefore the operating system interface to file managers and device drivers uses processor registers for parameter passing, and is not directly compatible with the output of the C compiler. However, file managers, device drivers, and other operating system components can be written in C, provided an appropriate assembly language "skeleton" is used. This can (and, in fact, must) be in a separate source file, so the programmer need only be concerned with writing in C. Once the appropriate skeleton has been written, it can be used without modification for other device drivers (or file managers, and so on).

As described above, the C compiler stack checking routine is not compatible with the system state stack usage within the operating system, so the '-s' option of the **cc** executive must be used, to instruct the C compiler not to make calls to the **_stkcheck()** function to check for free stack space. Alternatively, the programmer can write his own **_stkcheck()** function, to check for stack overflow within the system state stack (which uses the second half of the process descriptor). If this is done, any interrupt service routine must be in a separate source file which is compiled with stack checking disabled, because a different stack (the interrupt stack) is used during interrupt processing.

If variables are defined as static storage (either because they are defined in the outermost scope of the source file, or because they are prefixed with the **static** keyword), the C compiler generates a **vsect** to indicate to the linker that static storage is required, and generates instructions using addressing relative to the **a6** register to access them. When creating the final output module, the linker adds up all the static storage definitions from **vsect** sections, and puts the total in the "memory requirement" field of the module header (**M$Mem**). The linker also adjusts all instructions that address static storage locations, to take account of the previously allocated static storage from previous source files. Lastly, if the output module is of type "trap handler" or "program", the linker adjusts all static storage addressing references by -32k bytes, as the kernel adds 32k bytes to the static storage pointer register (**a6**) when forking a program or calling a trap handler. This helps compensate for the signed 16-bit constant offset indexing limitation of the 68000 (as described above).

The C compiler is intended for the production of programs, so it generates static storage variable references using the **a6** register. When writing an operating system component, such as a device driver or file manager, C static storage definitions can be used to provide references into a chosen memory structure, provided the address of that memory structure is placed in the **a6** register by the assembly language "skeleton". If the "memory requirement" field (**M$Mem**) of the module header of the operating system component is not used, any memory structure can be chosen. For example, a file manager might use static storage references to access the path descriptor fields. Otherwise, the chosen memory structure must be the structure whose size is determined by the "memory requirement" field, because the linker generates this field by adding the sizes of all the static storage definitions (**vsects**). For example, this field in the header of a device driver is used by the kernel to determine the size of the Device Static Storage to be allocated. Therefore static storage definitions and declarations in a device driver must refer to variables in the Device Static Storage, just as they would if the device driver were written in assembly language.

An important consideration when writing operating system components in C is the amount of stack used. During a system call, the second half of the process descriptor is used for the stack. This gives a total of about 1k bytes of stack. During an I/O call this will be used by the kernel, the file manager, and the device driver, and by any system calls that these components make themselves. It is easy to use up a lot of stack when writing in C. The compiler uses stack for preserving registers, for automatic variables, and for passing parameters when calling other functions. Because a stack overflow in system state is catastrophic (the upper part of the caller's process descriptor will be corrupted), it is important to be sparing in the use of C features that will cause stack usage. Stack space can be saved by minimizing the number of levels of nesting of function calls, and by using static storage variables rather than automatic variables and function parameters. Using register variables does not save stack space, as the compiler will save the current contents of the registers onto the stack on entry to the function.

If you suspect that stack overflow may be a problem, you can add a system state stack checking function, as shown in the example skeleton for a file manager below. Also, if you set a breakpoint (using the system state debugger) within the device driver, you can display the process descriptor memory, and see if the stack usage is approaching the top of the process descriptor variables structure. As a last resort, if you need more stack, you can add a stack switching capability to the device driver or file manager skeleton, perhaps using the field **P$ExpStk** in the Process Descriptor (see the section on the Process Descriptor in the chapter on OS-9 Internal Structure). This uses a stack space allocated by the driver when it is initialized, or by the file manager when the path is opened. The skeleton routine saves the current stack pointer (perhaps in the Device Static Storage or **P$ExpStk**), and then sets the stack pointer to the top of the allocated stack space. On return from the C function, it restores the original stack pointer.

If this technique is used, it is important to be sure that there cannot be concurrent calls that would use the same stack. For example, there cannot be concurrent calls on the same path (the kernel queues such calls), so a stack space allocated when a path is opened is secure against concurrent usage. Similarly, SCF and RBF queue concurrent calls into the device driver, by marking the Device Static Storage (except for SCF Get Status calls).

Therefore a device driver could use a stack space allocated during its initialization routine. However, a file manager should not use such a stack space, as there may be multiple concurrent calls to the file manager for the same device, but on separate paths. A file manager must therefore use a stack allocated for each path.

The problem of concurrent access can be completely relieved by allocating a stack extension to each process, saving the address and size in the **P$ExpStk** field of the process descriptor. If a stack extension is allocated for each process, the stack allocated must be sufficient for all file managers and device drivers needing extra stack space that the process may call. Note that the kernel does not use the **P$ExpStk** field. If a stack extension is allocated using this field, an extension to the **F$DelTsk** system call must be added (perhaps in a kernel customization module) to check this field and de-allocate any stack extension when the process is terminated. Similarly, an extension to the **F$AllTsk** system call could be added to allocate the stack extension when the process is created, rather than leaving it to the discretion of individual file managers and device drivers. Note that if this technique of "global" stack switching is used, any operating system components that perform stack checking by expecting the system state stack to be in the process descriptor will fail.

