IBM

BASIC
Compiler

by Microsoft

IBM

BASIC
Compiler
by Microsoft

Preface

How to Use this Book

This book explains what you need to know about the
IBM Personal Computer BASIC Compiler. It is designed
for people who are unfamiliar with the compiler as a
programming tool. Therefore, this book provides both
a step-by-step introduction and a detailed technical guide
to the BASIC Compiler and its use. After a few compila-
tions, you can use this book both as a refresher on
procedures and as a technical reference.

You should have a working knowledge of the BASIC
language to use this book. For reference information on
IBM Personal Computer BASIC, consult the *IBM
Personal Computer BASIC* manual.

Organization

This book contains the following sections:

Introducing the BASIC Compiler: Gives you an intro-
duction to compilers in general and a comparison of
interpreters and compilers. It also provides brief
descriptions of the contents of the BASIC Compiler
package.

Using the BASIC Compiler: Tells you what you need in
order to use the BASIC Compiler, and some things you
need to do the first time you ever try to compile a
program. It also gives you an overview of program
development with the compiler.

Sample Session: Takes you step by step through the
compiling, linking, and running of a demonstration
program.

Creating a Source Program: Describes how to create a BASIC source program for later compilation, and how to use the compiler metacommands.

Debugging with the BASIC Interpreter: Describes how to debug the BASIC source file with the BASIC interpreter before compiling it. Note that the last section, "Differences Between the Compiler and Interpreter," describes differences between the language supported by the compiler and that supported by the BASIC interpreter.

Compiling: Describes use of the BASIC Compiler in detail, including a description of the command line syntax and the various compiler parameters.

Linking: Describes how to use the linker to link your compiled programs to the needed library. (Note that Appendix B of this book contains more detailed reference material on the linker.)

Running a Program: Describes how to run your final executable program.

Using a Batch File: Tells how you can use the DOS Batch command facility to automatically perform the compile, link, and/or run steps. Sample Batch files are included.

Differences Between the Compiler and Interpreter: Describes all of the language, operational, and other differences between the language supported by the BASIC Compiler and that supported by the BASIC interpreter.

> **Note**: It is important that you study these differences and make the necessary changes in your BASIC program before you use the compiler.

The appendixes contain the error messages and more detailed technical information.

Syntax Diagrams

The syntax for statements, commands, and functions in this book is presented in the same format as in the *IBM Personal Computer BASIC* manual:

- Words in capital letters are keywords and must be entered as shown. They may be entered in any combination of uppercase and lowercase letters. BASIC always converts words to uppercase (unless they are part of a quoted string, remark, or DATA statement).

- You are to supply any items in lowercase italic letters.

- Items in square brackets ([ ]) are optional.

- An ellipsis (. . .) indicates an item may be repeated as many times as you wish.

- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Related Manuals

To use the BASIC Compiler and this book, you will also need the following manuals for reference:

- the *IBM Personal Computer BASIC* manual

- the *IBM Personal Computer Disk Operating System* manual

CONTENTS

Introducing the BASIC Compiler

What is a Compiler?

A computer can perform only its own machine instructions; it does not perform BASIC statements directly. Therefore, before a program can be run, some type of translation must occur from the statements in your BASIC program to the machine language of your computer. Compilers and interpreters are two types of programs that perform this translation.

Interpreters

An interpreter translates your BASIC program line by line as your program is running. In order for a BASIC statement to be carried out, the interpreter must analyze the statement, check for errors, then perform the BASIC function requested.

If a statement gets performed more than once (inside a FOR. . .NEXT loop, for example), this translation process must be repeated each time the statement is performed.

Also, BASIC stores your program as a linked list of numbered lines. This means the computer doesn't know exactly where in memory each line is. When you branch to a particular line (using a GOTO or GOSUB, for example), the interpreter must search through every line in a program, starting with the first, until the particular line number is found.

The interpreter maintains a list of the variables in your program in a similar way. When you use a variable in a BASIC statement, this list must be searched from the beginning until the variable is found.

Compilers

A compiler, on the other hand, translates an entire
BASIC program at one time and creates a new file
called an object file. The object file contains machine
code. All translation takes place *before* you actually
run your program; no translation of your BASIC
program occurs while your program is running. In
addition, memory addresses are associated with variables
and with the targets of GOTOs and GOSUBs, so that
lists of variables or of line numbers do not have to be
searched while your program is running.

Some compilers are known as *optimizing* compilers.
They will do things, such as changing the order of
expressions or eliminating common sub-expressions, to
either improve performance or to decrease the size of
your program.

Optimization and the elimination of the translation
step when your program is running combine to make
your program run faster.

Vocabulary

There are a number of words you may run into when
working with the BASIC Compiler (or any other
compiler). Most of these terms are listed in the
Glossary.

For now, you should understand that a program that
is input to the compiler for translation is called a *source*
file. It must be in ASCII format (a text file). The
compiler translates this source and creates a new file as
output. This new file is called a relocatable *object* file.
These two files have the default extensions .BAS and
.OBJ, respectively.

The BASIC Compiler

The BASIC Compiler is an optimizing compiler designed to complement the BASIC interpreter.

Creating application programs with the IBM Personal Computer BASIC Compiler provides two major benefits:

- Increased speed of execution for most programs

- BASIC source code security

A compiled program is optimized machine code, not source code, and consequently substantially improves execution time while protecting your source program from unauthorized alteration or disclosure.

