CHAPTER 6

C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER

This chapter is an overview of the Professional OS-9 program development utilities provided with OS-9 version 2.4 and version 3.2 of the Microware C compiler.

6.1 THE DEVELOPMENT SYSTEM

The development process consists of:

a) Edit the program source file(s).
b) Compile and/or assemble to Relocatable Object File(s) (ROFs).
c) Link ROFs to form a program module.
d) Test the program, using a debugger.
e) Repeat the cycle until the program works.

The development tools consist of utilities to facilitate and help manage this process. The development tools provided with Professional OS-9 are:

cc	C compiler executive
cpp	C preprocessor
c68	C compiler (68000 and 68010)

c68020	C compiler (68020, 68030, and 68040)
o68	assembly language optimizer
r68	assembler (68000 and 68010)
r68020	assembler (68020, 68030 and 68040)
l68	linker
make	automatic compilation manager
debug	assembly–level symbolic debugger

Also available is **sysdbg**, the system state debugger for debugging system state processes, operating system components and multi–tasking applications, and **srcdbg**, the C source level debugger.

The C executive **cc** and **make** are utilities to facilitate the use of the compiler, assembler, and linker, especially in projects with multiple source files.

6.2 THE C COMPILER

The compiler has three phases: preprocessing (**cpp**), compilation (**c68** or **c68020**), and optimization (**o68**).

The preprocessor performs the standard C preprocessing functions – all the lines that start with the '#' character. It produces a temporary file with macros expanded, "include" files inserted, and conditional compilation resolved, ready for compilation.

The compiler translates the C program into assembly language output. (Assembly language is the machine code instruction language for the processor, but in symbolic form). The **cc** executive program has an option ('–a') to halt compilation at this stage. This allows the programmer to see exactly what the compiler has done with his program – particularly important when trying to optimize a critical fragment of the program.

The optimizer looks through the assembly language for common instruction sequences that can be made more efficient. It may change instructions, or even alter the order of instructions.

These phases are not normally called directly by the user. The **cc** executive performs the task of calling each of the phases with appropriate parameters and options, and of creating and deleting all necessary temporary files. **cc** also calls the assembler and linker to produce a finished program module, normally in the execution directory, ready for testing. **cc** takes a command line option ('−r') to halt compilation after assembly, once a Relocatable Object File (ROF) has been produced (by the assembler). This allows the user to build programs from multiple source files by specifying the linker command line manually, usually within a script file for the **make** utility.

6.3 FILE NAMING CONVENTIONS

cc and **make** use certain conventions for filename extensions. Each extension is a period ('.') followed by a single character. The filename preceding the extension is called the "root". When **cc** and **make** create file names from a given filename, they use the root of the given filename, plus the appropriate extension. A file with no extension in its name is taken to contain an executable program module.

Extension	File contents
.c	C source file
.a	assembly language source file
.r	Relocatable Object File (ROF) − output of assembler
none	executable module − output of linker

The following conventions are also used, although neither **cc** nor **make** recognize them:

Extension	File contents
.h	C definitions source file (used with **#include** preprocessor directive)
.d	assembly language definitions source file (used with **use** directive)
.m	assembly language macro definitions source file (used with **use** directive)
.l	linker library − one or more ROFs in one file

The extension conventions '.c' and '.h' are also recognized by the **umacs** screen editor (it automatically switches on its 'CMODE' mode).

6.4 CC OPTIONS

cc has several options to provide control of the compilation, assembly, and linking phases. Some of the options control which phases will be executed, but most of the options are passed as options to the appropriate phase program. In the following descriptions, the Microware standard notation for command line syntax is used.