15.12 A DEVICE DRIVER IN C

As described above, a device driver written in C requires an assembly language "skeleton" file. This skeleton has four main functions which cannot be provided from a C source file:

a) Provide a root **psect**, giving the module type and the address of the table of routine offsets to the linker.

b) Generate the table of offsets to the device driver routines (the routines in the skeleton).

c) In each routine, convert the parameters passed by the kernel or file manager into a form suitable for passing to a C function, and then call the C function.

d) In each routine, on return from the C function, convert the returned error status into the standard OS-9 format expected by the kernel or file manager, and convert any returned values into the format expected by the kernel or file manager.

The device driver skeleton shown below is for a device driver working with the RBF file manager. It can be used for any RBF device driver. A different skeleton is required for each file manager, because the parameters passed are different, but the skeleton below can easily be adapted to any file manager – the principles are the same. In the skeleton below, the **a5** register is reset to zero on entry to each function, to indicate to a source level debugger that this is the top of the available stack frame. The instructions are not strictly necessary unless a source level debugger is going to be used to debug the driver. The present Microware Source Level Debugger cannot be used to debug operating system components, although at the time of writing a system state source level debugger is under development.

In addition to the skeleton, the device driver writer will also need to write assembly language functions to permit the C code to make any necessary operating system calls. This applies to the privileged (system state only) calls, for which there are no functions in the Microware C library, but also to the non-privileged calls. This is because many of the standard C library functions assume that they are being called from a user state program, and use variables and buffers that are not appropriate to a device driver or other operating system component. However, the math functions (in the library '/dd/LIB/math.l') can be used, and indeed calls to these functions will automatically be generated by the C compiler.

The skeleton shown below takes no account of the use of floating point registers in an FPU. This is not necessary because the C compiler generates instructions at the start of each function to save any FPU registers it will use, and at the end of the function to restore them.

The C language specifies that static storage variables can be defined with initializing values. When a program is forked under OS-9, the kernel uses information in the module header to initialize any such variables in the program's static storage. However, the kernel does not perform such a function when allocating a Device Static Storage (or any other operating system memory structure), so initialized static storage variables cannot be used. Note that the kernel *does* clear the Device Static Storage to zeros before calling the initialization function of the device driver.

Because the kernel does not initialize any static storage variables, the "jump table" created by the linker for accessing subroutines at a relative distance exceeding +/-32k bytes cannot be used. To ensure that the linker has not found the need to generate a jump table, the '-a' option of the linker should not be used. If the device driver is larger than 32k bytes in size, the '-k2cl' option of the **cc** executive can be used, provided the target processor is a

68020/030/040, or the '-k0cl' option if the target processor is a 68000/010/070. Otherwise some manually-coded technique must be used to overcome this limitation of the 68000 processor.

