Notes on the IBM C compiler.

Mike Lesk

This note assists in the setup and handling of the IBM C compiler. It describes the makeup of the compiler and library, the runtime environment, and the handling of assembler and operating system dependencies.

Six sections follow.

1. Contents of the system.
2. Code environment.
3. Outline of the library.
4. Operating System dependencies
5. Assembler dependencies.
6. Macro descriptions.

Since the compiler is still under development, beware of out of date copies of this memo. The next language feature scheduled for this compiler is the 'register' storage class for variables.

It is not expected that anyone unfamiliar with C or the IBM 370 systems will understand this memorandum. Nor is it likely to be profitable for anyone not involved in system maintenance to read it.

1. Contents.

There are several distinct programs associated with the C system; they are separated for ease of maintenance and portability. The C system is compatible with either OS or TSS or TSO operating system environments on an IBM 370; output code is assembler code, but can assemble with Assembler H, Assembler G, or the TSS assembler. At Bell Laboratories, the TSS operating system runs at Indian Hill; the OS operating system runs at Indian Hill, Holmdel, and some other locations. The character sets in use at Bell Laboratories may not agree with those in use elsewhere.

The *compiler* proper is the largest single set of programs. It translates the source language, except for the lines beginning with the # character, into assembly code. It exists in only one version; the assembly code contains macro instructions wherever operating system dependencies intrude.

The *preprocessor* is a C to C translator; it removes all lines beginning with # and produces a valid C program with the same meaning. Note that whereas the compiler is a simple one input stream one output stream program which makes minimal demands on the I/O library, the preprocessor (because of the *include* operation) has multiple inputs and expects to be able to open files by name.

The *run-time library* can be broken into two major parts. Routines such as *cgetc*, *cputc* and *printf* do not actually do any I/O, merely buffering up characters into lines. The actual system interactions are preformed by *inout* and *#access* and a few other routines. In general, system dependencies have been isolated in as few areas as possible. The actual transmission of records is outside of the C library; a large assembler program from Raritan River is used to perform OS I/O transfers, and the BLISS library is used on TSS.

The assembler code produced by the compiler is specialized to a particular operating system and assembler by the *macro definitions*. There are three sets of macro definitions: one for OS Assembler H, another for OS Assembler G, and one for TSS Assembler. There is no distinction between TSO and OS code.

Finally, note that the compiler contains two sections which are generated from *code tables* using the program *cvtab*. This is essential to modifying the compiler, although not necessary for reproducing the present version since the generated programs (c14.c and c15.c) are also supplied.

To get off the ground, it is necessary to have either a UNIX system or the load modules for the compiler, since it is self maintaining. The standard magnetic tape IBM C distribution thus contains load modules for the compiler

and preprocessor and a partitioned data set containing the library.

2. Code organization.

The machine's eye view of the object code organization is best understood in terms of the register usage. The registers are divided into several classes:

Registers 0, 1, 13, 14 and 15 are used in calling sequences. The BLISS calling sequence is default. In particular, it is the only way C programs are prepared to be called. The OS FORTRAN linkage can be used to call programs not written in C.

Registers 8, 9, 10, 11 and 12 are used for addressing. In particular, the stack is simulated by addresses off of register 12. Arguments, automatics and the save area are stored on the stack. In any one program, the layout of the stack is in the order: arguments (four bytes per fixed argument, eight byte per float argument), save area (64 bytes), automatics, temporaries, and finally items being passed to further subroutines.

Registers 0, 1, 2, 3, ... are used for expression evaluation (register 0 can not be used for certain pointer operations, however). It is unusual to find an expression requiring even register 4. The class of "register" variables is not yet recognized as distinct from ordinary automatics in the IBM C compiler.

Although the register numbers are parameterized in the object code, they are actually fixed by the specification of BLISS compatibility. For those not familiar with the BLISS call, here is the subroutine interface:

The *calling* program computes the arguments and stores them in addresses 0(13), 4(13), ... where register 13 points to the end of the stack space used by this program. Floating arguments are passed as double and therefore require eight bytes; all other kinds of arguments, including pointers, take four bytes each. The total number of bytes of arguments is placed in register 1. The address of the *called* program is placed in register 15 and (a) on OS a BALR 14,15 is done, (b) on TSS the address of the PSECT of the called program is placed in register 0 and a BASR 14,15 is done.