Another advantage of the BASIC Compiler is that, since the BASIC Compiler has been created to support most of the interpreted BASIC language, the interpreter and the compiler complement each other. Therefore you have a powerful programming environment in which you can quickly run and debug programs using the BASIC interpreter, and then later compile those programs to increase their speed of execution.

The Runtime Module: A feature of the BASIC Compiler is the ability to create programs to use the *runtime module*. The runtime module is a file named BASRUN.EXE, and it contains most of the routines needed to implement the BASIC language. It can be thought of as a library of routines, with the peculiarity that it is an executable file.

The runtime module is loaded when program execution begins; it is not reloaded when the program chains to another program. The BASIC routines which are part of the runtime module do not need to be saved on diskette as part of your final compiled (executable) program. Therefore, if you create an application consisting of several programs which use the runtime module, you can save a significant amount of duplicate code and diskette space. Refer to "The BASRUN.EXE Runtime Module" in the section called "Compiling," later in this book, for more information on the runtime module.

Application programs which require the BASRUN.EXE runtime module *cannot* be distributed without entering into a license agreement with IBM. A copy of the license agreement can be obtained by writing to IBM at:

> P.O. Box 1328-P
> Boca Raton, Florida 33432

Note, however, that it is possible to develop programs with the BASIC Compiler which do not use the BASRUN.EXE runtime module, and therefore don't require the license agreement.

Contents of the BASIC Compiler Package

The BASIC Compiler package contains:

- Two diskettes, named BASIC and LIBRARY

  - The BASIC diskette contains the following files:

    - BASCOM.COM — the BASIC Compiler
    - LINK.EXE — the LINK Linker
    - DEMO.BAS — a demonstration program
    - sample batch files

  - The LIBRARY diskette contains the following files:

    - BASCOM.LIB — the BASIC library
    - BASRUN.EXE — the runtime module
    - BASRUN.LIB — the runtime module library
    - IBMCOM.OBJ — the communications module

- The *BASIC Compiler* book (this book)

Using the BASIC Compiler

What You Need

To successfully compile BASIC programs on your IBM Personal Computer, you need:

- Your BASIC Compiler diskettes, BASIC and LIBRARY

- At least 64K bytes of random access memory

- One diskette drive (although we recommend two drives for easier operation)

- A display

- A printer (optional, but recommended)

- The IBM Personal Computer Disk Operating System (DOS) diskette

- Several blank 5-1/4 inch diskettes

Developing Your Program

You develop a program with the BASIC Compiler using these steps:

1. Create the BASIC source file

2. Debug the program

3. Compile the program

4. Link all modules needed by your program

5. Run your program

All these steps are explained in detail in later sections, but first we will remind you to back up your diskettes, and then we will lead you through a demonstration of using the compiler.

The First Time Through

Warning:
You should back up your BASIC Compiler master diskettes, BASIC and LIBRARY, as soon as possible. We recommend you back up the diskettes before you start the sample session which follows.

To back up the diskettes you must first format blank diskettes. Use the DOS FORMAT command with the S option. This formats the diskette and copies the DOS system files and COMMAND.COM onto your diskette. You can start DOS using this diskette. Next use the DOS COPY command to copy the master files to your backup diskettes. Refer to the *IBM Personal Computer Disk Operating System* manual for more information about formatting and copying.

Store your master diskettes in a safe place and work with the backup copies.

Diskette Setup for a Single Drive System

You will be working with these diskettes:

- BASIC diskette

- LIBRARY diskette

- The *work* diskette — a formatted diskette which
  will hold the following files:

    - Your source file
    - The library you'll be using during the linking
      step (BASCOM.LIB or BASRUN.LIB)
    - Any other modules you may be linking to your
      program

- (optional) Runfile diskette — when compiling large
  programs, you may need to write the run file to a
  separate diskette. This diskette will contain:

    - The run (executable) file created by the linker
    - BASRUN.EXE, if your compiled program uses
      the runtime module

    If you don't use a Runfile diskette, all output files
    from the compiler and linker will normally go to
    the work diskette.

Diskette Setup for a Two-Drive System

You will probably find it convenient to create two new diskettes: one with the linker (LINK.EXE from the BASIC diskette) and the BASIC library (BASCOM.LIB from the LIBRARY diskette); another with the linker and the runtime module library (BASRUN.LIB from the LIBRARY diskette).

The diskettes you will be working with will be:

*   BASIC diskette

*   Library-Linker diskette — the appropriate one of the two diskettes created above, containing:

    — LINK.EXE
    — The library you'll be using during the linking step (BASCOM.LIB or BASRUN.LIB)

*   The *work* diskette — a formatted diskette which will hold the following files:

    — Your source file
    — Any other modules you may be linking to your program

*   (optional) Runfile diskette — when compiling large programs, you may need to write the run file to a separate diskette. This diskette will contain:

    — The run (executable) file created by the linker
    — BASRUN.EXE, if your compiled program uses the runtime module

    If you don't use a Runfile diskette, all output files from the compiler and linker will normally go to the work diskette.

Getting Started

We recommend that you compile the demonstration program *before* compiling any other programs, because this sample session gives you an overview of the compilation process. Also, you should read all the following sections. They contain information that is very important to successful development of a program.

Sample Session

This section uses a demonstration program to illustrate the step by step instructions for using the BASIC Compiler. We include instructions for both single-drive and two-drive systems.

If you enter commands exactly as described in this section, you should have a successful session with the BASIC Compiler. If a problem does arise, check and redo each step carefully.