□ RBF Device Driver Skeleton

```
* File 'rbskel.a'
* Device driver skeleton for RBF device drivers
            use     /dd/DEFS/oskdefs.d
Typ_Lang    set     (Drivr<<8)+Objct
Attr_Rev    set     ((ReEnt+SupStat)<<8)+0
Edition     set     1
* The psect directive, giving the module type as "device driver":
            psect   rbskel,Typ_Lang,Attr_Rev,Edition,0,EntryTable
* The table of offsets to the driver routines:
EntryTable  dc.w    Init                initialize
            dc.w    Read                read
            dc.w    Write               write
            dc.w    GetStat             get status
            dc.w    SetStat             set status
            dc.w    Term                terminate
            dc.w    0                   (exception handler)
****************************************
* Init
*   Initialize device driver
*
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int init(dd)
*           mod_dev *dd;                device descriptor ptr
*
Init:
            movea.w #0,a5               reset stack frame ptr
            move.l  a6,sysglobs(a2) save the System Globals ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6               copy the static storage ptr
            move.l  a1,d0               pass the device descriptor ptr as
*                                       the first (only) parameter
            bsr     init                call the C function
            move.l  d0,d1               copy returned error code
*                                       (0 => no error)
            beq.s   Init90              ..no error; carry is clear
            ori     #Carry,ccr          set carry to indicate error
Init90      rts
```

```
****************************************
* Read
*   Read sectors
*
* Passed:   d0.l = number of sectors to read
*           d2.l = start LSN
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int read(n,b)
*           unsigned int n,         number of sectors to read
*               b;                  first sector to read
*
Read:
            movea.w #0,a5           reset stack frame ptr
            move.l  a1,pathdesc(a2) save the Path Descriptor ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6           copy the static storage ptr
            move.l  d2,d1           pass start LSN as second parameter
* The following line is needed prior to OS-9 version 2.4:
            andi.l  #$000000ff,d0   make sector count a long word
            bsr     read            call the C function
            move.l  d0,d1           copy returned error code
*                                   (0 => no error)
            beq.s   Read90          ..no error; carry is clear
            ori     #Carry,ccr      set carry to indicate error
Read90
            rts
****************************************
* Write
*   Write sectors
*
* Passed:   d0.l = number of sectors to write
*           d2.l = start LSN
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int write(n,b)
*           unsigned int n,         number of sectors to write
*               b;                  first sector to write
*
```

```
Write:
            movea.w #0,a5           reset stack frame ptr
            move.l  a1,pathdesc(a2) save the Path Descriptor ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6           copy the static storage ptr
            move.l  d2,d1           pass start LSN as second parameter
            bsr     write           call the C function
            move.l  d0,d1           copy returned error code
*                                   (0 => no error)
            beq.s   Write90         ..no error; carry is clear
            ori     #Carry,ccr      set carry to indicate error
Write90
            rts
****************************************
* GetStat
*   Get Status wild card call
*
* Passed:   d0.w = function code
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int getstat(c,r)
*           unsigned int c;         function code
*           REGISTERS *r;           ptr to caller's stack frame
*
GetStat:
            move.l  a5,d1           pass caller's register stack
*                                   frame ptr as second parameter
            movea.w #0,a5           reset stack frame ptr
            move.l  a1,pathdesc(a2) save the Path Descriptor ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6           copy the static storage ptr
            andi.l  #$0000ffff,d0   make function code long
            bsr     getstat         call the C function
            move.l  d0,d1           copy returned error code
                                    (0 => no error)
            beq.s   GetStat90       ..no error; carry is clear
            ori     #Carry,ccr      set carry to indicate error
GetStat90
            rts
```

```
****************************************
* SetStat
*   Set Status wild card call
*
* Passed:   d0.w = function code
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int setstat(c,r)
*           unsigned int c;         function code
*           REGISTERS *r;           ptr to caller's stack frame
**
SetStat:
            move.l  a5,d1           pass caller's register stack
*                                   frame ptr as second parameter
            movea.w #0,a5           reset stack frame ptr
            move.l  a1,pathdesc(a2) save the Path Descriptor ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6           copy the static storage ptr
            andi.l  #$0000ffff,d0   make function code long
            bsr     setstat         call the C function
            move.l  d0,d1           copy returned error code
*                                   (0 => no error)
            beq.s   SetStat90       ..no error; carry is clear
            ori     #Carry,ccr      set carry to indicate error
SetStat90
            rts
****************************************
* Term
*   Terminate device driver
*
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int term(dd)
*           mod_dev *dd;            device descriptor ptr
*
Term:
            move.l  a6,-(a7)        save register
            movea.w #0,a5           reset stack frame ptr
            move.l  a4,procdesc(a2) save the Process Descriptor ptr
            move.l  a2,a6           copy the static storage ptr
            move.l  a1,d0           pass the device descriptor ptr
*                                   as the first (only) parameter
            bsr     term            call the C function
            move.l  d0,d1           copy returned error code
*                                   (0 => no error)
            beq.s   Term90          ..no error; carry is clear
            ori     #Carry,ccr      set carry to indicate error
Term90
            movea.l (a7)+,a6        retrieve register
            rts
****************************************
* IRQSvc
*   Interrupt service routine. It is assumed that the C 'init()'
* function has installed this routine as the interrupt handler, using
* the F$IRQ system call.
*
* Passed:   (a2) = Device Static Storage
*           (a3) = Port Address
*           (a6) = System Globals
*
* Returns:  carry set if the interrupt is not from our device
*
* Calls C function:
*           int irqsvc(port)
*           void *port;             port address
*
IRQSvc:
            move.l  a5,-(a7)        save register
            movea.w #0,a5           reset stack frame ptr
            move.l  a2,a6           copy the static storage ptr
            move.l  a3,d0           pass the port address as the
*                                   first (only) parameter
            bsr     irqsvc          call the C function
            tst.l   d0              was interrupt handled?
            beq.s   IRQSvc90        ..yes; carry is clear
            ori     #Carry,ccr      set carry to indicate not our
*                                   interrupt
IRQSvc90    movea.l (a7)+,a5        retrieve register
            rts
            ends
```

