MICROWARE C COMPILER USER'S GUIDE
THE C COMPILER SYSTEM

INTRODUCTION

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

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

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

THE LANGUAGE IMPLEMENTATION

OS-9 C is implemented almost exactly as described in 'The C Programming Language' by Kernighan and Ritchie (hereafter referred to as K & R). A copy of this book, which serves as the language reference manual, is included with each software package.

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

DIFFERENCES FROM THE K & R SPECIFICATION

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

ENHANCEMENTS AND EXTENSIONS

The "Direct" Storage Class

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

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

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

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

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

Imbedded Assembly Language

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

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

Control Character Escape Sequences

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

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

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

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

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

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

IMPLEMENTATION DEPENDENT CHARACTERISTICS

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

Data Representation and Storage Requirements

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

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

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

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

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

Register Variables

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

A considerable saving in code size and speed can be made by judicious use of a register variable. The most efficient use is made of it for a pointer or a counter for a loop. However, if a register variable is used in a complex arithmetic expression, there is no saving. The "U" register is assigned to register variables.

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

Access To Command Line Parameters

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

"C" VARIABLE
NAMES ARE
UNIQUE ONLY UP
TO 8 CHARACTERS

SAME FOR C.PREP
#define ONLY 8CHR

SYSTEM CALLS AND THE STANDARD LIBRARY

Operating System Calls

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

The Standard Library

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

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

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

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

RUN-TIME ARITHMETIC ERROR HANDLING

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

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

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

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

ACHIEVING MAXIMUM PROGRAM PERFORMANCE

Programming Considerations

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

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

The Optimizer Pass

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

The Profiler

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

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

C COMPILER COMPONENT FILES AND FILE USAGE

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

OS-9 Level I Systems:

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

OS-9 Level II Systems:

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

In addition a file called "clib.l" contains the standard library, math functions, and system library. The file "cstart.r" is the setup code for compiled programs. Both of these files must be located in a directory named "LIB" on drive /D1. The DEFS directory must also be on /D1.

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

Temporary Files

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

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

RUNNING THE COMPILER

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

```
        cc [ option-flags ] file {file}
```

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

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

File Name Suffix Conventions

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

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

```
        cc prg.c
```

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

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

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

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

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

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

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

compiles the program with a definition identifier being passed to the compiler. The definition being passed is used within the source to control compilation via IFDEF/IFNDEF functions.

COMPILER OPTION FLAGS

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

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

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

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

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

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

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

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

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

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

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

-D identifier> is equivalent to "#define <identifier>" written in the source file. -D is useful where different versions of a program are maintained in one source file and differentiated by means of the "#ifdef" or "#ifndef" pre-processor directives. If the <identifier> is used as a macro for expansion by the pre-processor, "1"(one) will

CC1 -X
CREATE BUT DO NOT
EXECUTE C.COM | PG B-1

be the expanded "value" unless the form "-d<identifier>=<string>" is used in which case the expansion will be <string>.

COMMAND LINE AND OPTION FLAG EXAMPLES

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