Remember from the section called "Using the BASIC Compiler" that the five steps in developing a program with the BASIC Compiler are:

1. Creating a source file

2. Debugging

3. Compiling

4. Linking

5. Running the program

Creating and Debugging the DEMO Program

Because we have prepared a debugged demonstration program (DEMO.BAS) on diskette, you do not have to perform the first two steps in the program development process. Therefore, the demonstration begins with compilation. We saved the demonstration program on diskette in ASCII format, since the compiler can only read files that are in ASCII format.

Compiling the DEMO Program

Preliminary Steps: Before you actually start the BASIC Compiler, you must prepare the work diskette. You need a blank diskette to use as the work diskette. Follow these steps to prepare it for use with the compiler. Refer to the *IBM Personal Computer Disk Operating System* manual for detailed information on DOS commands.

1. Start DOS.

2. If your work diskette has not already been formatted, format it now. Use the DOS FORMAT/S command so DOS will be copied to your work diskette.

3. Use the DOS COPY command to copy the DEMO.BAS program from the BASIC diskette to your work diskette.

4. Also, copy the file named BASRUN.EXE from the LIBRARY diskette to your work diskette. The demonstration program will be compiled to use the runtime module, so BASRUN.EXE needs to be on diskette when you run your program.

5. **If you have a single diskette drive**, copy the file named BASRUN.LIB from the LIBRARY diskette to your work diskette.

If you have a two-drive system, create a Library-Linker diskette as described previously under "Diskette Setup for a Two-Drive System." Start with a blank formatted diskette, and copy BASRUN.LIB from the LIBRARY diskette and LINK.EXE from the BASIC diskette to this diskette. We will call this diskette the BASRUN-Linker diskette.

In this demonstration all files created by the compiler and by the linker will be placed on the work diskette.

Using the Compiler: Perform the following steps to compile your program:

1. Make sure DOS is started, and diskette drive A: is empty.

   If you have a two-drive system, change the default drive to B: by entering:

   Then insert your work diskette into drive B:.

2. Insert your copy of the BASIC diskette in drive A:. The BASIC diskette contains the BASIC Compiler program.

3. Enter the following in response to the DOS prompt:

   This loads the BASIC Compiler program into your computer. It will display a heading, then a prompt.

4. **If you have a single-drive system**, after the heading is displayed, remove your BASIC diskette and insert your work diskette in drive A:.

5. The first prompt displayed by the BASIC Compiler is:

The name of the demonstration program is DEMO.BAS, so you should respond to this prompt as follows:

(Remember to press Enter after typing the name of the demonstration program.) You don't have to include the extension **.BAS**, since the compiler will use **.BAS** as the default extension.

6. The next prompt will be:

Just press the Enter key:

This will cause the object file to have the filename **DEMO.OBJ**.

7. The last prompt will look like this:

We want a listing, so respond as shown:

The compiler will add the default **.LST** extension to the filename, so our listing file will be named **DEMO.LST**. The /e is not part of the filename, but a special parameter to the compiler. **/E** and the other compiler parameters are discussed under "Compiler Parameters," later on.

The screen should look like this when completed:

As soon as you have answered the last prompt, the compiler will begin its work. The compiler generates relocatable object code that is stored in the file you specified in response to the **Object filename** prompt.

At the same time, the listing file is written out to your diskette with the name you specified in response to the **Source listing** prompt.

8.   When the compiler has finished, it displays the message:

and control is returned to DOS. (The numbers nnnnn will depend on the amount of memory in your computer.)

If you enter the command **DIR**, you should see the two new files listed in your work diskette directory: DEMO.OBJ and DEMO.LST.

9.   At this point in the demonstration run, you can view or print out the source listing file (DEMO.LST).

One way to print the listing file is to use the TYPE command from DOS. Press Ctrl-PrtSc to send screen output to the printer, then enter **TYPE DEMO.LST**. The listing file will be simultaneously printed on your printer and displayed on your screen. When the file has been printed, press Ctrl-PrtSc again so the printer will stop printing everything that is displayed on the screen.

A second way to print the file is to use the DOS command **COPY DEMO.LST LPT1:**

10. When you are done looking at the listing file, you should delete it to gain additional diskette space. To do this from DOS, enter:

Further information on compiling and the listing file is given in the section called "Compiling." You are now ready for the next step — linking.

Linking the DEMO Program

Linking must be done using the version 1.10 linker program on the BASIC diskette (the file named LINK.EXE). The linker searches the BASRUN.LIB library to resolve any external references in your compiled object file, and creates an executable (.EXE) file on diskette.

To use the linker, follow these steps:

1. DOS should already be active. Remove any diskette from drive A:.

   If you have a single diskette drive, insert the BASIC diskette in drive A:.

   If you have two diskette drives, insert the BASRUN-Linker diskette in drive A:. The DOS default drive should be B:, and your work diskette should still be in drive B:.

2. Enter:

Your computer searches your diskette for LINK,
loads it, and then displays a heading followed by
a prompt.

3. **If you have a single-drive system**, remove the
BASIC diskette and insert your work diskette in
drive A:. Your work diskette contains the object
file produced by the compiler (DEMO.OBJ), along
with the library (BASRUN.LIB) needed by the
linker.

4. The first prompt from the linker is:

Respond to this with the name of the object file
created by the compiler:

You don't need to include the **.OBJ** extension,
because the linker provides it automatically.