Option	Affects	Explanation
-a	cc	Suppress assembly and linking - leave the assembly language in a '.a' file.
-bg	l68	Make the output module a "sticky" module.
-bp	cc	Display phase command lines in full (useful when debugging **make** files).
-c	c68	Copy C source code as comments to compiler output assembly language file. Use this option with '-a' to help interpret the output of the C compiler.
-d<str>	cpp	Define a symbol - equivalent to **#define <str>**. Examples: -dFRED #define FRED -dFRED=4 #define FRED 4
-e=<n>	l68	Set output module edition number (default is 1).
-f=<path>	l68	Set output object file name (default is the root part of the source file name for a single file linkage, or 'output' for a multi-file linkage). Output pathlist is relative to the current execution directory.
-fd=<path>	l68	Same as '-f', but output pathlist is relative to the current data directory. Note that the file will not have the execute permissions set.
-g	c68, r68 and l68	Generate symbol table module files with extensions '.dbg' and '.stb' for the symbolic debuggers. These files will go to a directory 'STB' if one exists within the directory to which the output module is directed, otherwise to the same directory as the output module.
-i	cc	Use calls to the **cio** trap handler for the common I/O functions, rather than including the functions in the program module (significantly reduces the size of small programs).
-j	l68	Do not produce an indirect jump table for long function calls. This suppresses the generation by the linker of an indirect jump table in the program's static storage for function calls where a relative branch offset of more than 16 bits is required. This

Option	Affects	Explanation
		option is not normally used, as a jump table is not created unless needed. Use it if you want to know if a jump table is needed - the linker will report a problem.
-k=[<n>][W¦L] [CW¦CL][F]	c68020	Specify processor-dependent compilation. This option enables the compiler generation of extended instruction sequences for 32 bit offsets to static storage and functions, and of 68020/030/040 additional instructions and addressing modes (such programs cannot run on a 68000 or 68010 processor): n=0 for 68000/010. n=2 for 68020/030/040 (allows the use of additional addressing modes and instructions, including long integer division). W indicates that word (16 bit) constant offset indexing be used for data references (default). L indicates that long word (32 bit) constant offset indexing be used for static storage references. Use this option if the program static storage exceeds 64k bytes. It will produce a larger and slightly slower program. For a 68000/010 the compiler will generate two instructions for every static storage reference. CW indicates that word (16 bit) constant offset indexing be used for program references and function calls (default). CL indicates that long word (32 bit) constant offset indexing be used for program references and function calls. Use this option if a program larger than 64k bytes is being generated, *and* string literals are being referenced at offsets greater than 32k, or the indirect jump table approach is considered slow (each long call takes two instructions through the jump table). This option will produce a larger and (for short calls) slightly slower program. For a 68000/010 the compiler will generate two instructions for every function call or string literal reference. F indicates that in-line FPU instructions be used for maths, rather than function calls. Such programs can only be used with a 68881 or 68882 co-processor, or with a 68040 processor, but floating point maths will be very much faster.
-l=<path>	l68	Specifies an additional library. The library will be searched before any of the default libraries. If multiple '-l' options are used the libraries will be searched in the order of the '-l' options.

Option	Affects	Explanation
-m=<n>	l68	Add to linker default stack allocation of 3k (units are k bytes).
-n=<name>	l68	Set output object module name - default is same as output object file name.
-o	cc	Exclude optimization phase.
-q	cc	"Quiet" mode - don't display phase execution messages.
-r	cc	Don't execute linker phase - leave output in '.r' Relocatable Object File(s).
-s	c68	Omit stack checking code. The compiler normally generates a call to the subroutine **_stkcheck** at the beginning of each function, to check for stack overflow. This subroutine is contained in 'cstart.r', and is suitable for programs, but not (for example) for device drivers.
-t=<dir>	cc	Specify directory for temporary files - default is current data directory. This could be a RAM disk ('/r0'), to speed up compilations.
-u<str>	cpp	Undefine a symbol - cancels a preceding '-d' option.
-v=<dir>	cpp	Specify an additional directory to search for **#include** files. The additional directory is searched before the default ('/dd/DEFS'). If multiple '-v' options are used, the directories are searched in the order of the options.
-w=<dir>	cc	Specify directory for implicit library files - default is '/dd/LIB'.
-x	cc and c68	Use calls to the **math** trap handler for floating point (and difficult integer) maths, rather than including the maths subroutines in the program.