On entry the *called* program stores the registers on the stack, sets register 12 to address its stack space (merely copying register 13 to register 12), and initializes register 13 to point to the end of the stack space it uses, therefore the beginning of the arguments for any routine it calls. On exit the registers are restored and the value of the function (if any) is returned in register 0 (in floating register 0 if the result is float or double)

The allocation of base registers is as follows: (code) registers 10 and 8; (data) registers 11 and 9. These base registers are established by the prolog macro. A few easy-to -determine facts from a C core image are that (register 10)-16 contains the name of the currently executing routine and (register 15)-16 contains the name of the most recently called routine.

The C language considers underscore (_) to be an alphabetic character. Except on UNIX, this is not a legal character for the loader. On the 370, therefore, underscore is translated to # in external ames. When a program is named in this memo as (e.g.) *#isddn* it must be called from C as

```
_isddn()
```

and the name with the # character must never be used in a C program.

Since the compiler does not bother to 8-align double precision numbers the code it generates will not execute on a 360, as opposed to a 370, processor.

3. Library

The library organization is motivated by the desire to separate the functions performed as much as possible in the interests of portability, maintainability, and permitting users to use some functions without others. There are some routines, such as printf and scanf, which are more or less the same on all systems. They use putchar and getchar and do not care how the stream I/O is actually performed. Next most portable are the cgetc and cputc routines. These routines must know what buffering strategy to employ, so they differ between UNIX and the record-oriented systems. On all the record-oriented operating systems, however, they are similar and use a common strategy. This is to translate "newline" to "record gap" on output and vice versa on input. A buffer is provided for up to 512 character lines. Each newline causes a record to be transmitted.

Underneath these routines, however, is a routine named (on the 370 library) *inout.* It performs the record I/O function by calling some other facility, considered outside the scope of the C library. On the OS system, this is the

RR INOT program; on the TSS system, this is the BLISS library.

A similar strategy is employed with the storage allocator. The routines*calloc* and *cfr ee* are identical on OS and TSS; the differences are represented by the two different routines for performing *getmain* supervisor calls.

A few other low level routines are needed in the library for I/O. At present the file accessing mechanism is complete only at IH TSS. The extra low level routines are:

#isddn to perform a FINDJFCB supervisor call and determine whether its argument is a ddname;

#isdsn to perform a FINDS supervisor call and determine whether its argument is a dsname; and

#ddef to perform the DDEF supervisor call required to create new data sets; and

#rel to perform the REL supervisor call to release previously attached data sets.

The most complex problem faced by the I/O library is the need to provide wrapup (buffer flushing, data set closing) on exit when the standard I/O has been used, without loading the I/O library and causing wrapup when there has been no use of standard I/O. Also, the standard default file opening of files 0, 1 and 2 should be provided for anyone using the standard library; but not forced on people using their own code. This is handled (a) on OS by using weak external references and (b) on TSS by using a program *cgate* and two external cells, *#gate* and *#gates*.

(OS) On entry to any program in the I/O library, the routine *cgate* is called to interrogate the one_time gate cell *#gate* and on the first call to any I/O routine, the standard files are opened. In addition, the gate routine contains a normal external reference to *cexit.* The standard main program invocation contains a weak external reference to *cexit* and a conditional invocation of *cexit* if the weak reference has been supplied. Thus, (a) if the user calls anything in the I/O library, when he executes it the gate cell is set, the standard files are opened, and since *cexit* is referenced it is called for wrapup; while (b) if the user doesn't call anything in the I/O library, none of it except the gate cell is loaded, and on exit there is no wrapup call since *cexit* was not loaded.

In addition, to permit an exit from the middle of a program module without performing a stack trace to find the save area from the operating system, the stack base address (STACK$) is made an entry point so that it is accessible to the exit routine, which can determine the original save area and simulate a normal return to the supervisor.