5. The next prompt from the linker is:

DEMO.EXE is the name we want to give the
executable object file, so just press Enter:

6. The next prompt the linker gives you is:

Just press Enter:

This gives the linker list file the name **NUL.MAP**,
which tells the linker not to create a list file.

The last linker prompt is:

Again, you should just press Enter. The linker
automatically knows which library to use.

This is what the completed screen should look like:

7.   After you respond to the last prompt, the linker
      goes to work. BASRUN.LIB is automatically
      searched to satisfy any unresolved external
      references before linking ends.

      After the linker is finished, control returns to DOS.
      Examine your directory to confirm that you have
      created an executable DEMO file.

      Enter:

      You should see a file named DEMO.EXE.

Running the DEMO Program

Once you have compiled and linked your program, it is
simple to run it. From DOS, enter the program
filename, without its .EXE extension.

In the case of this demonstration, with your work
diskette still in the drive, enter:

The runtime module, BASRUN.EXE, will be loaded
into memory by your program. The compiled
program should run quite fast compared to running
the same program with the BASIC interpreter. You
may want to compare execution speeds by running
the DEMO.BAS program with the interpreter.

Learning More About Developing a Program

You have successfully compiled and run a simple
BASIC program. You are now ready to learn the more
technical details that you need to know to compile
other BASIC programs. The next sections, "Creating
a Source Program," "Debugging with the BASIC
Interpreter," "Compiling," "Linking," and "Running
a Program" contain more extensive descriptions of
each of the steps you followed in this section. The
last section, "Differences Between the Compiler and
Interpreter," describes all of the language, operational,
and other differences between the BASIC Compiler
and the BASIC interpreter.

Creating a Source Program

You may create a BASIC source file using any general purpose text editor. But perhaps the best way is to use the editing facilities of the BASIC interpreter. Remember that the compiler expects its source file to be in ASCII format, so if you do use the BASIC program editor you should SAVE your program using the A option. Otherwise, you will get a "Binary Source File" error when you try to compile your program. For more information on editing, saving, and loading files with BASIC, you should refer to Chapter 4 of the *IBM Personal Computer BASIC* manual.

The BASIC language of the interpreter has some differences from that of the compiler. The interpreter allows a number of editing and file manipulation commands that are useful mainly when creating a program. Examples are LOAD, SAVE, LIST, and EDIT. These are operational commands not supported by the compiler. Some differences also exist for some of the other statements and functions. You should take these differences into consideration while you are editing. See "Differences Between the Compiler and Interpreter," later in this book, for a full description of these differences.

Note also, that the interpreter cannot accept lines greater than 254 characters in length. In contrast to the interpreter, the BASIC Compiler accepts *physical* lines of up to only 253 characters in length. (A *physical* line for the compiler is one which ends in a carriage return-line feed.) However, you can make the compiler accept much longer *logical* lines of input by ending the physical lines with an underscore character (underscores in quoted strings or remarks do not count). The underscore tells the compiler to ignore the following carriage return, so all it sees in the carriage return-line feed sequence at the end of the line is the line feed character. The line feed is the line continuation character understood by the compiler.

For example, the following two physical lines:

are read by the compiler as a single INPUT statement,
which inputs seven values into the array A.

> **Note:** It is somewhat impractical to use this
> technique when using the program editor in the
> BASIC interpreter, because each line created with
> the BASIC program editor must begin with a
> number.
>
> Also, source programs that use this technique
> cannot be debugged using the interpreter.

Compiler Metacommands

A special feature of the BASIC Compiler that is not
available with the BASIC interpreter are the compiler
metacommands. They are called compiler meta-
commands rather than BASIC commands because they
are not really a part of the BASIC language, but rather
they are commands to the compiler. The meta-
commands for the BASIC Compiler are:

- $INCLUDE

- $LINESIZE

- $LIST

- $OCODE

- $PAGE

- $PAGEIF

- $PAGESIZE

- $SKIP

- $SUBTITLE

- $TITLE

Note the distinctive "$" prefix on the compiler metacommands.

The metacommands are included in your source file as part of a remark, after the keyword REM or the single quote. Because they are imbedded in a remark, the metacommands do not cause a syntax error when you run your program under the BASIC interpreter, even though the interpreter does not support them. All the metacommands are discussed in more detail under "Compiler Metacommands" in the section called "Differences Between the Compiler and Interpreter," later in this book.

All the metacommands except $INCLUDE simply affect the format of the listing file created by the compiler.

The $INCLUDE metacommand, however, is a very useful feature that allows you to combine files for your source file. The $INCLUDE metacommand looks like this:

REM $INCLUDE: *'filespec'*

The compiler *includes* the specified file into the source file at the point where it encounters this metacommand. That is, the contents of *filespec*, known as the *included* file, are read and processed as though the included file were inserted into your source file immediately following the $INCLUDE metacommand. When the compiler finishes processing the included file, it goes back to the original BASIC source file and resumes processing at the line that follows the $INCLUDE metacommand.

This process may be thought of as imbedding *filespec* into your source file at the location of the $INCLUDE metacommand.

The included file, like any file to be read by the compiler, must be in ASCII format.

Included files can be very useful for COMMON declarations existing in more than one program, or for subroutines that you might have in an external library of subroutines.

If you use a text editor other than the BASIC program editor, you can create a file of lines without line numbers. The compiler supports sequences of lines without line numbers if the /N parameter is specified when you start the compiler. This feature can make it very easy to include the same file in many different programs.

