Volume 3, Number 3
IBM Personal Computer C Compiler

September 1985

IBM Personal Computer Seminar Proceedings

**The Publication for Independent Developers
of Products
for IBM Personal Computers**

**Published by International Business Machines Corporation
Entry Systems Division**

Changes are made periodically to the information herein; any such changes will be reported in subsequent Proceedings.

It is possible that this material may contain reference to, or information about IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such products, programming or services in your country.

IBM believes the statements contained herein are accurate as of the date of publication of this document. However, IBM makes no warranty of any kind with respect to the accuracy or adequacy of the contents hereof.

This publication could contain technical inaccuracies or typographical errors. Also, illustrations contained herein may show prototype equipment. Your system configuration may differ slightly. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever.

All specifications are subject to change without notice.

Contents

Introduction and Welcome

These are the Proceedings of the IBM Personal Computer Seminar, designed for independent developers of products for IBM Personal Computers. The purpose of these Proceedings is to aid you in your development efforts by providing relevant information about new product announcements and enhancements to existing products. This issue is prepared in conjunction with this seminar. The Proceedings of future seminars for the IBM Personal Computers also will be published and will cover topics presented at those seminars.

Throughout this Proceedings (3.3), the terms Personal Computer and family of IBM Personal Computers address the IBM Personal Computer, the IBM Personal Computer XT, the IBM Portable Personal Computer, and the IBM Personal Computer AT. The IBM Personal Computer C Compiler is not supported on the IBM PC*jr.*

Purpose

What is our purpose in issuing a publication such as this? It is quite simple.

The IBM Personal Computer family is a resounding success. We've had a lot of help in achieving this success, and much of it came from the independent developers.

As you proceed with your development, do you at times wish for some bit of information or direction which would make the job easier? Information which IBM can provide? This is the type of information we want to make available to you.

Since we want to be assured of giving you the information you need, we ask you to complete the questionnaire which appears at the end of these Proceedings. Your response to this questionnaire will be taken into account in preparing the content of future issues, as well as the content of seminars we will present at microcomputer industry trade shows.

Topics

The following list gives a general indication of the topics we plan to cover in future seminars and include in the IBM Personal Computer Seminar Proceedings:

- Information exchange forum — letters to the editor format
- Development tools — languages, database offerings
- Compatibility issues
- New devices — capacities and speeds
- System capacities — disk and memory
- Enhancements in maintenance releases
- Tips and techniques
- New system software
- Hardware design parameters
- Tips on organizing and writing documents for clear and easy reading
- Changes to terms and conditions

IBM Personal Computer C Compiler

Overview

The C language is a powerful, general-purpose programming language that is capable of generating efficient, compact and portable code. The IBM Personal Computer C Compiler (referred to as CC) is a high function C language compiler designed for application programmers.

The C Compiler runs under the IBM Personal Computer Disk Operating System (DOS, Versions 2.10 or later) and can coexist in an IBM Personal Computer Network environment. In addition, the C Compiler offers:

- High Performance
- Code Optimization Levels
- Comprehensive Symbolic Debugger
- Interrupt Support for System Calls (DOS and BIOS)
- Selectable Memory Models (Small, Medium, Large and Huge)
- Floating-Point Emulation
- 8087/80287 Support
- Overlay Support (at link time)
- Library Manager
- Program Maintenance (MAKE)
- Macro Assembler Support
- 80286 Compile Option

The C language does not provide such standard features as input and output capabilities and string manipulation. The IBM Personal Computer C Compiler provides these features as part of the run-time library of functions. Because the functions that require interaction with the operating system are logically separate from the language itself, the C language is especially suited for producing portable code.

Command Line Processor

This section describes the command line processor and the available options for the IBM Personal Computer C Compiler. The processor is called 'CC'.

The general capabilities provided by CC are as follows:

- Compiles one source file at a time
- Can use several methods for input of filenames and options:
  - Prompts
  - Full command line (may be redirected)
- Access a wide range of options covering such items as:
  - Generation of assembly and combined source/object code listings
  - Methods for handling floating point operations
  - Type of optimization performed
  - Preprocessor control, including definition of macro names
  - Control of search paths for include files
  - Selection of memory model and target processor
  - Emission of debugging information

Operation of CC

Unlike Pascal and FORTRAN, which require the user to invoke each of the compiler passes separately, CC handles everything. It takes the information supplied to it in the form of source, object, source listing, and object listing filenames and options, and invokes each of the compiler passes, passing to them the information needed to correctly execute the passes in light of the options selected. However, the interface seen by the user is much the same as that used by other IBM languages and utilities such as Pascal, FORTRAN, the linker and the library manager.

CC can gather the information it needs using two basic methods:

- Prompts
- Command lines (may be redirected)

Each of these methods is described in more detail in the following sections.

Prompts

This is the default mode of operation for CC. Typing CC will cause the driver to prompt for the information it needs in order to compile the program. There are four basic items of information which CC will prompt the user for:

- Pathname for the source file to be compiled (required)
- Pathname for the generated object file (optional, defaults if omitted)
- Pathname for the generated source listing file (optional)
- Pathname for the generated object listing file (optional)

The driver will prompt for each of these items in the order given above. With the exception of the source file name, each of the prompts will have a default file name which will be used unless the user overrides it by providing a different file or pathname for that prompt. In addition, each of these items will have a default filename extension which will be used unless the user overrides it explicitly. The prompts will look as follows:

- Source filename (.C):
- Object filename (basename.OBJ):
- Source listing (NUL.LST):
- Object listing (NUL.COD):

The source file name is required. If the user does not give a file name in response to this prompt, an error message will be displayed and the program will be terminated. The default filename extension .C is assumed, so no extension is required for the filename given. The user may override this by giving a filename with any extension desired, or by giving a filename followed by '.' if no extension is desired.