(TSS)

On entry to any program in the I/O library, the routine *cgate* is called to interrogate the one-time gate cell *#gate* and on the first call to any I/O routine, the standard files are opened. In addition, the gate cell is stuffed with the address of the *cexit* routine to perform wrapup. On exit, the *#gate* cell is interrogated; if non zero it is taken as a function pointer and branched to. Thus, (a) if the user calls anything in the I/O library, when he executes it the gate cell is set and on exit wrapup takes place; while (b) if the user doesn't call anything in the I/O library, none of it except the gate cell is loaded, and on exit the gate cell is still zero and no wrapup is performed.

In addition, to permit an exit from the middle of a program module without performing a stack trace to find the save area from the operating system, the cell *#gates* is stuffed on entry with the base of the stack; from this the exit routine can determine the original save area and simulate a normal return to the supervisor.

Aside from the above discussion, there is no use of weak external references in OS, and no use of the stack base cell in TSS.

The remaining tricky routine is *getargs* which is called before the main program to fetch the 'parm string' or command line to set up the arguments for the main program. This program exists in two different formats, one for OS and one for TSS. On OS the equivalent of the UNIX command line is in the PARM='...' string of the EXEC card for the C job step; on TSS an appropriate invocation of a C program is a line of the form

module 'command line'

and again, the quoted string is the equivalent of the UNIX command line. Within this string both > and < are recognized as on UNIX for default I/O diversion; as explained above, if the user does not call any C library I/O routines they are ignored. The only immediate effect on recognizing these command line diversions is to change the default name pointers that will be used to open files 0, 1 and 2 if and when *cgate* is finally called.

4. Operating System dependencies

Regrettably, there are many differences between OS and TSS operating conventions, and not all of these have been papered over by the C implementation. There is always a choice between making everything UNIX compatible, at the expense of making usage quite unusual on the IBM system; and being compatible with the local system, annoying those bringing programs over from UNIX. In general an effort has been made to keep the semantics of the language constant, while allowing the methods of compiling and loading to agree with local practice.

C was designed on a computer which supports the ASCII character set. Source programs must, therefore, be typed on a terminal which contains all the ASCII characters, including particularly the square and curly brackets, and the backslash. Although all of these characters are defined in the EBCDIC character set, on TSS in particular there is some disagreement about what they are. Since the keywords are only recognized in lower case, TSS users must enter programs in KA mode; three characters differ in KA and KB mode, and to avoid a horrendous problem of associating every program with the mode in which its input must be entered, the input character fetch routine *cgetc* translates the three strange characters (single quote, backslash, and vertical bar) to standard representations. This means that C can not copy an arbitrary file without changing it. Note also that the Holmdel and Indian Hill character sets are different; and that the common C expression

```
c >= 'a' && c<= 'z'
```

does *not* test whether c is a lower case character in the EBCDIC character set. Table 1 shows the character set differences between Holmdel and Indian Hill IBM systems.

Table 1

Character Set Variation

	Character	KA mode	KB mode	Holmdel	Standard
\	backslash	E0	5F	E0	E0
'	single quote	AE	7D	7D	7D*
[	open bracket	8C	8C	AD	AD
]	close bracket	AC	AC	BD	BD
{	open brace	C0	C0	8B	C0
}	close brace	D0	D0	9B	D0
~	tilde	A1	A1	5F	A1
^	circumflex	BD	BD	9A	9A
|	vertical bar	6A	4F	4F	4F

* The standard expects the Ascii ' (047) to correspond to hex 90, but it is clear that programming languages on the 370 will continue to use 7D as the single quote/apostrophe character.

Notes: KA and KB modes refer to the two keyboard input modes on IH TSS. "Standard" is the proposed new Bell Laboratories IBM character set, to be adopted in March 1976. The table entries are the hexadecimal representation of the characters.

In entering the program, care should also be taken either to stick to short lines or avoid formats in which record lengths are limited to 80 characters. In particular, when moving programs from UNIX, be aware that many UNIX C programs contain lines of 120 characters or so in length.

If the character set in the compiler is not what you want, it can be changed by altering the initialization constants in the arrays *atoe* (ascii to ebcdic) and *etoa* (ebcdic to ascii) in the file *c04.c* of the compiler. Note, if you start changing these files, that each must be a 256-character array, that the mapping between character sets MUST be one to one, and that the two tables must be exact inverses. This is so important that you should check any changes you make with a program before installing a new version. When these tables are changed, the entire compiler should be recompiled; the new compiler will then use the new character set. It does not matter what you do with the characters numbered above 127 in the ASCII set so long as the tables are one to one and invert correctly.