Debugging with the BASIC Interpreter

It is usually very helpful to use the BASIC interpreter to debug your BASIC source program, in order to check for syntax and program logic errors. Note, however, that this is an optional step; it is certainly possible to create a program without ever running it with the interpreter.

You may use some commands or functions in your source program that execute differently with the interpreter. In those cases, you probably need to use the compiler for debugging. The compiler meta-commands and VARPTR$ are the only instructions supported by the compiler that are not supported in some form by the release 1.00 BASIC interpreter. Also, the interpreter does not support double precision loop counter variables as does the BASIC Compiler.

Nevertheless, the language supported by the compiler is intended to be as similar to the interpreter BASIC as possible. This means you can make the BASIC interpreter your primary debugging tool, which will save you from doing unnecessary compilations and links. Also, the RUN, CONT, TRON and TROFF commands make the interpreter a very powerful interactive debugging tool. Refer to the *IBM Personal Computer BASIC* manual for more information on these commands.

One consideration about using the interpreter as a debugging tool: the interpreter will stop when an error occurs while running a program. Any subsequent errors are not caught until the first detected error is corrected and the program is run again. The compiler, on the other hand, scans all the lines in your program and reports all the errors it detects. Therefore, you may find it helpful to compile your program first to catch syntax errors, then test the logic with the interpreter after you correct all the syntax errors.

Compiling

After you have created and debugged your BASIC source program, the next step is to compile it. During this step you can check out differences that may exist between interpreter and compiler BASIC. The compiler flags all syntax errors as it reads your source program. If compilation is successful, the compiler creates a relocatable object file.

Preliminary Steps

Before you actually start the BASIC Compiler, you should do the following:

1. Start DOS if it's not already started.

2. Remember to format your work diskette if it has not already been formatted. Refer to the *IBM Personal Computer Disk Operating System* manual for information about formatting. You should use the /S parameter with FORMAT so DOS will be copied on your work diskette.

3. Copy your (debugged) BASIC source program to the work diskette. Remember that this program will be input to the compiler, so it must be in ASCII format. If it is already in ASCII format, you can use COPY to put it on the work diskette; otherwise you must enter the BASIC interpreter, load the program, and put it on the work diskette using SAVE with the A option.

4. You may want to copy the files you will need for the linking and running steps now. Consider the amount of space on your work diskette and refer to the sections "Linking" and "Running a Program," later in this book.

5. **If you have a two-drive system** you can make the process a little easier by using the following steps:

   a. Make B: the DOS default drive.
   b. Place the BASIC diskette in drive A: and your work diskette (which has your source file and room for the output file(s)) in drive B:.

   With B: as the default drive, you don't need to include B: as the device for each of the file specifications.

You are now ready to compile your BASIC program.

Device Names: When you start the BASIC Compiler, you must give file specifications for the files it needs. In all cases the default device for the files is the DOS default diskette drive; you can override this by including the device name as part of the filespec.

The device name for the source file indicates the device the file is read from. Allowable devices are:

   A:, B: – diskette drives
   CON – keyboard (buffered input)
   USER – keyboard (non-buffered input)

A device name with an output file indicates where the file is to be written. Allowable devices for the output files (the object file and the listing file) are:

   A:, B: – diskette drives
   CON – screen (buffered output)
   USER – screen (non-buffered output)
   LPT1:, PRN – printers
   COM1:, AUX – asynchronous communications
                     adapters
   NUL – no output file

CON is a DOS reserved word for the display device. Output written to CON is buffered as a file and thus appears in the display in blocks of 512 characters at a time.

USER is a special reserved word for the display. Output written to USER is not buffered and thus appears on the screen on a character-by-character basis.

It makes little sense to send the object file to the screen or printer, since the object file is in binary code and is unreadable as text. Note that the cassette (CAS1:) is not allowed.

Starting the Compiler

You can start the BASIC Compiler in either of two ways. Which one you choose will depend on your own preference and how you happen to have your system set up.

• You can let the BASIC Compiler prompt you for the information it needs. If you have a single diskette drive, you can change diskettes before you proceed with answering the prompts.

• You can enter all the information the compiler needs on a single command line. This is a fast way to start the compiler if you have two diskette drives.

Letting the BASIC Compiler Prompt You

If you want to let the BASIC Compiler prompt you for the information it needs, you can start it as follows:

If you have a two-drive system, you would normally have B: as the default drive.

With the BASIC diskette in drive A:, enter:

The BASIC Compiler will be loaded into your computer.
After a short time, the compiler will display a heading
and the following prompt:

```
Source filename [.BAS]:
```

Before you respond, you should insert the work diskette
containing your program into a diskette drive. If your
system has a single diskette drive, you must remove the
BASIC diskette and replace it with your work diskette.
If you have a two-drive system, you would normally
put your work diskette in drive B:.

Source filename is the name of the file which will be
the input to the compiler. Here you should enter the
name of the source file, that is, your program file. For
example:

```
Source filename [.BAS]: MYPROG
```

The name shown in the brackets [.BAS] is the default
filename extension that the compiler will use if you
don't include a filename extension of your own. In this
example, the compiler will be using the file
MYPROG.BAS on the DOS default diskette drive.

After you enter the source filename, you will see this
prompt:

```
Object filename [MYPROG.OBJ]:
```

Object filename is the name you want the object file to
have. If you wish to have your object file stored under
the default name (in the brackets — MYPROG.OBJ in
this example), you may simply press the Enter key. If
you want the object file to have a different name,
enter that name after this prompt. The compiler will
add the extension **.OBJ** to the filename if you don't
include a filename extension.