One or more separate C source files must be created, containing the code to perform the actual work of the device driver. The example below shows a "null" driver, compatible with the RBF driver skeleton shown above. Each function returns zero if there was no error, otherwise it returns the appropriate OS-9 error code. Note that the structure **rbfs (struct rbfstatic)** defines the kernel and file manager static storage, including the drive tables. This structure type is declared in 'DEFS/rbf.h'. Because this static storage is defined within the source code of the device driver, the static storage files such as 'LIB/drvs4.l' are not needed (and must not be used) at link time.

```
/* RBF device driver in C. Must be linked with 'rbskel.r'.
    Operating system definitions: */
#define RBF_MAXDRIVE 4          /* number of drives (required by
                                    'rbf.h') */
#include <errno.h>              /* error codes */
#include <modes.h>              /* file access modes */
#include <rbf.h>                /* RBF structures */
#include <MACHINE/reg.h>        /* register stack frame */
#include <procid.h>             /* process descriptor */
#include <sg_codes.h>           /* Get/Set status codes */
#include <path.h>               /* common path descriptor structure */
/* Functions in 'rbskel.a':
extern int IRQSvc();            /* interrupt handler skeleton */
/* Static storage definitions (for Device Static Storage): */
struct rbfstatic rbfs;          /* kernel and file manager static */
void *sysglobs;                 /* System Globals ptr */
Pathdesc pathdesc;              /* path descriptor ptr */
procid *procdesc;               /* process descriptor ptr */
int errno;                      /* general error number storage */
unsigned short irqmask;         /* status register image for masking
                                    interrupts */
/* Initialize */
int init(dd)
mod_dev *dd;                    /* pointer to device descriptor */
{
    rbfs.v_ndrv=RBF_MAXDRIVE;   /* set number of drives supported */
    irqmask=dd->_mirqlvl<<8 | 0x2000;   /* build status register image
                                    for masking interrupts */
    /* Install interrupt handler: */
    if (f_irq(dd->_mvector,dd->_mpriority,IRQSvc,rbfs.v_port)==ERROR)
        return(errno);          /* error - return error code */
    return(0);                  /* no error */
}
/* Read sectors */
int read(n,s)
unsigned int n,                 /* number of sectors to read */
    s;                          /* start LSN */
{
    register Rbfdrive dtb;      /* drive table ptr */
    register struct rbf_opt opts;   /* path descriptor options ptr */
    unsigned char *buffer;      /* ptr to buffer to read into */
    int drvnum;                 /* logical drive number */
    dtb=pathdesc->rbfpvt.pd_dtb;    /* get drive table ptr */
    buffer=pathdesc->path.pd_buf;   /* get buffer ptr */
    opts=&pathdesc->rbfopt;     /* point at path descriptor options */
    drvnum=opts->pd_drv;            /* get logical drive number */
    return(0);                          /* no error */
}
```

```
/* Write sectors */
int write(n,s)
unsigned int n,                 /* number of sectors to write */
    s;                          /* start LSN */
{
    register Rbfdrive dtb;      /* drive table ptr */
    register struct rbf_opt opts;   /* path descriptor options ptr */
    unsigned char *buffer;      /* ptr to buffer to write from */
    int drvnum;                 /* logical drive number */
    dtb=pathdesc->rbfpvt.pd_dtb;    /* get drive table ptr */
    buffer=pathdesc->path.pd_buf;   /* get buffer ptr */
    opts=&pathdesc->rbfopt;     /* point at path descriptor options */
    drvnum=opts->pd_drv;        /* get logical drive number */
    return(0);                  /* no error */
}
/* Get status */
int getstat(f,r)
int f,                          /* function code */
REGISTERS *r;               /* ptr to caller's register stack frame */
{
    register Rbfdrive dtb;      /* drive table ptr */
    register struct rbf_opt opts;   /* path descriptor options ptr */
    dtb=pathdesc->rbfpvt.pd_dtb;    /* get drive table ptr */
    opts=&pathdesc->rbfopt;     /* point at path descriptor options */
    switch (f) {                /* act according to function code */
        default:                /* unknown code */
            errno=E_UNKSVC;
            break;
    }
    return(errno);
}
/* Set status */
int setstat(f,r)
int f,                          /* function code */
REGISTERS *r;               /* ptr to caller's register stack frame */
{
    register Rbfdrive dtb;      /* drive table ptr */
    register struct rbf_opt opts;   /* path descriptor options ptr */
    dtb=pathdesc->rbfpvt.pd_dtb;    /* get drive table ptr */
    opts=&pathdesc->rbfopt;     /* point at path descriptor options */
    switch (f) {                /* act according to function code */
        case SS_WTrk:           /* format track */
            /* Use caller's parameters: */
            errno=format(r->d[2],r->d[3],r->d[4]);
            break;
        default:                /* unknown code */
            errno=E_UNKSVC;
            break;
    }
    return(errno);
}
```

```
/* Terminate */
int term(dd)
mod_dev *dd;                    /* pointer to device descriptor */
{
    /* Remove interrupt handler: */
    f_irq(dd->_mvector,0,NULL,NULL);
    return(0);                  /* no error */
}
/* Interrupt service routine */
int irqsvc(port)
void *port;                     /* interface chip address */
{
    return(0);                  /* interrupt successfully handled */
}
```

Normally a **make** file would be used to compile and link the source files to make the device driver. As an example, however, typical command lines are shown below:

```
$ r68 rbskel.a -q -o=RELS/rbskel.r
$ cc -qs rbdrv.c -r=RELS
$ l68 RELS/rbskel.r RELS/rbdrv.r -l=/dd/LIB/math.l
-l=/dd/LIB/sys.l -O=OBJS/rbdrv
```

15.13 A FILE MANAGER IN C

The principles described above for writing a device driver in C apply equally well to writing a file manager in C. In addition to receiving calls from the kernel – for which the parameter convention will always be the same – the skeleton must also provide one or more functions to call the device driver routines. The skeleton below has been written with a function (**CallDriver**) to pass a fixed set of parameters to the device driver. This function is suitable for file managers using the same driver calling conventions as used by SCF and RBF, and would also be suitable for any file manager being defined from scratch. Therefore this skeleton can be used without modification for almost any file manager.

The skeleton passes the address of the path descriptor in the **a6** register. This means that any static storage definitions within the file manager source files will be storage within the path descriptor. The file manager writer must take care that the total of such definitions – including the fields used by the kernel – does not exceed the 128 bytes available in the first half of the path descriptor.

The definitions file 'DEFS/path.h' declares a structure describing the path descriptor. It assumes that a file manager definitions file – such as 'DEFS/rbf.h' or 'DEFS/scf.h' – has already been read. To create a new file manager, the programmer should take a copy of 'DEFS/path.h', and edit it to include the structures declared for his file manager in a new file analogous to 'DEFS/rbf.h'. The "null" file manager shown after the skeleton uses the file 'DEFS/rbf.h', and so has the variables and options section defined by Microware for RBF.

The skeleton shown below includes a stack checking function **_stkcheck()**. The C source files can therefore be compiled with normal stack checking (that is, without the '-s' option).