For example:

```
$ cc -qix test.c
```

The '-i' and '-x' options control which libraries **cc** automatically specifies to the linker (**l68**). These libraries are taken from the directory '/dd/LIB', unless the '-w' option is used to indicate a different directory to search:

Options	Libraries specified
none	clibn.l, math.l, sys.l
x	clib.l, sys.l
i	cio.l, clibn.l, math.l, sys.l
ix	cio.l, clib.l, sys.l

cc automatically specifies 'cstart.r' (also from the default libraries directory) as the first ROF on the command line to the linker. For example, the **cc** command:

```
$ cc -q -bp -ix test.c
```

will (after compilation) produce the linker command line:

```
l68 /dd/LIB/cstart.r ctmp.000006 -o=test
-l=/dd/LIB/cio.l -l=/dd/LIB/clib.l -l=/dd/LIB/sys.l
```

'cstart.r' contains the ROF for the startup code for a C program. It is a "root psect" produced from the file 'cstart.a', and must be the first ROF in the linking of a C program. It is *not* appropriate for trap handlers, file managers, device drivers, and other executable modules, for which the programmer must produce (in assembly language) a substitute for 'cstart.a'.

6.5 THE ASSEMBLER

This section describes the Microware assemblers **r68** and **r68020**. It assumes that the reader is already familiar with the 68000 instruction set, and with Motorola-type assemblers. Here the aim is to highlight the special features of the Microware assemblers.

The Microware assembler is a full macro assembler. Two versions are available. **r68** is the standard 68000/010 assembler, while **r68020** supports the additional instructions and addressing modes of the 68020/030/040, plus the coprocessor instructions for the 68881/2 FPU, the 68851 MMU, and the built-in MMUs of the 68030/040.

The assembler contains special functions to help in the production of OS-9 modules, and for use by the C compiler. The syntax of the assembler is Microware's own. The instruction and addressing mode syntax is Motorola standard, but the directives and pseudo-instructions are not compatible with other 68000 assemblers.

The output of the assembler is a Relocatable Object File (ROF). A ROF contains the object code, plus symbolic information, and information required by the linker to allow multiple ROFs to be linked into one object module. In particular, the ROF contains tables identifying code and data offsets within instructions, so that the linker can resolve these at link time into offsets relative to the start of the module and of the static storage. Note that the assembler does *not* produce an output ROF unless the '-o' option is used, to specify the ROF pathlist.

All OS-9 object code is position-independent, and uses address register indirect addressing for data accesses. Therefore the assembler does not provide special functions for the management of absolute-addressed programs.

Symbol names may be of any length. All characters are significant, and letter case is significant. Case is not significant in opcode mnemonics (but it is in user-defined macro names).

6.5.1 The psect Directive

The **psect** directive indicates the start of the program code segment of a source file. The **ends** directive indicates the end. Only one **psect** is allowed in a source file. Code-generating instructions are not allowed outside of the program segment. Therefore the **psect** directive is normally one of the first instructions in a source file, and the **ends** directive is usually the last instruction.

The purpose of the **psect** directive is to supply information in the output ROF used by the linker in producing the output module header. The **psect** directive is essentially a pre-defined macro. The syntax of the directive is:

```
psect name,type_lang,att_revs,edition,stacksize,entrypoint,trapentry
```

Parameter	Description
name	psect name - commonly the same as the file name.
type_lang	Output module type and language (word).
att_revs	Output module attributes and revision number (word).
edition	Output module edition number.
stacksize	Stack estimation (zero to use linker default).
entrypoint	Offset to the program execution entry point.
trapentry	Offset to the routine to call for uninitialized trap instructions.

"trapentry" must be omitted if the program does not have a routine to handle uninitialized **TRAP** #n instructions. The offsets to the execution entry point and uninitialized trap instruction handler are relative to the beginning of the psect. At link time the linker adjusts these values to be relative to the start of the module header.