28

The last prompt will look like this:

Source listing is the name of the file that will contain the compiled source listing. The source listing contains errors and other messages produced by the compiler, and is discussed in detail in the section called "Sample Compiler Listing," later on.

If you do not want a listing, press the Enter key. This will give you the default filename **NUL.LST**, which tells the compiler not to create a source listing file, although any messages will still be displayed on the screen. If you do want the listing, enter the name you want to give to the listing file here. The compiler will add the **.LST** extension if you don't include one in your filename.

You may have the listing file printed out by answering this prompt with the device name of the printer. For example,   .

This method is a fast way to get the listing printed, since it does not require the creation of a diskette file.

Optional Compiler Parameters: You may include special compiler parameters by adding them after the file specification in response to any prompt. Each parameter must begin with a slash (/). The optional compiler parameters are discussed in the section called "Compiler Parameters," later on.

29

Using a Single Command Line

The BASIC Compiler can also be started by using the
following command line:

BASCOM *sourcefile, objectfile, listfile* [*parm*] . . .;

sourcefile is the name of your source file.

objectfile is the name you want to give the object file
which is created by the compiler.

listfile is the name you want to give the listing file.

parm is an optional compiler parameter. Each
parameter must begin with a slash (/). These
parameters are explained in the section
called "Compiler Parameters" later on.
Take special note of the **/O** parameter, which
affects the way you link and run your
compiled program.

Both the BASIC Compiler (BASCOM.COM) and your
source file must be accessible on diskette when the
command is executed. After you enter the command
line from DOS, the BASIC Compiler is loaded and
immediately performs the tasks indicated by the
command field.

Variations on the Command Line: If you enter the
complete BASCOM command line, as shown above,
you will not be shown the prompts for the filenames
as described previously.

If you do not have an entry for all three files in the
command line and the command line does not end in a
semicolon (;), the BASIC Compiler will prompt you
for the remaining unspecified files. If necessary, you
may at this point remove the BASIC diskette in order
to change diskettes. As explained previously under
"Letting the BASIC Compiler Prompt You," each
prompt will display its default which may be accepted
by pressing the Enter key, or overridden by entering
your own file specification.

If you do not include the source filename in the
command line, the compiler will ask for one. The
parms will never be prompted for, but may be added
to the end of the command line or to any file
specification given in response to a prompt.

If the list of filenames is incomplete but the command
line does end in a semicolon (;), the unspecified files
will be set to the defaults without further prompting.

Some examples:

BASCOM myprog.any

> The source file is MYPROG.ANY. The compiler
> prompts for the object file, showing a default
> name of MYPROG.OBJ. After a response is
> given, another prompt is displayed showing the
> default of NUL.LST.

BASCOM myprog.any;

> If the semicolon is added, the compiler uses the
> default names for the remaining files:
> MYPROG.OBJ for the object file, and no source
> listing.

BASCOM myprog.any,,;

> The comma overrides the "no listing" default for
> the source listing file. Instead, you get a listing
> file with the source filename and the extension
> .LST. For this example, you get an object file
> named MYPROG.OBJ and a source listing named
> MYPROG.LST.

BASCOM myprog.any,,

> Using the same example, but without the
> semicolon, the source file MYPROG.ANY is
> compiled, the object file produced is named
> MYPROG.OBJ, but you get a prompt for the
> listing file with the default of MYPROG.LST.

COMPILING

BASCOM myprog,,list

> This causes the compiler to use MYPROG.BAS
> as its source file, and to generate a MYPROG.OBJ,
> and a LIST.LST file. Even though the line does
> not end in a semicolon, no prompts are
> produced.

BASCOM myprog.any,nul,prn;

> This compiles the source program MYPROG.ANY,
> but no object file is produced; the source listing
> file is sent to the printer. This technique is
> useful for debugging.

Compiler Parameters

In addition to specifying filenames, extensions, and devices to direct the compiler to produce object and listing files, you can tell the compiler to perform additional or alternate functions by specifying extra parameters.

These compiler parameters may be placed at the end of the command line after the file specifications, or after any file specification given in response to a prompt. Additional parameters may follow other parameters. For example, the following command line compiles a program named FROG.BAS and includes the /D and /X parameters:

Parameters signal special instructions to be used during compilation. The parameter tells the compiler to perform a special function or to alter a normal compiler function. More than one parameter may be used, but all must begin with a slash (/). *Do not confuse these parameters with similar parameters on other IBM Personal Computer software products.*

All the compiler parameters are described on the following pages. First, we'll give you the detailed descriptions of each parameter. Then you'll find a chart that summarizes the function of each parameter.

Error Trapping Parameters

If your BASIC source program contains error trapping routines that involve the ON ERROR statement plus some form of a RESUME statement, you need to use one of the two error trapping parameters, /E or /X. Error trapping routines require line numbers in the object (.OBJ) file. If you do not use one of the error trapping parameters, the compiler does not include line numbers in the object file, and a compiler error results.

The error trapping parameters allow you to use ON ERROR statements in your program. These statements can aid you greatly in debugging your BASIC programs. Note, however, that extra code is generated by the compiler to handle ON ERROR statements.

Parameter   Action

/E   The /E parameter tells the compiler that the program contains an ON ERROR with a RESUME *line* construction. To handle ON ERROR properly, the compiler must generate extra code for the GOSUB and RETURN statements. Also, the compiler must include a line number address table (one entry per line number) in the binary file, so that each runtime error message includes the number of the line in which the error occurs. The /E parameter generates four extra bytes of code for each line number in your program. To save memory space and execution time, do not use this parameter unless your program contains an ON ERROR statement.