The object filename defaults to the basename of the source filename given in response to the previous prompt (the basename is the name minus the extension). The default extension for the object file is .OBJ, and it will be used if the user responds to this prompt with a filename without an extension. The user may override this by giving a filename with any extension desired, or by giving a filename followed by '.' if no extension is desired. The user may also specify a directory name in response to this prompt, in which case the object file is created in the specified directory using the default filename as defined above. In order to distinguish a directory name from a filename, the directory name must be terminated by a '\' (backslash) character.

The source listing filename defaults to NUL.LST, which means that no source file listing is created. If a filename with no extension is given in response to this prompt, the default extension .LST is appended to the given filename. The user may override this by giving a filename with any extension desired, or by giving a filenname followed by '.' if no extension is desired.

The object listing filename defaults to NUL.COD, which means that no object listing is created. If a filename with no extension is given in response to this prompt, the default extension .COD is appended to the given filename. The user may override this by giving a filename with any extension desired, or by giving a filename followed by '.' if no extension is desired.

Other compilation options may be specified, either before or after any of the filenames for any of the prompts. These options are described later in this document. Typing ';' in response to any of the prompts causes the driver to cease prompting for the remaining files and use the default values for those prompts.

Examples:

```
CC
Source filename [.C]:DEMO;
```

This causes the program DEMO.C (see sample program) to be compiled and the default names to be used for the remaining prompts. This means the object file is named DEMO.OBJ and no source or object listing files are created.

```
CC
Source filename [.C]:SIEVE.C
Object filename [SIEVE.OBJ]:SIV
Source listing [NUL.LST]:SIEVE
Object listing [NUL.COD]:SIV.OUT
```

This would compile the program SIEVE.C and generate an object file named SIV.OBJ, a source listing file named SIEVE.LST and an object code listing named SIV.OUT.

Redirected Input

Since the compiler must be run under DOS 2.1 (or higher) and these versions of DOS support redirection of terminal input, this method (redirection) is available rather than response files. The advantage is that CC does not have to open and read another file to get its input, thus adding code, but can instead operate independent of whether the input is actually being read from the terminal or a file.

Command Lines

There is also a command line form which can be used. The syntax for this is as follows:

CC Source-file [, [object-file] [, [source-listing] [, [object-listing]]]] [;]

The source file name is required. This is followed by optional comma-separated fields giving the names for the object file, source listing and object listing file. If ';' is present, it terminates the command line and causes any omitted fields to be given the default values as defined above for the prompting mode. The file names following each comma also are optional and, if omitted, default file names are used for each of the omitted names.

The presence of a comma causes the default file names to be changed to the basename of the source file with the default extension as defined above for each field. The user also may specify a directory name for any of the optional filenames, in which case the file is created in the specified directory using the default filename as defined above. In order to distinguish a directory name from a filename, the directory name must be terminated by a '\' (backslash) character. If the command line is not terminated by ';' and not all of the fields are given, CC will revert to the prompt mode for the omitted file names. Options may be given anywhere a space may occur. This is identical to the Pascal and FORTRAN compilers.

Options

A wide variety of options are available when using the C compiler. These options are fully described in the following sections by their general categories and are again listed in alphabetical order in the last section. Options are indicated by either a leading '/' or '-' character.

Memory Model Options (/A)

The C compiler has the capability to generate code for four standard memory modes and has library support for each of these models.

The memory model is selected via the /A option. There are four of these:

/AS - Select the 'small' memory model (default)

/AM - Select the 'medium' memory model

/AL - Selects the 'large' memory model

/AH - Selects the 'huge' memory model

Programs compiled using the 'small' memory model can have a maximum of 64K of code and 64K of data. Programs compiled using the 'medium' memory model can have up to 1M (megabyte) of code and a maximum of 64K of data. Programs compiled using the 'large' memory model can have up to 1M of code and 1M of data. Limits can be circumvented to some extent by the use of the near, far and huge language extensions. Programs compiled using the 'huge' memory model are allowed to declare arrays larger than 64K. Only one memory model option can be selected in a given compilation.

Code Generation and Optimization Options (/G,/O)

The C compiler can generate code for the 8086/8088, 80186/80188 or the 80286 processor depending on the target selected. Code generated for the 8086/8088 will run on any of the processors. Because the 80186/80188 and 80826 processors use an extended instruction set, code generated using one of these options cannot be run on an 8086/8088 processor. However, code generated for the 80186/80188 will run on an 80286.

The available code generation options are as follows:

/G0 - Generate code for 8086/8088 processor (default)

/G1 - Generate code for 80186/80188 processor

/G2 - Generate code for 80286 processor

/GS - Suppress generation of stack checking code

/GT - [Number] - Set data size threshold to number (default is 256 bytes)

The compiler normally generates a call to a stack checking routine (__chkstk) on entry to each function. This sets up local frame and detects any possible stack overflow. The /GS option tells the compiler to suppress this call and set up the stack frame in line, without checking for possible stack overflow. This results in faster and smaller code but is not recommended for routines which have large stack frames or are heavily recursive. The data size threshold option, /GT, is used only when compiling a program in a 'large' or 'huge' model. All structures and arrays larger than the threshold size are allocated their own data segments. The default size is 256 bytes, but this can be changed to any value desired using the /GT switch. Only one of the /G0, /G1 or /G2 options may be selected in given compilation.