The command procedures to invoke the compiler on the now-written program are still in a state of flux.

(TSO)

There is a command procedure (not yet public) which assumes the source data set name conforms to NAME.data, takes an argument of NAME, and delivers a data set named NAME.obj. There need be no relation between NAME and any entry point of the program being compiled. However, the normal mode of storing library programs is as incompletely linked modules stored in a partitioned data set; for the loader and link editor to search these correctly it is necessary for the module name to agree with an entry point name. If there are several entry points, the remaining entries should be alias names of the member. In the library, this occurs with the routines COPEN (alias #FBUFFP) and #DEFINP (aliases #DEFOUT and #DEFERR).

(OS, not TSO)

Since the "include" operation can not be performed under OS/370 the compiler is not much use here.

(TSS)

Normally C programs are stored in data sets of the style source.NAME. The 'cc' command (not yet public) when invoked as

cc name

compiles such a program and places the result in the top job library of your job library stack. The 'name' argument is used as the module name. Because of restrictions imposed by the IH loader, this module name must *not* be the same as the entry name of the program. In fact, *all* module names and entry point names on a library must be distinct.

Since main programs are also placed on libraries at IH, rather than being kept as object modules as on OS, the TSS macros have been modified to suppress the external definition of MAIN as an entry and avoid any common entries for main programs. This is in contrast to the OS situation, where every main program contains an entry point STACK$ pointing to the stack base.

Having managed to compile the C program, it must now be loaded. The C library is not yet publicly available at Holmdel. At Indian Hill it is stored on 'HANIA.C.LIB' which may be shared by anyone. The BLISS parts library (SPOT.ZPNPARTS(0) on Red TSS and BANG.ZPNPARTS(0) on Green TSS) must also be on your joblib stack.

Much of this command language may be avoided on IH TSS by using the *ccrun* procedure. The command

ccrun name,'arguments'

will run program *name* with a command line of *name* concatenated with *arguments.*

When running, note one enormous difference in the I/O systems. On TSS, data sets may be accessed dynamically whereas on OS they may not be. Hence the "file name" argument to COPEN on OS must be a *ddname.* On TSS the name may be either a DDNAME or a DSNAME. The I/O library first checks for a valid DDNAME; if the given name is not a DDNAME a DSNAME is tried; if neither is found, and the file is opened for writing, a new data set is created with the indicated name.

5. Assembler dependencies.

The major difference between the assemblers is caused by the existence of the PSECT/CSECT pair system on TSS, and the presence of the LOCTR (location counter) pseudo-operation on OS Assembler H. On TSS all compiler output must be sorted into read-only and read-write areas, and assigned appropriately. For convenience, the code is also sorted on OS, but the distinction is not important. With Assembler H there is only one CSECT per program file, with a blank name; the various location counters needed are handled with the LOCTR pseudo-op. Assembler G requires multiple CSECTS; the name is taken from the file name (as in the case of the module name on TSS) and the file names for assemblies with Assembler G must therefore be unique. The UNIX compiler also separates code and data, but in a different way.

Compiler output is sorted into seven categories, each imagined as a different 'location counter' although not implemented that way. A set of seven macro instructions define imaginary location counters, as shown below. Only the code and data areas need be addressed; the initialized string and main areas are only addressed by explicit address constant reference.

code Executable instructions are assembled under the macro *codeloc.* These are read-only and placed in a PUBLIC CSECT on TSS.

data Static data in a program file are placed under the macro *dataloc* and placed in a PSECT on TSS.

strings

Strings occurring in ordinary context (e.g. assignment to character pointers) are placed under the macro *strgloc* and mixed into the data segment.

Strings occurring in initialization context can not be mixed with data (consider the problem of handling both the pointers and the characters in the initialization for

char *array[] {"Washington", "Adams", "Jefferson", ...};

and so the *istrloc* provides another location counter for initialized strings. This requires a third control section in TSS assembler or assembler G.

literals

The literals are placed in the *litloc* macro area. This is basically read-only, but may contain address constants; hence literals are placed with code on OS and with data on TSS.

externals

The transfer vector for externals is similarly read-only but contains address constants; it also goes with code on OS and data on TSS. The name of the macro is *tvecloc.*

main The main program must have a separate control section because the stack, which it contains, is too large to have anything after it which is addressable. Hence a macro *mainloc* provides a final control section. At most, then, in assembler G or TSS assembler, there may be four control sections.