If multiple ROFs are to be linked to form an output module, only one may have a non-zero "type_lang" and "att_revs". This is known as the "root psect" (or "non-null psect"). The type, language, attributes, revision, and edition of the root psect determine those of the output module. The C compiler always produces non-root (null) psects. These are then linked with 'cstart.r', which contains a root psect.

The linker uses the execution offset defined in the first psect on the linker command line as the value to put in the module header execution offset location. Therefore the first ROF on the command line normally contains the root psect. In the case of C programs, 'cstart.r' *must* be the first ROF on the command line, as it provides the initialization function for the C program, which calls the **main()** function.

The following example **psect** uses symbolic names defined in the header file 'DEFS/oskdefs.d' supplied by Microware. This appears to contradict the usual Microware technique of supplying symbolic names in pre-assembled libraries (such as 'LIB/usr.l' and 'LIB/sys.l'). The problem is that the assembler and linker only allow simple addition and subtraction of symbols that are not known at assembly time (external references that will be resolved from a library at link time), and - as can be seen in the example - the "shift left" operator '<<' is frequently used with the **psect** directive.

```
* Program to print a string
          use     /dd/DEFS/oskdefs.d
typelang  equ     (Prgrm<<8)+Objct
attrevs   equ     (ReEnt<<8)+0
edition   equ     1
stacksize equ     1000
          psect   fred,typelang,attrevs,edition,stacksize,progstart
progstart lea     string(pc),a0  point at string to print
          moveq   #1,d0          print to standard out
          moveq   #strlen,d1     string length
          os9     I$WritLn       print the string
          os9     F$Exit         and exit
string    dc.b    "hello world",13
strlen    equ     *-string
          ends
```

6.5.2 The vsect Directive

The **vsect** directive creates static storage segments. Within a vsect, pseudo-instructions are used to reserve static storage and assign symbolic names to static storage locations. Again, the **ends** directive indicates the end of the segment. Any number of vsect segments may appear in a source file, but they must all lie inside the psect. The linker adds up the size of the vsects in multiple ROFs to determine the total static storage required by the program, and to adjust static storage references in program instructions.

The 'ds' directive is used to define uninitialized static storage. The extensions '.b', '.w', and '.l' are used to indicate byte, word, and long word locations respectively. For example:

```
            vsect
fred        ds.l    1                   one long word
henry       ds.w    1                   one word
jim         ds.b    20                  20 bytes
            ends
```

The assembler ensures that word and long word fields are word-aligned (that is, they are on an even address).

The 'dc' directive can be used to define initialized storage, in the same way that it is used to define constant data within a program:

```
            vsect
george      ds.l    2                   two long words (not initialized)
percy       dc.l    2                   one long word initialized to 2
            ends
```

6.5.3 External Symbols

In a project with multiple source files it is likely that some symbols defined in one source file will be used in one or more other source files, and that symbols defined in libraries will be used in program files. (Note: OS-9 libraries are simply ordinary ROFs merged together).

r68 generates an external reference in the ROF if it encounters a reference to a symbol not defined within the source file.

r68 generates a public declaration in the ROF if a symbol is defined with a terminating colon:

```
fred:       equ     36
```

or

```
henry:      moveq   #0,d0
```

If a symbol is not defined with a terminating colon it is "private" to the source file. It will not appear as a symbol in the ROF, and so cannot conflict with an identical name defined in another source file, even if the other name is defined as public. The C compiler produces public definitions for all objects defined at the outermost scope (functions and static storage), unless the definition is preceded by the **static** keyword, in which case a private definition is generated. Private definitions are generated for all storage defined within functions.

The linker, if requested by the '-g' option, records all public definitions in a separate "symbol table" module, with the extension '.stb'. The symbolic debuggers (**ROMbug**, **debug**, the system state debugger **sysdbg**, and the C source level debugger **srcdbg**) can link to this module to permit the use of symbolic names in debugger commands and expressions (**srcdbg** reads the '.stb' file rather than linking to the module). Symbols defined privately are not known to the debuggers, except to **srcdbg**, which uses the '.dbg' file produced by the C compiler.