The compiler optimizes the generated code by default. However, it also can be instructed to disable optimizations or to optimize for either maximum speed or for minimal code size.

The available optimization options are as follows:

/OA - Assume no 'aliasing' during optimization

/OD - Disable optimizations

/OS - Optimize for minimal code size (default)

/OT - Optimize for maximum speed of execution

/OX - Perform maximum optimization (equivalent to /OA/OS/GS

The /OA option instructs the optimizer to ignore the possibility of pointer 'aliasing'. Normally, the compiler assumes that for any variable reference there also may be a pointer to this variable, which means that if 'p' is a pointer to an integer and 'i' is an integer, then '*p' is a possible 'alias' for 'i'. It also assumes that you may be referencing the integer directly and indirectly (through the pointer) in the same routine. Thus, any time there is a store into the data location referenced by '*p', you also may be changing the value of 'i', so any knowledge about such values is lost at that point. Using the /OA option instructs the compiler that you do not reference memory locations using two different names, and so it can make assumptions about such values which would not otherwise be true and produce better code as a result. This is particularly nice in the case of multiple references to members of a structure through a pointer to the structure.

The /OD option makes it easier to follow code during the debugging phase of program development. It suppresses the optimizations that perform some code movement and make the code difficult to follow.

The /OS and /OT options tell the compiler how to decide which code sequence to generate in cases where there are two different but functionally identical code sequences available for an operation. The /OS option tells the compiler to choose the sequence which takes the least amount of code space, whereas the /OT option tells the compiler to choose the sequence which will execute fastest.

The /OX option is simply a shorthand notation for 'do maximum possible optimization'. It assumes no aliasing or optimizing for minimum space, and it suppresses stack overflow checking.

The /OA option may be used with the /OS or the /OT options in the same compilation. If the /OD option is used, it overrides any other optimization options given and no optimization is performed. If the /OX option is given, it overrides any other optimization options given.

Floating-Point Options (/FP)

The compiler can generate code for handling floating point operations in either of two ways, and the resulting object code can be linked in one of three ways, depending on the options chosen. The options provide trade-offs between the size of the final program and the speed of operation, and also specify whether or not an 8087/80287 co-processor is used (if one is present).

The compiler can generate either in-line 8087 instructions or calls to run-time routines which perform the given operation. If in-line 8087 code is generated, the program may be linked either with an 8087 emulator or with a special 8087 library. If the emulator is chosen and an 8087/80287 is present, the chip will be used, otherwise the instructions will be emulated. If the special 8087 library is linked in, an 8087/80287 must be present for the program to run.

If the run-time calls are generated, the program may be linked with the emulator or the 8087 library, as above, and the same conditions hold. It also may be linked with an alternate math library, which is smaller and faster than the emulator, but gives slightly less precision in the result.

This is very similar to the options available for handling floating-point operations in Pascal and FORTRAN compilers.

Use of any of these options causes library search records to be emitted into the generated object module for the appropriate floating-point libraries for that option. This can be overridden at link time, and any library compatible with the method chosen can be used. The search record generation can be suppressed via the /ZL option (described in a later section).

The available floating point options and their meanings are as follows:

/FPC	- Generate calls to run-time routines and link with the emulator (default)
/FPC87	- Generate calls to run-time routines and link with the 8087 library
/FPI	- Generate in-line 8087 instructions and link with the emulator
/FPI87	- Generate in-line 8087 instructions and link with the 8087 library
/FPA	- Generate calls to run-time routines and link with the alternate math library

The run-time call interface was chosen as the default because code compiled under this option can be linked with any of the floating point libraries without having to recompile the program.

Preprocessor Control Options (/D,/E,/I,/U)

The C language uses a preprocessor to handle macro substitution, include files, and conditional compilation. Although it is treated as if it were part of the compiler, the preprocessor can be used on any text file which contains preprocessor directives and not just source files. (Preprocessor directives are lines which begin with a #define, #elif, #else, #endif, #if, #ifdef, #ifndef, #include, #line or #undef symbol.) CC has a set of options which directly control the operation of the preprocessor. These options provide the capability to run the preprocessor by itself and generate its output in human-readable form, define macros, and add/delete search paths for include files.

The available preprocessor control options and their meanings are as follows:

/Dname[=string]	- define name as a macro during this compilation
/E	- preprocess source file and write to standard output (stdout), with #line directive the output
/Idirectory	- add directory to the search path for include files
/Uname	- remove definition of the predefined macro name

The /E option causes the source file to be preprocessed and the output written to stdout (normally the console screen, but it may be redirected). The preprocessor strips comments from the source. The preprocessor inserts a #line directive in the source file to aid the compiler in determining the actual line number for error messages. The output of the /E option can be redirected to a file and later resubmitted to the compiler for continued compilation.

The compiler uses the INCLUDE environment variable to determine a list of 'standard directories' in which to look for include files. (The current directory is used if no INCLUDE environment variable exists.) This list can be added to using the /I option.