□ File Manager Skeleton

```
* File 'fmskel.a'
* File manager skeleton
* Non-null psect giving a module type of "file manager":
Typ_Lang     equ      (FlMgr<<8)+Objct
Att_Revs     equ      ((ReEnt+SupStat)<<8)+0
             psect    fmskel,Typ_Lang,Att_Revs,Edition,0,fmskel
             use      /dd/DEFS/oskdefs.d
******************************
* Calling convention from kernel
*
* Passed:    (a1) = Path Descriptor
*            (a4) = Process Descriptor
*            (a5) = Caller's Register Stack Frame
*            (a6) = System Globals
*
* Returns: carry set if error, with error code in d1.w
*
* Destroys: may destroy ccr, d0-d7/a0-a4
******************************
******************************
* Calling convention to C routines
*
* The parameters passed to the C routine are:
*   Caller's Register Stack Frame ptr
* The C routine returns:
*   OS-9 error code if error, else 0
******************************
******************************
* Entry table
*
fmskel
             dc.w     Create-fmskel   create
             dc.w     Open-fmskel     open
             dc.w     MakDir-fmskel   make directory
             dc.w     ChgDir-fmskel   change directory
             dc.w     Delete-fmskel   delete
             dc.w     Seek-fmskel     seek
```

```
            dc.w    Read-fmskel     read
            dc.w    Write-fmskel    write
            dc.w    ReadLn-fmskel   read line
            dc.w    WriteLn-fmskel  write line
            dc.w    GetStat-fmskel  get status
            dc.w    SetStat-fmskel  set status
            dc.w    Close-fmskel    close
Create:
            lea     create(pc),a0   point at C routine
            bra.s   FMCall
Open:
            lea     open(pc),a0     point at C routine
            bra.s   FMCall
MakDir:
            lea     makdir(pc),a0   point at C routine
            bra.s   FMCall
ChgDir:
            lea     chgdir(pc),a0   point at C routine
            bra.s   FMCall
Delete:
            lea     delete(pc),a0   point at C routine
            bra.s   FMCall
Seek:
            lea     seek(pc),a0     point at C routine
            bra.s   FMCall
Read:
            lea     read(pc),a0     point at C routine
            bra.s   FMCall
Write:
            lea     write(pc),a0    point at C routine
            bra.s   FMCall
ReadLn:
            lea     readln(pc),a0   point at C routine
            bra.s   FMCall
WriteLn:
            lea     writeln(pc),a0  point at C routine
            bra.s   FMCall
GetStat:
            lea     getstat(pc),a0  point at C routine
            bra.s   FMCall
SetStat:
            lea     setstat(pc),a0  point at C routine
            bra.s   FMCall
Close:
            lea     close(pc),a0    point at C routine
* Fall through to FMCall
* Call the appropriate C function (function address is in a0)
FMCall
            move.l  a6,pd_sysglob(a1)   save System Globals ptr
            movem.l a5-a6,-(a7)     save registers
            move.l  a5,d0           pass caller's stack frame ptr
            movea.l a1,a6           copy Path Desc ptr as C static
            jsr     (a0)            call C routine
            move.l  d0,d1           copy error status
            beq.s   FMCall10        ..no error; leave carry flag clear
            ori     #Carry,ccr      set carry to indicate error
FMCall10    movem.l (a7)+,a5-a6     retrieve registers
            rts
******************************
* CallDriver
*  C-callable routine to call the device driver
*
* Passed:   d0.l = Offset to routine offset in entry table, e.g. D_READ
*           d1.l = First parameter for driver
*           4(a7) = Second parameter
*           8(a7) = Third parameter
*           12(a7) = Fourth parameter
*           (a6) = Path Descriptor
* Returns: 0 if carry clear from driver, else d1.w extended to int
* Destroys: ccr
*
* Passes:   d0-d3 = Parameters
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a5) = Caller's register stack frame
*           (a6) = System Globals
*
* Driver may destroy: ccr, d0-d7/a0-a6
*
CallDriver:
            movem.l d2-d7/a0-a6,-(a7)   save regs
            move.l  d0,d4           copy routine offset offset
            move.l  d1,d0           copy first parameter for driver
            movem.l 56(a7),d1-d3    get three more parameters
            movea.l a6,a1           copy Path Descriptor ptr
            movea.l PD_DEV(a1),a2   get Device Table
            movea.l V$DRIV(a2),a0   get address of driver module
            movea.l V$STAT(a2),a2   get Device Static Storage ptr
            movea.l PD_RGS(a1),a5   get caller's stack frame ptr
            movea.l PD_SysGlob(a1),a6   and System Globals ptr
            movea.l D_Proc(a6),a4   and Process Descriptor ptr
            add.l   M$Exec(a0),d4   get offset to routine offset table
            move.w  0(a0,d4.l),d4   get offset to routine
            jsr     0(a0,d4.w)      call driver routine
            bcc.s   CallD10         ..no error
            moveq   #0,d0
            move.w  d1,d0           copy error code as a long word
            bra.s   CallD20
CallD10     moveq   #0,d0           indicate no error
CallD20     movem.l (a7)+,d2-d7/a0-a6   retrieve registers
            rts
```

```
********************************
* _stkcheck
*   System state stack overflow check, assuming the system state stack
* is in the process descriptor.
* Hangs at '_stkerr' if stack overflow
*
* Passed:    (a6) = Path Descriptor
*            (a7) = stack
*
* Minimum stack leeway to insist on, in addition to C compiler default:
_stkmin:     equ      200
_stkcheck:   move.l   a0,-(a7)         save register
             movea.l  PD_SysGlob(a6),a0   get ptr to System Globals
             movea.l  D_Proc(a0),a0    get Process Descriptor ptr
             add.l    a7,d0            calculate desired new stack bottom
             subi.l   #_stkmin+P$Last,d0  with some margin
             cmp.l    a0,d0
             movea.l  (a7)+,a0         retrieve register
             bcs.s    _stkerr          ..not enough stack
             rts
_stkerr:
* Insufficient stack - loop forever:
             bra.s    _stkerr
             ends
```

