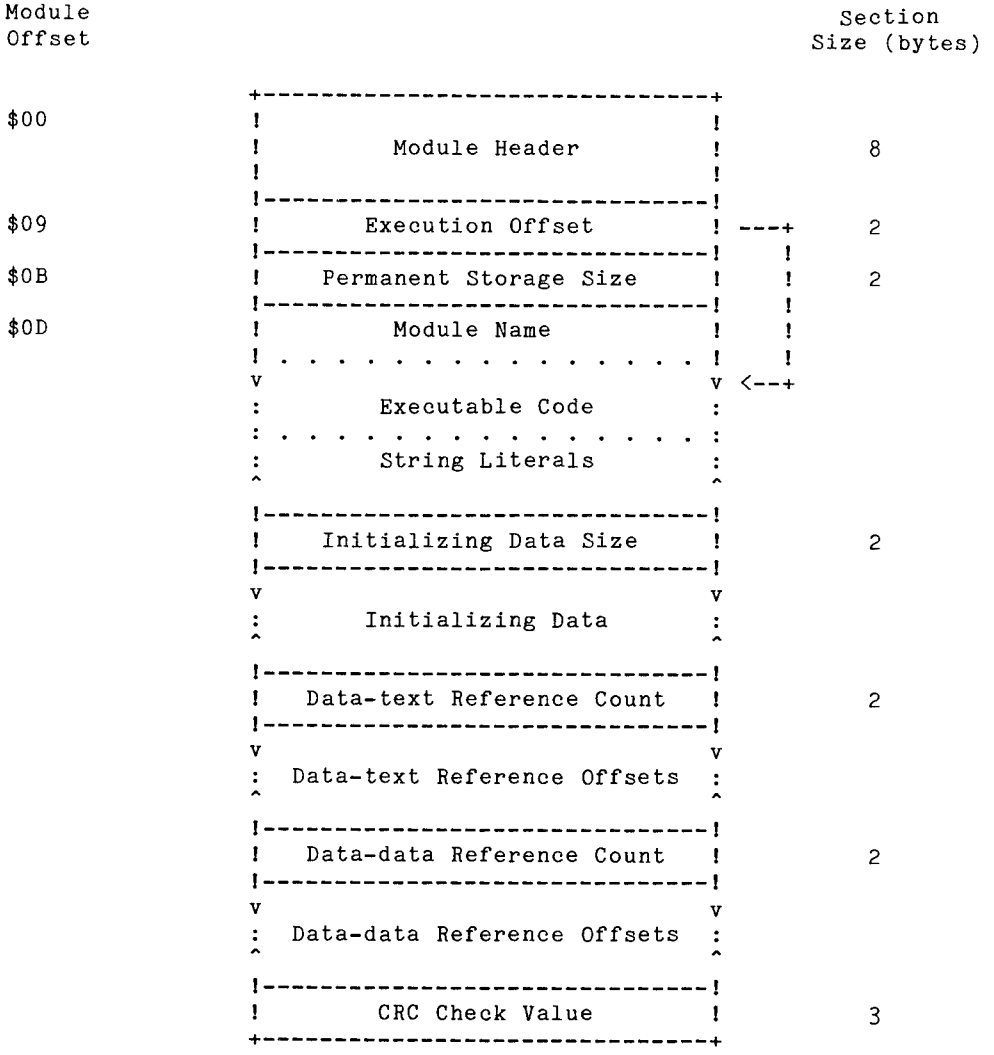
The Object Code Module

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



Module Header

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

Execution Offset

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

Storage Size

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

Module Name

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

Information

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

Executable Code

The machine code instructions of the program.

String Literals

Quoted strings in the C source are placed here. They are in the null-terminated form expected by the functions in the Standard Library. NOTE: the definition of the C language assumes that strings are in the DATA area and are therefore subject to alteration without making the program non-reentrant. However, in order to avoid the duplication of memory requirements which would be necessary if they were to be in the data area, they are placed in the TEXT (executable) section of the module. Putting the strings in the

executable section implies that no attempt should be made by a C programmer to alter string literals. They should be copied out first. The exception that proves the rule is the initialization of an array of type char like this:

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

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

Initializing Data and its Size

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

Data References

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

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

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

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

The startup routine will first copy all the entries in the initializing data section into their allotted places in the DATA section.