Macros can be defined on the command line using the /D option. These macros then act as if they were defined at the beginning of the source file. This is extremely useful for 'make' files and conditional compilation, since it allows the source to be compiled in a number of ways without having to edit the source file each time a different definition is needed. There are three ways that the macro can be defined:

/Dname

causes name to be defined and given the value 1. This is equivalent to putting the line

#define name 1

at the beginning of the source file.

/Dname=

causes name to be defined and given a single space (blank) character as its value. This is equivalent to putting the line

#define name

at the beginning of the source file.

/Dname=string

causes name to be defined and given the value specified by string. It is equivalent to putting the line

#define name string

at the beginning of the source file.

CC has several predefined macro names that are automatically passed to the preprocessor to aid in creating code which may be ported to a different environment or which may be compiled in a different memory model. These names are DOS, M__I86 and M__I86xM, where x is replaced by S, M, L or H depending on the memory model used. They are defined using the first type of the /D option shown above (e.g.,/DDOS). The /Uname option causes the predefined macro name to be undefined.

Language and Compiler Control Options (/S,/W,/Z)

These options control the use of language extensions and how the compiler operates. Such things as the level of compiler warnings and the kind of information generated are controlled by these options. The available options and their meanings are as follows:

/S	- performs syntax checking only on the source file; no code is generated and no object file created
/Wnumber	- controls the type and amount of warning messages emitted by the compiler (/W1 is the default)
/ZD	- emits line number and symbol table information for use by a symbolic debugger into the object file
/ZE	- enables the extended keywords, near, far and huge
/ZG	- generates function declarations for functions defined in the source file and writes them to stdout
/ZL	- suppresses generation of library search records in the object file
/ZP	- forces members of a structure to be packed

The C language allows many type conversions and statements which are potentially erroneous or dangerous, but which are legal and may be used intentionally by an expert programmer. The compiler will attempt to warn the user about many of these, without terminating the compilation. The /W option can be used to control the number and kind of warnings which are emitted. The number may range from 0 (suppress all warnings) to 3 (warn about all questionable constructs). The default value is set to /W1.

The C compiler has been extended with three keywords, near, far and huge, to allow the user to take full advantage of the architecture of the 8088 family of processors and to create arrays larger than a segment (64K). Since these keywords are not part of the standard C language, they must be enabled via the /ZE option in order to be used. If this switch is not given, they are treated as normal identifiers in the program and will cause compilation errors if used as if they were extensions.

The C compiler has tried to anticipate developments in the C language stemming from the ANSI standard for C, which is currently under development. One of these is the ability to declare the types of the formal parameters of a function and have the compiler perform type checking on the actual parameters to these functions. This capability is implemented in the C compiler. However, since it is a new feature in C, old programs do not contain such declarations. To make it easier to implement such declarations in new code as well as existing code, the C compiler can be instructed, via the /ZG option, to generate such declarations automatically for each function defined in the source file. The source file is lexically analyzed and the declarations are created and written to stdout. This output can be redirected into a file, which can then be included in the source file, to automatically enable this type checking capability when the source is subsequently compiled. No code is generated and no object file is created when this option is selected.

The compiler normally emits library search records into each object module for the appropriate floating-point libraries (see the section on floating-point options) and the appropriate C library. The linker reads these records and uses them to determine which libraries to search for at link time. This option can be suppressed using the /ZL option. This is useful when creating objects which are intended to reside in a user-created library since the search records are not needed in library routines. (The objects with which the library is linked will contain the search records, which is sufficient to cause the linker to search the specified libraries.) As a result, the object modules (and thus the library) will be smaller.

The compiler normally aligns each member of a structure larger than a byte on a word boundary. Using the /ZP option forces the compiler to 'pack' the structure so that each member, regardless of its size or type, begins on the next byte boundary. This saves space and also may be useful in accessing structures which were defined in assembly language routines; however, this also may result in larger and slower code when accessing the members of the structure.

Output File Control Options (/F)

The compiler is capable of generating various kinds of object listing files. The standard object code listing filename is one of the items which is supplied to the driver via the prompts. It contains a listing of the generated object code, with offsets and machine code listed next to the machine language instructions. The compiler also is capable of generating an assembly listing which is acceptable as input to the IBM Personal Computer Macro Assembler (/FA). Another option mixes the source file and the object code listing file into a combined listing, showing the object code generated for each source line (/FC). The latter is particularly useful when debugging large, complex programs.

The available options are as follows:

/FA(filename) - generates an assembly code listing in the file filename which can be used as input to the Macro Assembler

/FC(filename) - generates a combined source and object code listing in the file filename

If more than one option is given, or if one of the options above is given as well as a response to the object listing prompt, then the following rules hold:

1. If the /FC option is given, a combined listing is always produced.

2. If both an assembly listing and a standard object listing are requested, the standard object listing will be produced.

The filename arguments to these options are optional and, if left off, the filename defaults to the basename of the source file with the extension .ASM (for the /FA option) or .COD (for the /FC option). Filename also may be the name of a directory, in which case the listing file is created in the named directory with the default names as defined above. If a directory name is given for filename, it must be terminated with a '\' (backslash) character.

Help Option (/HELP)

This option allows the user to get a listing of available options (/HELP).

The form of this option is as follows:

/HELP - prints a list of the available options on the console screen

If this option is given, no compilation is performed, but a listing of the available options, followed by a short description of their function, is printed to the screen.