Below is the corresponding C source code to make a "null" file manager.

```
/* RBF format file manager in C. Must be linked with 'fmskel.r'.
   Operating system definitions: */
#include <errno.h>                  /* error codes */
#include <modes.h>                  /* file access modes */
#include <rbf.h>                    /* RBF structures */
#include <MACHINE/reg.h>            /* register stack frame */
#include <procid.h>                 /* process descriptor */
#include <sg_codes.h>               /* Get/Set status codes */
#include <path.h>                   /* common path descriptor structure */

/* Functions in 'fmskel.a':
extern int CallDriver();            /* call the device driver */

/* Static storage definitions (for Path Descriptor): */
union pathdesc pd;                  /* path descriptor */
/* Create */
int create(r)
REGISTERS *r;                       /* caller's register stack frame */
{
    char *plist;                    /* ptr to pathlist */

    plist=r->a[0];                  /* get ptr to pathlist */
    return(0);                      /* no error */
}
```

```
/* Open */
int open(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    char *plist;                 /* ptr to pathlist */

    plist=r->a[0];               /* get ptr to pathlist */
    return(0);                   /* no error */
}
/* Make directory */
int makdir(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    char *plist;                 /* ptr to pathlist */

    plist=r->a[0];               /* get ptr to pathlist */
    return(0);                   /* no error */
}
/* Change directory */
int chgdir(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    char *plist;                 /* ptr to pathlist */

    plist=r->a[0];               /* get ptr to pathlist */
    return(0);                   /* no error */
}
/* Delete */
int delete(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    char *plist;                 /* ptr to pathlist */

    plist=r->a[0];               /* get ptr to pathlist */
    return(0);                   /* no error */
}
/* Seek */
int seek(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    unsigned int pos;            /* desired position */

    pos=r->d[1];                 /* get desired position */
    return(0);                   /* no error */
}
/* Read */
int read(r)
REGISTERS *r;                    /* caller's register stack frame */
{
    unsigned int n;              /* number of bytes to read */
    unsigned char *p;            /* buffer to read to */
```

```
        n=r->d[1];                  /* get number of bytes to read */
        p=r->a[0];                  /* get ptr to buffer */
        return(0);                  /* no error */
    }
    /* Write */
    int write(r)
    REGISTERS *r;                   /* caller's register stack frame */
    {
        unsigned int n;             /* number of bytes to write */
        unsigned char *p;           /* buffer to write from */

        n=r->d[1];                  /* get number of bytes to write */
        p=r->a[0];                  /* get ptr to buffer */
        return(0);                  /* no error */
    }
    /* Read line */
    int readln(r)
    REGISTERS *r;                   /* caller's register stack frame */
    {
        unsigned int n;             /* number of bytes to read */
        unsigned char *p;           /* buffer to read to */

        n=r->d[1];                  /* get number of bytes to read */
        p=r->a[0];                  /* get ptr to buffer */
        return(0);                  /* no error */
    }
    /* Write line */
    int writeln(r)
    REGISTERS *r;                   /* caller's register stack frame */
    {
        unsigned int n;             /* number of bytes to write */
        unsigned char *p;           /* buffer to write from */

        n=r->d[1];                  /* get number of bytes to write */
        p=r->a[0];                  /* get ptr to buffer */
        return(0);                  /* no error */
    }
    /* Get status */
    int getstat(r)
    REGISTERS *r;                   /* caller's register stack frame */
    {
        /* Call the driver's Get Status routine, passing the
            function code: */
        return(CallDriver(D_GSTA,r->d[1]));
    }
    /* Set status */
    int setstat(r)
    REGISTERS *r;                   /* caller's register stack frame */
    {
        /* Call the driver's Set Status routine, passing the
            function code: */
        return(CallDriver(D_PSTA,r->d[1]));
    }
    /* Close */
    int close(r)
    REGISTERS *r;                  /* caller's register stack frame */
    {
        return(0);                 /* no error */
    }
```

Normally a **make** file would be used to compile and link the source files to make the file manager. As an example, however, typical command lines are shown below. Note that the **cc** '-s' option is not used, because a stack checking function is provided in 'fmskel.a':

```
$ r68 fmskel.a -q -o=RELS/fmskel.r
$ cc -q rbmgr.c -r=RELS
$ l68 RELS/fmskel.r RELS/rbmgr.r -l=/dd/LIB/math.l
-l=/dd/LIB/sys.l -O=OBJS/rbmgr
```

15.14 HINTS ON C PROGRAMMING

C is a very powerful programming language, but it can also be a very confusing language. This section attempts to clarify some of the most common difficulties with C.

15.14.1 Declarations and Definitions

A **definition** is a statement that creates static storage or program code. A **declaration** is a statement that describes an object – a type, or a variable, or a function – without causing the compiler to generate any output. The syntax of definitions and declarations is very similar. For example, a declaration of a function simply omits the arguments and the compound statement body of the function that would make it a definition. A definition of a variable is converted to a declaration by preceding it with the keyword **extern**.

Understanding the syntax of declarations and definitions is one of the most common difficulties in C programming. In fact, once the principle is known, the process is very simple. In short, you should start with the symbol name, and work outwards using the standard order of precedence of the operators, as shown in the book "The C Programming Language", by Kernighan and Ritchie. The example below shows the technique. It is easiest if an appropriate phrase is used to replace each operator. The example shows the declaration of an array (it is a declaration rather than a definition, because the array size is not given), followed by a line for each operator, in the order of precedence, together with an appropriate phrase describing the operator:

```
        int *(*fred[])();
        fred                    /* fred is */
        fred[]                  /* an array of */
        *fred[]                 /* pointers to */
        (*fred[])()             /* functions returning */
        *(*fred[])()            /* pointers to */
        int *(*fred[])()int     /* int */
        /* fred is an array of pointers to functions returning pointers
            to int */
```

Exactly the same technique is used in reverse to create a desired definition or declaration. As with all expressions, parentheses are used to change the order of precedence of operators (as in the example above).