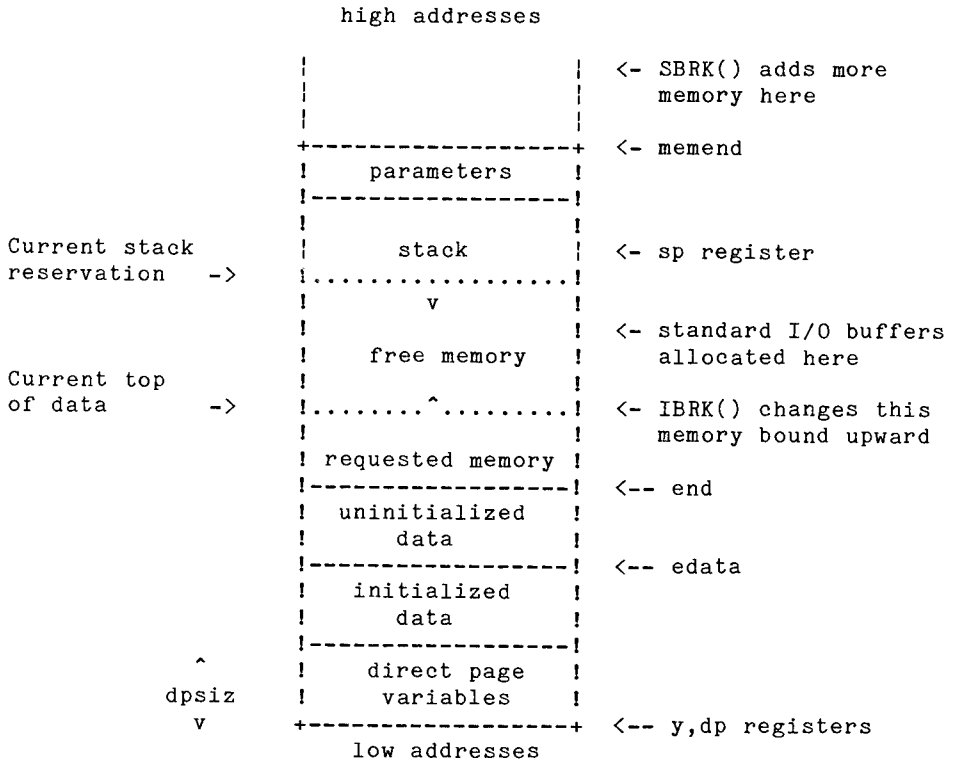
Then it will scan the data-text reference table for the offsets of values that need to have the addresses of the base of the TEXR section added to them. Among these will be "p" which, after updating, will point to the string which is in the TEXT section. Similarly, after a scan of the data-data references, "q" will point to(contain the absolute of) "p".

MEMORY MANAGEMENT

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

Typical C Program Memory Map

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



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

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

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

**** STACK OVERFLOW ****

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

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

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

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

and in assembler:

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

The Y register points to the base of the data area and variables are

addresses using Y-offset indexed instructions.

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

Compile Time Memory Allocation

If not instructed otherwise, the linker will automatically allocate 1k bytes more than the total size of the program's variables and strings. This size will normally be adequate to cover the parameter area, stack requirements, and Standard Library file buffers. The allocation size may be altered when using the compiler by using the "-m" option on the command line. The memory requirements may be stated in pages, for example,

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

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

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

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

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

1. The parameter area should be large enough for any anticipated command line string.

2. The stack should be not less than 128 bytes and should take into account the depth of function calling chains and any recursion.

3. All function arguments and local variables occupy stack space and each function entered needs 4 bytes more for the return address and temporary storage of the calling function's register variable.

4. Free memory is requested by the Standard Library I/O functions for buffers at the rate of 256 bytes per accessed file. This does not apply to the lower level service request I/O functions such as "open()", "read()" or "write()" nor to "stderr" which is always un-buffered, but it does apply to both "stdin" and "stdout" (see the Standard Library documentation).

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