Memory Models

Overview

This section discusses the different C Compiler memory models available varying for the 8086/8088, 80186/80188 and 80286 processors.

The segmented architecture of the 8088 family of microprocessors lends itself to a number of possible methods for handling memory usage in C programs. All these processors have four segment registers: CS, DS, ES and SS. The CS register is assumed to always point to the current code segment; the DS register is assumed to point to the current data segment; the SS register is assumed to point to the current stack segment; and the ES register is the ' extra ' segment register which can be used to point to an arbitrary segment.

Instructions which involve code segment references, such as CALL instructions always use the CS register to determine which segment the code resides in. This means that to call functions resident in the current code segment, only a 16-bit offset value is needed. If the routine resides in another segment, a ' long call ' is needed. The ' long call ' takes both the new value to be put into CS as well as the offset of the function within that segment. This requires more code and takes longer to execute.

Data references may use either the DS or SS segment register values, depending on the addressing mode used. DS is used by most addressing modes; however, addressing modes involving either the SP or BP registers use the SS segment register. Although it is not strictly required, the memory models used by the C compiler assume that both SS and DS contain the same segment value (that is, the stack and the static data all reside in a single 64K segment).

To access data in other segments, the ES register is generally used. The segment value of the data is loaded into ES and the data is accessed by overriding the default data segment for the instruction with the ES value.

The register structure imposes some rather tight constraints on a code generator that tries to generate small, efficient code for programs with widely varying code and data requirements. This structure permits only 64K per segment, and only four segments are addressable at any given time. Also, the cost of addressing other segments is fairly high. As a result of these constraints, four common memory models have been evolved to handle the different requirements which may be imposed by various types of programs.

Memory Models

The 'standard' memory models are commonly referred to as the small, medium and large models. In addition to these models, IBM has implemented an extended version of the large model, called the huge model, to handle arrays larger than 64K.

Small Model

The 'small' model assumes that there is at most 64K (one segment) for code and 64K for data and stack. This allows the compiler to generate very efficient code since the segment register values are fixed at load time and never need to be changed or overridden. All pointer values in small model C programs require just 16 bits, to hold the offset of the code or data value from the base of the code or data segment. However, programs with very large amounts of data or code may not fit within these limits.

Medium Model

The 'medium' model assumes that there may be multiple code segments, each up to 64K in size, but only a single 64K data segment. This requires long calls to be used for all function calls. (Both segment and offset of the called routine must be specified.) Code addresses (function pointers) must be 32 bits to hold both the segment and offset of the function. However, data pointers remain 16 bits and all data references can use only the offset value from the base of the default data segment. This model provides a tradeoff between speed of execution (long calls are slower since the code segment register is often changed) and code size (since there usually are far more data references than function calls).

Large Model

The 'large' model assumes that there are multiple code segments and multiple data segments. This means that all data and code references require both a segment and offset, so all addresses are 32 bits long. This is used primarily for programs which require very large amounts of code and/or data. Because each data reference must load a segment value, code generated for this model will generally be much slower and larger than for the other models (except Huge). However, it gives the user the ability to use all of the available memory.

Huge Model

In order to accommodate programs which need to be able to handle arrays larger than the 64K, the C compiler has added a new model, called the 'huge' model. This model is similar to the 'large' model, but the compiler allows arrays larger than 64K to be declared and operated on. Because of the way in which C treats arrays and pointers, pointer arithmetic, so common in C programs, must recognize that the item being pointed to may be larger than 64K and simple offset arithmetic cannot be used. This causes another increase in the size of the code and a decrease in the efficiency of the code involved in both accessing and pointer arithmetic.

For example, the common C statement

```
p++;
```

(where p is a pointer to int, for example) is no longer a simple increment operation since the increment may cross a segment boundary.

Mixed Model Programming

The IBM C Compiler defines four standard memory models (small, medium, large and huge) to accommodate programs with different memory requirements.

One limitation of the predefined memory model structure is that pointers for code or data immediately change size when you change memory models. To overcome this limitation, the IBM C Compiler lets you override the default addressing convention for a given memory model and access an item with either a near, far or a huge pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small or medium model program. You can access that item in another segment, saving space in your default data segment.

The special keywords near, far and huge can be used to declare near, far and huge data items and pointers. To use these special keywords in a program, you must specify the /ZE option at compile time to enable the keywords. Without the /ZE option, the compiler will treat near, far, and huge as ordinary indentifiers, causing program errors.

The near keyword defines an object with a 16-bit address. The far keyword defines an object with a full 32-bit segmented address. Any data item or function can be accessed with a far pointer. However, the size of a far data item is restricted to 64K bytes maximum (one segment). The address arithmetic required to refer to individual elements of a far item is performed on just 16 bits (the offset portion) of the address because all elements are known to reside in the same segment.

The huge keyword identifies a data object with a full 32-bit segmented address. A huge data item can exceed 64K bytes. Because elements of a huge array occupy more than one segment, full 32-bit address arithmetic is required to refer to individual elements of the object.

In medium, large and huge model programs, near lets you access data with just an offset. In small, medium or large model programs, the huge keyword lets you declare and access an array spanning more than 64K bytes (one segment).

When using the near, far and huge keywords to modify addressing conventions for particular items, you usually can use one of the standard libraries (small, medium or large) with your program. The large model libraries are also used with huge model programs. However, you must take care when calling library routines; for example, you cannot pass far data items to a small model library routine.