            Note: If you use any RESUME statement other than RESUME *line*, or you use a combination of RESUME *line* with other RESUME statements, use the /X parameter instead.

Parameter	*Action*

/X   The /X parameter tells the BASIC Compiler that the program contains one or more RESUME, RESUME NEXT, or RESUME 0 statements.

The /X parameter performs all the functions of the /E parameter, so the two need never be used at the same time. For instance, the /X parameter, like the /E parameter, causes a line number address table (one entry per statement) to be included in your binary object file, so that each runtime error message includes the number of the line in which the error occurs. The /X parameter also performs additional functions not performed by the /E parameter. The /X parameter generates four extra bytes of code per statement. For example, the program line:

consists of two statements, so eight bytes of extra code are required when you use the /X parameter.

Note that to handle RESUME statements properly, the compiler cannot optimize across statements. Therefore, do not use /X unless your program contains RESUME statements other than RESUME *line*.

Event Trapping Parameters

If your BASIC source program uses event trapping
(COM(n), KEY(n), PEN, or STRIG(n)), you need
to use one of the two event trapping parameters, /V
or /W. These parameters tell the compiler to add the
extra code to check for the particular event at each
line or at each statement.

Parameter   Action

/V   The /V parameter tells the BASIC
          Compiler to include an event trap check
          at each statement. This generates one
          extra byte of code per statement. The
          compiler also cannot do as much
          optimization as it can with /W. When
          you specify /V, the compiler can only
          optimize within a single statement, not
          across statements. Event trapping with
          the /V parameter most closely resembles
          event trapping in the interpreter.

/W   The /W parameter tells the compiler to
          add an event trap check at each line
          number. This parameter generates one
          extra byte of code for each line number
          in your program. If you are using event
          trapping, the /W parameter uses less space
          and takes less execution time than the /V
          parameter. If there is only one statement
          per line in the program, then the /W
          parameter is the same as the /V parameter.

Convention Parameters

The /**4** and /**T** parameters are used to affect the scanning and execution conventions of the compiler. These two parameters are a feature of the BASIC Compiler that allow you to compile and run programs that were written in another version of BASIC — in particular, Microsoft's version 4.51.

The convention parameters may be used together (/**4**/**T**). The individual action of each parameter is explained below:

Parameter   Action

/**4**   The /**4** parameter tells the compiler to use the scanning conventions of the Microsoft 4.51 BASIC-80 interpreter. Scanning conventions are the rules that the compiler uses to recognize the BASIC language.

When you specify /**4**, the compiler follows these rules:

- Spaces are not significant

- Variable names with imbedded reserved words are invalid

- Variable names are restricted to two significant characters

You use the /**4** parameter to correctly compile a source program in which spaces do not delimit the reserved words. For example:

Without the /**4** parameter, the compiler would assign the value of the variable ATOBSTEPC to the variable FORI. With /**4**, the compiler recognizes the line as a FOR statement.

Note: The /**4** and /**N** parameters may not be used together.

37

Parameter	*Action*
/T	The /T parameter tells the compiler to use the Microsoft BASIC-80 version 4.51 execution conventions. "Execution conventions" refers to the implementation of BASIC functions and commands and what they actually do when you run your program. In particular:

- FOR. . .NEXT loops are always executed at least one time.

- The value returned by POS and LPOS may range from 0 to *width*-1. (This differs from IBM Personal Computer BASIC in that the first column is 0, not 1.)

- The argument to TAB may range from 0 to 255, and again, 0 is the leftmost column and *width*-1 is the rightmost column. Also, if the current print position is already beyond space *n*, then TAB(*n*) has no effect.

- The argument to SPC may range from 0 to 255. SPC(*n*) prints *n* spaces, even if *n* is greater than the defined width of the device (it does not do modulo arithmetic in this case).

Parameter   Action

/**T**
(continued)

- When BASIC requires an integer, it truncates single- or double-precision values instead of rounding, except in INPUT statements.

- The INPUT statement leaves the variables in the input list unchanged if only Enter is pressed. The "?Redo from start" message is only displayed when an invalid response is given. In this case, a valid input list must be entered or the message "?Redo from start" is displayed again.

Special Code Parameters

The BASIC Compiler can generate code for special uses or situations. Some of these special code parameters cause the BASIC Compiler to generate larger and slower code, so you should only use them when necessary.

Parameter   *Action*

/A   The /A parameter includes a listing of the object code for each source line in the source listing. The object code is in mnemonic (like assembly language) form. Normally, the source listing produced by the compiler consists of a listing of your BASIC source program with any error messages, plus the relative locations of your code and the size of your accumulated data area. If the /A parameter is set, the source listing also includes the object code generated for each statement. Use of this parameter will give you a longer listing file. Use it only if you want the additional information.

/C   The /C compiler parameter is similar to the /C: option you can use when you start Disk or Advanced BASIC from DOS. It sets the size of the buffer for receiving communications data. This parameter is specified as:

              /C:combuffer

              where *combuffer* is the desired size of the buffer. If the /C parameter is omitted, 256 bytes are allocated to the receive buffer. For example, the command line:

/C   will compile the source program named
(continued)   COMTEST.BAS. The compiler will
              generate a special call so that when the
              compiled program is run, the
              communications buffers will be allocated
              to 1024 bytes.