The assembler and linker do not permit complex expressions containing external references. Such expressions are limited to adding or subtracting the external symbol. There are also limitations in the use of external symbols in the definitions of other symbols, and of course external symbols cannot be used in conditional assembly statements.

6.6 THE LINKER

The OS-9 linker **l68** is not complex in operation. It takes one or more ROFs and links them to produce an OS-9 object module. One (and only one) ROF must contain a root psect - normally the first ROF. ROFs need not contain object code. For example, they may consist only of public symbol definitions, or static storage definitions. ROFs may be supplied in two ways:

a) In a file whose name is given as a command line parameter (the file can contain multiple ROFs merged together).

b) In a library file whose name is given by the '-l' option.

A library is simply one or more ROFs merged together:

```
$ merge rofone.r roftwo.r rofthree.r >mylib.l
```

The linker will include in the output object module all the ROFs specified as command line parameters, plus any ROFs in the libraries required to satisfy external symbol references. ROFs are linked in the order in which they appear on the command line. ROFs in libraries (specified with the '-l' option) are linked after all ROFs not in libraries.

As the linker reads each ROF it attempts to resolve any external references in the ROF from public symbols defined in earlier ROFs. External references that cannot be resolved are added to a table of outstanding references. Therefore once the linker has read all the ROFs specified as command line parameters, it has built a table of outstanding external references that must be satisfied from the ROFs in the libraries.

The linker only scans the libraries once, in the order they are given on the command line. It discards library ROFs that do not satisfy currently outstanding external references. The public symbols defined in a discarded library ROF are also discarded. Therefore it is important to avoid backward references to earlier library ROFs within library ROFs. If a ROF satisfies an outstanding external reference, the whole psect in the ROF (including any vsects within the psect) is added to the output module, and all public symbols defined in the ROF are added to the table of public symbols. The '-l' option of the **rdump** utility can be used to check that a library does not have any backward references within it:

```
    $ rdump -l mylib.l
```

rdump will report any backward references within the library 'mylib.l'.

The linker recognizes a special symbol **_sysedit** to set the module edition number, overriding the entry in the root psect. This can be used to set the edition number from within a C source file. The example below uses the '@' character to introduce a single line of assembly language in a C source file:

```
    #include <stdio.h>
    #include <errno.h>
    @_sysedit:  equ    3               edition number
```

6.6.1 Linker Options

The linker has several command line options, of which the most important are shown below. Note that the case of the option letter is significant:

-a	Generate jump table in static storage for function calls with offsets greater than 32k.
-e=<n>	Set output module edition number – overrides edition number in root psect.
-g	Output '.stb' symbol module for symbolic debugging.
-j	Print jump table information (see '-a').
-l=<path>	Specifies a library file.
-m	Print linkage map (values of all public symbols) to standard output.
-M=<n>	Specify addition to output module stack size in k bytes. The linker accepts but ignores a negative value. The default stack size is 3k bytes.

-n=<name>	Set output module name (default is name of output file).
-o=<path>	Specify output file, relative to the current execution directory.
-O=<path>	Specify output file, relative to the current data directory. Note that the file will not have execute permissions set (use the **attr** utility after linking to set the execute permissions).
-p=<n>	Set module header permissions word (in hexadecimal). For example, '-p=777' sets read, write and execute permissions for public, group, and owner.
-r[=<n>]	Generate output without module header or CRC, with absolute addressing relative to address <n> (default 0) in hexadecimal. This option is used to generate boot ROMs, for example.
-s	Print symbol table to standard output.
-S	Make output a sticky module.
-w	Sort printed symbol table alphabetically, rather than by order of value (used with '-s').
-z=<path>	Get list of ROFs from a file (or from standard input, if no pathlist is given), instead of from the command line.

Example:

```
   $ l68 first.r second.r -l=/dd/LIB/sys.l -o=prog -msw
```

The linker has no interactive features, such as defining symbols at link time.