A *cast* operator is exactly the same as placing a declaration of an object of the desired type in parentheses, but omitting the object name. It causes the compiler to convert the value of an object of one type into the form of another type for use in an expression. For example:

```
        int (*p)();             /* 'p' is a pointer to a function returning
                                    an int */
        char *s;                /* 's' is a pointer to char */
        p=(int (*)())s;         /* cast 's' to the type of 'p', and copy it
                                    to 'p' */
```

15.14.2 Pointers and Arrays

The concept of a **pointer** does not appear in most programming languages, but it is essential in a language designed for writing operating system components. In C, a pointer variable contains a memory address, just as a processor address register does. A pointer variable can be assigned any memory address (even an illegal address which will cause a bus error when the pointer is used to access memory). However, a C pointer has an additional property – the type of object it points to. For example, the statement:

```
        int *jim;
```

defines a pointer that will be used to address objects of type **int**. Because the type of the objects pointed to is known to the compiler, the compiler can generate correct code when the pointer is used to read or write memory, and when arithmetic operations are used on the pointer. In the Microware 68000 C compiler, an **int** is a 32-bit long word. That is, it requires four bytes of storage. Thus **jim** is used to point to groups of four bytes:

```
    int *jim;
    int x,y;
    jim=(int *)0x00000000;        /* point at memory location zero */
    x=*jim;                       /* read the long word at address zero */
    y=jim[7];                    /* read the long word at address 28 */
    jim=jim+1;                   /* add the size of one integer to jim */
```

At the end of this sequence of instructions **jim** has the value 4.

The relationship between pointers and arrays is another very common source of confusion. Again, once the principle is known the confusion disappears. Whereas in most programming languages an array name refers to the whole of the storage used for the array, in C the array name is actually a *constant pointer* to the array storage – it does not refer to the storage itself. Therefore an array name is syntactically a pointer. Its type is a pointer to objects of the type of the array elements. For example:

```
    int fred[5];
```

Here **fred** is actually a constant pointer to objects of type **int**. Its "value" is the address of the storage allocated for five integers. To illustrate this, the following statement defines a variable of type pointer to **int**:

```
    int *jim;
```

If **jim** is to be used to point to the elements of the array **fred**, the programmer's instinctive reaction is to write:

```
    jim=&fred;
```

This is wrong (and syntactically illegal), as it asks the compiler to put into the variable **jim** the address of a constant, **fred**, and of course a constant has no address. In addition, there would be a type mismatch – the address of a pointer to **int** has type pointer to pointer to **int**, which does not match the type of **jim**. Instead, either of the two following statements can be used:

```
    jim=fred;
    jim=&fred[0];
```

The first statement asks the compiler to put the value of a constant pointer to **int**, **fred**, into the variable **jim**. The second statement asks the compiler to put the address of the first element pointed to by **fred** into the variable **jim**, which evaluates to the same thing (and indeed the Microware C compiler produces the same instructions). Conversely, because **fred** is a pointer to **int**, all pointer operations can be used on **fred**, except operations that attempt to change its value. The statements below show some examples where **fred** and **jim** are used to produce the same results (although the code from the compiler is different). Some illegal statements are also shown, to illustrate the constant nature of **fred**:

```
    jim=fred;   /* copy the constant value of fred to the variable jim */
```

The next four statements all assign the value 4 to the third element of the array:

```
    fred[2]=4;
    *(fred+2)=4;
    jim[2]=4;
    *(jim+2)=4;
```

The following statements illustrate the constant nature of an array name:

```
    fred=jim;           /* WRONG - can't assign a value to a constant */
    jim++;
    fred++;             /* WRONG - can't alter the value of a constant */
```

15.14.3 Pointers and Functions

The use of pointers to functions also often causes confusion. Pointers to functions can be very useful. They allow dynamic configuration of the functionality of a program. Pointers to functions are also useful when a program wishes to use a separate subroutine module – the program links to the subroutine module, and then (using a table of routine offsets at the start of the module) builds an array of pointers to the functions in the module.

A pointer to a function simply contains the memory address of the first instruction of the function. When a pointer to a function is used to call the function, the compiler generates code that loads the pointer into a processor address register, and performs a call to the subroutine at the address in the address register:

```
                movea.l funcptr(a6),a0
                jsr     (a0)
```

The type of a pointer to a function includes the type of the object returned by the function. For example:

```
    double (*mary)();
```

defines **mary** as a pointer to a function returning type **double**. This allows the compiler to make correct use of the value returned by the function when the pointer is used within an expression to call the function.

Just as with arrays and pointers, the type of a function name is exactly the same as the type of a pointer to the function. That is, the function name is a constant pointer to the function. Its value is the absolute memory address of the first instruction of the function. This means that function names and variables that are pointers to functions (returning objects of the same type) can syntactically be used interchangeably. The example below shows the declaration of a function, and the definition of a pointer to a function returning an object of the same type (pointer to **char**).

```
    char *fred();   /* function returning a pointer to char */
    char *(*jim)(); /* a pointer to such a function */
```

Just as with an array name, it is not possible to take the address of a function name, because it is a constant (a constant pointer to the function code):

```
jim=&fred;        /* WRONG */
```

Instead, to copy the address of the function to a variable (a pointer to a function), the function name alone is used:

```
jim=fred;
```

The constant value **fred** – the address of the function – is copied to the variable **jim**.