Since there is no type-checking between items in separate source files, the near, far and huge keywords should be used with great care.

Library Support For Mixed Model

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the four standard memory models (small, medium, large and huge) through three separate sets of run-time libraries. When you write mixed model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

When using the near, far and huge keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries with your program. However, you must take care when calling library routines; for example, you cannot pass far data items to a small model library routine.

Huge Model Constraints

In the current definition of memory models for languages, 'large' model is defined as having 32-bit code pointers and 32-bit data pointers. Arithmetic involving addresses (array references, pointer manipulation, etc.) only involves the low order 16 bits of an address. Thus, data structures involving address calculations (arrays, unions or structures) must have a size less than 64K. Huge model is an attempt to remove this restriction.

For this section, an 'object' is defined as data structures that can be elements of an array: scalar objects (integers and floating-point numbers) and composite objects (structures and unions).

There are a few constraints necessary in defining huge model. Address arithmetic is more expensive than in large model, so efficiency is of some concern. Compatibility with existing languages is required.

To meet these constraints, huge model has the following restrictions:

1. Structures and unions are not permitted to cross segment boundaries.

2. The address space must simulate a linear address space.

3. Arrays of objects requiring more than a segment must align to 64K boundaries.

What are the consequences of these restrictions? No structure or union can be greater than 64K. If an array is greater than 64K but less than 128K, then it can be offset within a segment to ensure that the objects align to the 64K boundry. If the array requires three or more segments (greater than 128K), then the size of the objects in the array must be a power of two (this is true for scalar objects; composite objects will be padded by the compiler). There are two ways that huge address calculations can be required: (1) discrete arrays or pointers can be declared as 'huge' (keywords or attributes), and (2) all addressing can be declared as 'huge' (huge model).

What arithmetic is required that involves addresses? There are pointer increment/decrement operations (add/subtract one object size to a pointer) and static address calculations (static array indexing). Finally, there are based address calculations (pointer to structure element, indexing into an array on a stack frame). The code sequences for each of these operations are almost the same as long integer addition except the segment calculations involve shifting the segment calculation.

The large model libraries are also used for huge model programs.

Error Messages

Overview

This section discusses error messages for warnings and errors issued by the C compiler, including the command line processor (referred to hereafter as 'CC'), and by programs which were compiled by the C compiler.

Several types of error messages may be encountered when running the C compiler or when executing programs compiled by the C compiler. They are:

- Errors issued by the command line processor
- Errors issued by the compiler
- Errors issued at run-time, as the program is being executed.

Command Line Error Messages

The C compiler will issue both error and warning messages. When possible, it will issue a warning and continue, while in other cases it will issue an error message and terminate the compilation. For example, if conflicting listing options are specified, CC will choose one, warn the user that conflicting options were given and state which one will be used. However, some conflicts such as giving too many /D options, may cause CC to terminate the compilation.

Compilation Error Messages

There are three types of messages which can be issued at compile time:

- Fatal error messages
- Warning messages
- Nonfatal error messages

Fatal Error Messages

Fatal error messages are issued when the compiler finds it impossible to continue the compilation; for example, unexpected end-of-file in the source file. In these cases the compiler issues a message indicating a fatal error and describing the error; the compilation is then terminated.

Warning Messages

The compiler issues warning messages when it encounters questionable or potentially erroneous, but syntactically legal, constructs in the program source. These messages are not fatal, and do not prevent the program from being fully compiled and an object module created. It is left up to the individual user to determine if the construct was written as intended or is indeed, in error. The user also may control the number and type of warning messages emitted via the /W command line option.

Nonfatal Error Messages

This section lists and describes the values to which the **errno** variable can be set when a error occurs in a call to a library routine. Note that only some routines set **errno** upon error explicitly mention the **errno** variable. If no mention of **errno** occurs, the routine does not set **errno**.

An error message is associated with each **errno** value. This message, along with a user-supplied message, can be printed by using the **perror** function.

The value of **errno** reflects the error value for the last call that set **errno** The **errno** value is not automatically cleared by later successful calls. Thus, you should test for errors and print error messages, if desired, immediately after a call to obtain accurate results.

The include file errno.h contains the definitions of the **errno** values.

Errno Values

The following list gives only the errno values used under DOS, the system error message corresponding to each value and a brief description of the circumstances that cause the error. The list includes the errors produced by math routines. These errors correspond to the exception types defined in math.h and returned by the **matherr** function when a math error occurs.

Value	Message	Description
E2BIG	Arg list too long	The argument list exceeds 128K bytes or the space required for the environment information exceeds 32K bytes.
EACCES	Permission denied	Access denied; the file's permission setting does not allow the specified access. This error can occur in a variety of circumstances; it signifies that an attempt was made to access a file (or, in some cases, a directory) in a way that is incompatible with the file's attributes. For example, the error can occur when an attempt is made to read from a file that is not open or to open an existing read-only file for writing. Under DOS 3.0 and later, **EACCES** may also indicate a locking or sharing violation. The error also can occur in an attempt to rename a file or directory or to remove an existing directory.
EBADF	Bad file number	The specified file handle is not a valid file handle value or does not refer to an open file, or an attempt was made to write to a file or device opened for read access (or vice versa).
EDEADLOCK	Resource deadlock would occur	Locking violation; the file cannot be locked after ten attempts (DOS Version 3.0 and later only).
EDOM	Math argument	The argument to a math function is not in the domain of the function.
EEXIST	File exists	The O__Creat and O__EXCL flags are specified when opening a file, but the named file already exists.
EINVAL	Invalid argument	An invalid value was given for one of the arguments to a function; for example, the value given for the origin when positioning a file pointer is before the beginning of the file.
EMFILE	Too many open files.	No more file handles are available.