6.7 THE PROGRAM DEBUGGER

The Microware program debugger **debug** is an assembly-level symbolic debugger. It debugs user-state programs, using the special "debug process" system calls provided by the operating system. The process being debugged exists in its own right, with all the normal facilities and resources of an ordinary process. The difference is that it is not run until the parent (**debug**) makes the appropriate system call, and the parent can install breakpoints (using a system call).

debug automatically attempts to link to or load a symbol table module with the same name as the program module, and the extension '.stb'. It searches first the current execution directory, and then each directory specified in the **PATH** environment variable. For each directory it first searches the 'STB' subdirectory, and then the directory itself. Once the '.stb' file has been found, all publicly declared symbols can be displayed and referenced by name. If **debug** cannot find a '.stb' file for the program it reports an error, but does not abort.

debug also attempts to find a '.stb' file for each trap handler module the program links to. For example, programs generated with the '−i' option to the C compiler use the **cio** trap handler. **debug** reports that no '.stb' file can be found for **cio** when the first **cio** trap call is made by the program.

The debugger provides:

- Program breakpoints.
- Inspection/modification of memory.
- Inspection/modification of processor registers.
- Disassembly of memory.
- Forking the program to be debugged.
- Linking to OS−9 modules.
- Controlled program execution.

The debugger considerably simplifies memory and program inspection by providing an extended set of operators for expressions. For example:

```
dbg: d [.a0+6]+.d0 20
```

means "display $20 bytes from the address calculated by taking the address stored at the location given by the **a0** register plus 6, and adding the **d0** register". Also, wherever constants are allowed, symbolic references may be used.

Because the 'l' command of **debug** allows linking to any module, **debug** can be used to inspect or modify any module in memory. For example, temporary patches can be made to device descriptors. Such patched modules can be saved to disk (using the **save** utility), and the CRC and header parity can be corrected in the saved file using the **fixmod** utility.

The 'l' command causes **debug** to link to the named module. **debug** puts the address of the module in relocation register 7, known as '.r7'. (**debug** maintains eight relocation registers, which are logically software extensions to the processor registers). Any relocation register can be named as the default base address to use for display and disassembly, using the '@' command. This is very convenient for inspecting and modifying modules. For example:

```
dbg: l term
dbg: @7
dbg: c 50
0x00000050+r7:18 19
0x00000051+r7:08 .
dbg:
```

This example links to the device descriptor **term** (the usual name for the first serial port), and sets relocation register 7 as the default base address. Location $50 within the module is inspected and modified. In the case of an **SCF** device descriptor, this has changed the "lines per page" entry from 24 to 25. Of course, this particular operation can be carried out much more simply using the **xmode** utility:

```
$ xmode /term pag=25
```

Note that if the system is using the System Security Module (SSM) for inter–task memory protection, the device descriptor module must have write permission in the module permissions field of its module header. If not, **debug** and **xmode** will generate a bus error (error number 102 – **E$BusErr**) when trying to write to the module. By default the linker does not set write permissions when creating a module. The **fixmod** utility can be used to change the permissions of a module in a file. For example:

```
$ fixmod dd.d0 -up=777
```

sets the read, write, and execute permissions for private, group, and public access in the module in the file 'dd.d0'.

debug has two ways of running the program. The first uses kernel-controlled single stepping ("tracing") through the program. The kernel maintains a list of breakpoints for the process being debugged. It executes the program instruction by instruction, until a breakpoint is hit or the program exits. This is slow, but allows breakpoints to be set even if the program is in ROM. The second method ('x –1' command) runs the program at full speed – breakpoints are put into the program code as illegal instructions. When the program hits an illegal instruction the kernel stops the program and wakes up the debugger.

The C source level debugger **srcdbg** uses exactly the same approach as **debug** (except that it reads the '.stb' file, rather than linking to the '.stb' module). In addition, it uses the information in the '.dbg' file generated by the C compiler, assembler, and linker to associate the machine code program counter with the C source file. This allows the user to view and step through the program at the C source level, and to view and change C variables. Note that the '.dbg' files are not loadable modules.