C does not adjust the number of base registers used for addressing to the program size. There are two base registers assigned to code and two to data. The transfer vector, which must be addressable, is placed at the end of the data. Hence (1) no function compiling into more than 8192 bytes of code can be expected to run and (2) no file containing more than 8192 bytes of static data and any code can be expected to run. Note that (1) a file may contain many programs so long as no individual function compiles into more than 8192 bytes and (2) a file containing no functions, only declarations may allocate any amount of storage.

Alternatively, programs written to obtain their working storage arrays with the *calloc* dynamic allocator will not have addressing problems. It is in fact planned to replace all large arrays with pointers to dynamically obtained arrays in the compiler; this will eliminate the addressing problem for data. The code addressing problem is not as serious, since the source and object code are roughly the same size. A function 8000 bytes long is very unusual.

The most serious difference between the OS and TSS environments is in the initialization of storage for programs executed several times in sequence. On OS, all initialized storage is re-initialized when the program is re-executed. But on TSS, the data areas from the previous program load are retained when the same program is started again. This is at variance with UNIX and GCOS practice as well. Users not planning to use this TSS 'feature' would do well to type *unload name* before entering *ccrun name* for security. By contrast, storage obtained dynamically on UNIX, GCOS and OS is initially random; on TSS it is zero.

6. Macro definitions.

The macros used by the compiler, and their meanings, are listed here.

bcall

Executes a subroutine call using the BLISS linkage. The arguments are assumed set up. Since all pointers are one word, while 2 words of information are required to call a function on TSS, a TSS function pointer actually points to a two word vector containing the CSECT and PSECT addresses of the actual function. On OS a function pointer is simply the entry point address.

fcall

Executes a subroutine call using the FORTRAN linkage. The arguments are assumed set up.

mainloc

Uses the location counter for the main program.

codeloc

Uses the location counter for code.

dataloc

Uses the location counter for data.

strgloc

Uses the location counter for strings that may be mixed with data.

istrloc

Uses the location counter for strings being initialized externally that may not be mixed with data. Consider declarations of the form char *listp[] {"first", "second", "third"};

litloc

Uses the location counter for literals. Mixed with either code or data.

tvecloc

Uses the location counter for the transfer vector. Mixed with either code or data.

intaddr

Define an address pointer to an internal data cell.

extaddr

Define an address pointer to an external data cell.

intfunc

Define a function pointer for a function defined in this file. On OS this macro defines a single word pointing to the function entry; on TSS three words are defined; the first points to the remaining two words, which are the function CSECT and PSECT addresses.

extfunc

Define a function pointer for a function defined outside of this file.

startup

Define the appropriate location counters for this program file. In the case of Assembler H, these are internal names; for Assembler G and TSS Assembler, these are external names and are generated from the macro argument, which is usually the name of the file being compiled. These names must be unique within a single collection of programs (presumably this is implied by their tie to the file names).

subsave

Accept a call from the supervisor program.

subretrn

Return for a call received by subsave. Define the automatic variable stack.

prolog

Entry code for a C routine: defines base registers, stack pointers. Suitable only for calls from the BLISS calling sequence. Immediately before the code for the program are 16 bytes of information: the name of the function (EBCDIC, 12 bytes) and the length of the list in bytes (binary, one word).

epilog

Exit code for a C routine: returns to caller. Only suitable to return from a BLISS style call.

runmain

Get the command line arguments, call the main program.

stackdo

Define the stack. The argument is the stack length in words, 1000 by default. There is no provision in the C runtime package for growing the stack.

lc

Load a character into a register.

movif

Convert integer to floating; register to register conversion, floating register given first.

movfi

Convert floating to integer; register to register conversion, floating register given first.

7. Acknowledgments

The IBM C compiler was originally written by T. G. Peterson, based on the UNIX compiler by D. M. Ritchie. It has been revised by H. Gajewska and S. Johnson. The IH command procedures and support are by Joe Hall.