Value	Message	Description
ENOENT	No such file or directory	The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a pathname does not specify an existing directory.
ENOEXEC	Exec format error	An attempt is made to execute a file that is not executable or has an invalid executable file format.
ENOMEM	Not enough memory	Not enough memory is available. This message can occur when insufficient memory is available to execute a child process or when the allocation request in a **sbrk** or **getcwd** call cannot be satisfied.
ENOSPC	No space left on device	No more space for writing is available on the device (for example, the disk is full).
ERANGE	Result too large	An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error also can occur in other functions when an argument is larger than expected; for example, when the pathname argument to the **getcwd** function is longer than expected.
EXDEV	Cross-device link	An attempt was made to move a file to a different device (using the **rename** function).

Math Errors

The following errors can be generated by the math routines of the C runtime library. These errors correspond to the exception types defined in math.h. and returned by the **matherr** function when a math error occurs.

Error	Description
DOMAIN	An argument to the function is outside the domain of the function.
OVERFLOW	The result is too large to be represented in the function's return type.
PLOSS	A partial loss of significance occurred.
SING	Argument singularity: an argument to the function has an illegal value (for example, passing the value zero to a function that requires a nonzero value).
TLOSS	A total loss of significance occurred.
UNDERFLOW	The result is too small to be represented.

Run-Time Error Messages

Because of the nature of the C language and its historical background, very few run-time error messages are ever issued. Unlike languages such as Pascal, the C Language philosophy has been that errors, such as passing a NULL pointer to a string handling function need not be caught by the function but are the province of the user to find. Such errors may or may not cause the program to fail because of hardware memory protection, but the functions are written to assume that arguments passed to them are valid. This makes the functions as fast and as small as possible and frees the careful programmer from forcing run-time checks to be executed whenever a library function is invoked.

There are three categories of run-time error messages:

1. Error messages generated by functions in the run-time library
2. Execution errors
3. Floating-point exception errors

Error Messages Generated by Library Routines

The first category involves messages issued by the abort, assert, perror and floating-point math functions (e.g., sin, log, sqrt, etc.). These messages are unique in that they can be generated only if the user explicitly calls one of these library functions in his program.

The abort function is used to force an abnormal program termination due to some fatal error which the programmer has detected in his code. This function is called only when the user explicitly codes it into his program; none of the other library functions use it. It issues a message to standard error (stderr) stating that the program was aborted.

The assert function is similar to the abort function except it first tests the value of an expression, provided by the user, and terminates the program only when the expression is false. This provides the programmer with a way to test whether certain assertions hold at various locations in the program. If an assertion fails, a message is printed to stderr stating that the assertion failed and giving the name of the module containing the assertion and the line within that module containing the failed assertion. The program is then terminated.

The perror function is informational only; it does not cause the program to terminate. It is typically used by the programmer when the program detects; for example, a failed low level I/O function and the user wishes to print a message indicating the failure and the reason for it. The function prints both a message provided by the user as an argument to perror and a short message indicating the actual error which occurred (there is a global variable, errno, in the run-time which is set to a value indicating the actual error). Since many of these error codes do not necessarily indicate a fatal flaw in the program which requires termination, the user may choose in his program whether to continue and attempt to recover from that failure, or terminate the program at that point by some other means.

The floating-point math functions use the UNIX System V method of handling argument errors for these functions. This method involves printing an error message to stderr indicating the type of error and the name of the function in which it occurred. In addition, the errno variable is set so that the user may determine the actual error and take steps to correct it. Errors such as attempting to take the logarithm of zero or the square root of a negative number are handled in this manner. As with the perror messages, it is left to the user to determine whether to attempt to recover (for example by returning the square root of the absolute value of the number instead) or to terminate the program at that time.

All such errors produce messages indicating the actual error which occurred.

Execution Errors

This category covers the messages which are generated when a serious error is detected by the run-time support functions at execution time. Unlike the messages in the previous category, the user has no control over these messages as they are generated automatically. The errors may indicate algorithmic flaws or may simply require the program to be re-linked. There are only three errors in this category:

1. Floating-point not loaded
2. Stack overflow
3. Null pointer assignment

Floating-Point Exceptions

These errors are generated by the 8087 emulator or the 8087/80287 chip if one is present. When such an exception is detected, it is trapped and an error message is printed to stdout indicating the exception. There are only five exceptions which can be generated when using the C compiler and libraries. The error messages which may be seen are:

- Floating point error: Divide by 0
- Floating point error: Integer overflow
- Floating point error: Invalid
- Floating point error: Overflow
- Floating point error: Stack overflow

Other Features and Utilities

MAKE

Macros can be defined on the MAKE command line as follows:

MAKE filename $macro_1$ ...$macro_n$

where $macro_i$ is of the form

name=text or name='text'

Additionally, macros can be defined in the MAKE file itself with the above format before the macro is referenced or the text associated with each environment variable is available as a macro. The precedence for macro definitions from highest to lowest is:

1. Command line definition
2. MAKE file definitions
3. Environment variable definitions

Macros are referenced in the MAKE file using the following syntax: $(macroname). The text associated with a macro replaces the macro reference and the line is reprocessed for other possible macro substitutions.