Because a function name is a constant pointer to the function, it has all the properties of a pointer, except that it cannot be modified. The example statements below show that once **fred** has been copied to the variable **jim**, the two can be used interchangeably to produce the same results, except that **fred** cannot be modified. In the example statements, **george** is an array of pointers to functions returning objects of type "pointer to **char**":

```
char *fred();     /* declare a function returning a pointer to char */
char *(*jim)(); /* a pointer to such a function */
char *(*george[5])();   /* an array of such pointers */
char *s;          /* 's' is a pointer to char */

jim=fred;         /* copy the address of the function */
george[3]=fred; /* copy the address of the function */
george[3]=jim;  /* exactly the same result */
s=fred();         /* call the function, and assign the result to 's' */
s=jim();          /* exactly the same effect */
s=george[3](); /* exactly the same effect */
```

A syntactic laxity in the original specification of C allows an alternative usage of a pointer to a function when calling the function:

```
char *fred();     /* declare function returning a pointer to char */
char *(*jim)(); /* a pointer to such a function */
char *s;          /* 's' is a pointer to char */

jim=fred;         /* copy the address of the function */
s=fred();         /* call the function, assign the result to 's' */
s=jim();          /* exactly the same effect */
s=(*jim)();       /* alternative syntax */
```

The alternative syntax effectively implies that the parentheses operator **()** operates on the function as a whole – the object pointed to by the pointer. This is clearly not the case – **fred** and **jim** have the same type, as one can be assigned to the other, so if **(*jim)()** is needed, then **(*fred)()** would also be required. Unhappily, it is this alternative syntax that was used in the first edition of Kernighan and Ritchie's book, so it is likely that some C compilers will not accept the first form. The reference section of later editions of K & R clearly shows that the first form is syntactically the correct form, but also states that in deference to older compilers the ANSI standard allows both forms. The Microware C compiler accepts both forms.

15.14.4 Pointers and Structures

Finally, structures and pointers to structures have often caused confusion. The syntax of C considers a structure to be a data object, exactly the same as a simple data object such as an **int** or a **double**. The structure name refers to the whole of the object. This is very different from an array, where the array name is a constant pointer to the first element of the array. The example below shows the declaration of a structure type, followed by the definition of a structure object of that type, and a pointer to such an object.

```
typedef struct {          /* declare the structure type */
    int item_1;
    char *item_2;
} mystruct;               /* the new type is 'mystruct' */
mystruct fred;            /* 'fred' is an object of that type */
mystruct *jim;            /* 'jim' is a pointer to such an object */
```

The assignment operator '=' will copy the whole of the structure. The structure name refers to the whole of the structure, not the address of the structure:

```
jim=fred;                 /* WRONG */
jim=&fred;                /* point at the structure */
```

jim now contains the address of the first byte of the structure. Because the type of the object pointed to by **jim** is known to the compiler, the object pointed to by **jim** can be used in expressions, and simple arithmetic can be used on **jim**:

```
mystruct george;          /* define a new object of that type */
george=fred;              /* copy the structure as a whole */
george=*jim;              /* same effect */
jim=jim+1;                /* add the size of one structure to 'jim' */
```

The elements of the structure can be accessed using both the structure name, and the pointer to the structure. However, the operators used are different:

```
fred.item_1=5;            /* set the first element to the value 5 */
jim->item_1=5;            /* same effect - different operator */
```

The declaration and definition of structures also causes some confusion. A simple structure type declaration has the following syntax:

```
struct henry {            /* declare the structure type */
    int item_1;
    char *item_2;
};
```

This is a declaration of an object type **struct henry** – no storage is reserved at this time. The symbol **henry** is known as the structure tag – it identifies the structure type in subsequent statements. The structure type can be used to define objects:

```
struct henry x;          /* 'x' is a structure */
```

The C language specification allows the two statements to be merged together:

```
struct henry {           /* declare the structure type */
    int item_1;
    char *item_2;
} x,y,z;                 /* 'x', 'y', and 'z' are structures */

struct henry a,b,c;      /* and so are 'a', 'b', and 'c' */
```

The example above merges the declaration of the structure type **struct henry** with the definition of three structures of that type, **x**, **y**, and **z**. The second statement shows that the declared structure tag can still be used to define further objects of that type – **a**, **b**, and **c**. The structure tag can be omitted if only the merged form of the statement is to be used (further structures of that type cannot be defined):

```
struct {
    int item_1;
    char *item_2;
} x,y,z;                 /* 'x', 'y', and 'z' are structures */
```

As with any definition, a structure definition can be converted to the declaration of a new object type by prefixing it with the **typedef** keyword. Thus:

```
struct henry {
    int item_1;
    char *item_2;
} george;
```

declares a structure type **struct henry**, and defines a data object **george** which is a structure. Storage is reserved for **george**. However:

```
typedef struct henry {
    int item_1;
    char *item_2;
} george;
```

declares a new type **george**. No storage is reserved at this time. Objects can be defined with that type (or by using the structure tag, here **henry**):

```
george a,b,c;            /* 'a', 'b', and 'c' are structures */
struct henry x,y,z;      /* 'x', 'y', and 'z' are of the same type */
```

The structure tag can be omitted:

```
typedef struct {
    int item_1;
    char *item_2;
} george;
george a,b,c;            /* 'a', 'b', and 'c' are structures */
```

However, if the structure is to contain pointers to other structures of the same type (for example, in a linked list), the structure tag is necessary, as the compiler will not see the type name until it has reached the end of the type definition, and C is designed to be compiled in a single pass:

```
typedef struct {
    int item_1;
    george *item_2;              /* WRONG */
} george;

typedef struct henry {
    int item_1;
    struct henry *item_2;        /* correct */
} george;
```

Note that in the above example **struct henry** and **george** are the same type, and can be used interchangeably.