Case Dependency

The burden of resolving case dependencies lies with the user. The default link method is to ignore case conflicts in public names. If the user links with multiply-defined public symbols when case is ignored, the linker will issue an error. If the user tries to build a library with object modules that contain multiply-defined public symbols when case is ignored, the librarian will issue an error. The IBM Macro Assembler cannot generate public symbols with lower case names. To interface IBM Macro Assembler with C, the case must be ignored at line time. If the user wants linking to be sensitive to case differences, he must specify the /ignorecase switch at link time.

CLINK

With this linker, a much more advanced overlay system is supported. The major advantage of this overlay scheme is that the overlay structure is specified at link time and requires no change to the user's source. Overlays are performed implicitly by modifying the user's program at link time to reflect the specified overlay structure.

Symbolic Debugger

The IBM Symbolic Debug Utility (SYMDEB) is a debugging program that helps you test executable files. You can display and execute program code, set breakpoints that stop the execution of your program, change values in memory and debug programs that use the floating point emulation conventions. SYMDEB lets you refer to data and instructions by name rather than by address. SYMDEB can access program locations through addresses, global symbols or line number references, making it easy to locate and debug specific sections of code. With SYMDEB, you can debug C programs at the source file level as well as at the machine level. You can display the source statements of a program, the disassembled machine code of the program or a combination of source statements and machine code. SYMDEB accepts source line numbers as arguments to commands for displaying and changing data, setting breakpoints and tracing execution.

The general capabilities beyond the standard DOS DEBUG functionality are outlined below.

- Source line display for C, Assembler
- Expanded breakpoint control
- Public symbols can be used instead of addresses
- Line numbers can be used instead of addresses
- Expressions can be used instead of addresses
- Alternate display forms - word, long, floating
- Emulated 8087 instructions recognized
- Procedure traceback and parameter display for C
- Escape to DOS shell and exit back to debugger
- Execute any DOS command in debugger
- Enhance E command to support additional data types
- Support larger symbol files (> 64K total)
- Execute multiple commands at breakpoints

DEMO.C

This demo program prints out the parameters on the command line and the DOS environment settings. It is the same as DEMO.C on the compiler diskettes.

```
/* Sample program which accepts parameters
 * then prints out those parameters and
 * any environment variables set
 */


main(argc, argv, envp)
    int argc;
    char **argv;
    char **envp;


    {
    register char **p;

    /* print out the argument list for this program */

    for (p = argv; argc > 0; argv--,p++)
    {
    printf("%s\n", *p);
    }


    /* print out the current environment settings.  Note that
     * the environment table is terminated by a NULL entry
     */


    for (p = envp; *p; p++)
    {
    printf("%s\n", *p);
    }


    exit(0);
    }
```

Summary

The IBM Personal Computer C Compiler will enable you to develop small, fast C language applications with a maximum amount of function.

IBM Personal Computer Seminar Proceedings

Publication Number	Volume	Topic
	V1.1	Contains identical information as V1.2
(G320-9307)	V1.2	DOS 2.0 and 1.1 Comparison Compatibility Guidelines for Application Development 8087 Math Co-Processor IBM Macro Assembler
(G320-9308)	V1.3	DOS 2.1, 2.0 and 1.1 Comparison Disk Operation System 2.1 IBM PC*jr* Architecture IBM PC*jr* Compatibility Overview Cartridge BASIC IBM Personal Communications Manager-Modem Drivers
(G320-9309)	V2.1	Contains identical information as V2.2
(G320-9310)	V2.2	IBM Software Support Center International Compatibility Requirements IBM Personal Computer Cluster Program
(G320-9311)	V2.3	IBM Personal Computer Cluster Program Sort, Version 1.00 FORTRAN and Pascal Compiler, Version 2.00 PC*jr* Cartridge Tips and Techniques
(G320-9312)	V2.4	IBM Personal Computer AT Architecture ROM BIOS Compatibility DOS 3.0 Software Compatibility
(G320-9313)	V2.5	IBM PC Network Overview IBM PC Network Hardware IBM PC Network BIOS (NETBIOS) Architecture IBM PC Network Program
(G320-9314)	V2.6	TopView
(G320-9315)	V2.7	IBM Personal Computer Resident Debug Tool
(G320-9319)	V2.8	IBM PC Network SMB Protocol
(G320-9316)	V2.9	IBM PC Xenix
(G320-9317)	V2.10	IBM PC Professional Graphics Software IBM PC Graphical Kernel System IBM PC Graphical File System IBM Plotting System Library IBM Professional FORTRAN IBM PC Data Acquisition & Control Adapter & Software IBM General Purpose Interface Bus Adapter & Software
(G320-9318)	V2.11	IBM Enhanced Graphics Adapter
(G320-9320)	V3.1	IBM PC Information Panel (3295 Plasma Display)
(G320-9321)	V3.2	IBM BASIC Compiler 2.00
(G320-9322)	V3.3	IBM Personal Computer C Compiler

Notes

G320-9322

IBM Corporation
Editor, IBM Personal Computer Seminar Proceedings
Internal Zip 4629
Post Office Box 1328
Boca Raton FL 33432