              Note: A minimum of 256 bytes are
              always allocated to the communica-
              tions receive buffer regardless of the
              size specified with the /C parameter.

/D   The /D parameter causes debugging and
              error handling code to be generated by the
              compiler. Use of /D allows you to use
              TRON and TROFF in your program.
              Without /D set, TRON and TROFF are
              ignored and the compiler generates a
              warning message.

              With /D, the BASIC Compiler creates
              somewhat larger and slower object code
              that performs the following checks:

              ● **Arithmetic overflow**. All numeric
                operations are checked for overflow
                and underflow.

              ● **Array bounds**. All array references
                are checked to see if the subscripts
                are within the bounds specified in
                the DIM statement.

              ● **Line numbers**. The generated object
                code includes line numbers so that
                errors that occur when you run your
                program will indicate the line where
                the error occurred.

              ● **RETURN**. Each RETURN statement
                is checked for a prior GOSUB
                statement.

Parameter	*Action*

/D
(continued)

Without the **/D** parameter set, array bound errors, RETURN without GOSUB errors, and arithmetic overflow errors do not cause error messages when you compile your program. No error messages occur when you run your program either, even though the program may run incorrectly. Use the **/D** parameter to make sure that you have thoroughly debugged your program.

/N

The **/N** parameter tells the compiler to relax line numbering constraints. When **/N** is specified, line numbers in your source file may be in any order, or they may be eliminated entirely. Any line numbers which exist have nothing to do with the sequence of the lines; they serve only as labels for GOSUBs, GOTOs, and any other statements which use line numbers as references for branching.

With **/N**, lines are compiled normally, but unnumbered lines cannot be targets for GOTOs or GOSUBs.

There are three advantages to using **/N**:

- Elimination of line numbers increases program readability.

- The BASIC Compiler optimizes over entire blocks of code rather than single lines (for example in FOR...NEXT loops).

- BASIC source code can more easily be included in a program with $INCLUDE.

Parameter   Action

/R   The **/R** parameter tells the compiler to store
              multidimensional arrays in row-major order.
              For a two-dimensional array, this means that
              the second subscript varies faster than the
              first.

              The default is to store multidimensional
              arrays in column-major order.

/S   The **/S** parameter forces the compiler to
              write string constants more than four
              characters long to your .OBJ file on
              diskette as they are encountered, rather
              than keeping them in memory as your
              program is compiled. Without **/S**, you
              may run out of memory space while the
              compiler is running if your program
              contains many long string constants
              (quoted strings).

              The **/S** parameter allows programs with
              many quoted strings to take up less
              memory during compilation. However, it
              may increase the amount of memory
              needed to run your program, since
              identical strings may be coded several
              times. Without **/S**, any references to
              identical strings are combined so that the
              string is only coded once in your final
              compiled program.

/O   The **/O** parameter tells the compiler to
              compile the program so it does *not* use
              the runtime module BASRUN.EXE. The
              next pages discuss the runtime module in
              more detail.

43

The BASRUN.EXE Runtime Module

If you do not specify the /O parameter, your program is compiled to use the BASRUN.EXE runtime module. As explained under "Introducing the BASIC Compiler," BASRUN.EXE can be thought of as a library of routines, with the peculiarity that it is an executable file. It contains the body of routines that, in essence, make up the BASIC language. Your compiled object file, on the other hand, implements the particular algorithm that makes your program a unique BASIC program.

The runtime module contains the more frequently used BASIC routines. If your program uses other less frequently used routines, these routines are searched for and found in BASRUN.LIB when you link your program.

The runtime module is loaded when you run your executable program, and both BASRUN.EXE and your program reside in memory at the same time. Once loaded, BASRUN.EXE takes up 30K-bytes of memory when you run your program.

Remember, you must enter into a license agreement with IBM before you distribute programs which use the runtime module.

Using the runtime module gives you the following advantages:

- Chaining works as in the interpreter. That is, files are left open, device status is preserved, and you can use COMMON to pass variables to the chained-to program.

- It may take less time to link, since unresolved external references do not have to be searched for in multiple library modules.

When you do specify the **/O** parameter to the compiler, the runtime module is not used by your program at all. BASCOM.LIB is the library which is searched when you link the program.

Advantages of using **/O** are:

- For small and simple programs, you may be able to compile and link smaller programs than the 30K-byte minimum required to accommodate the BASRUN.EXE module.

- Execution of a compiled and linked .EXE file does not require the existence of the runtime module on diskette when you run your program.

There are, however, some disadvantages to using **/O**:

- COMMON is not supported between programs. That is, the CHAIN command is semantically equivalent to the RUN command.

- Complete code for required BASIC routines is included in every .EXE file generated, thus increasing the size of each of your .EXE files. This is not the case for .EXE files using the runtime module.

Summary Chart of Compiler Parameters

Category	Parameter	Action
Error Trapping	/E	Program has ON ERROR with RESUME *line*
	/X	Program has ON ERROR with RESUME, RESUME 0, or RESUME NEXT
Event Trapping	/V	Perform event trap test at every statement
	/W	Perform event trap test at every line
Convention	/4	Use Microsoft 4.51 scanning conventions (not allowed together with /N)
	/T	Use Microsoft 4.51 execution conventions
Special Code	/A	Include listing of object code in the listing file
	/C:*combuf*	Set size of communications buffer
	/D	Generate debug code for error checking when program runs
	/N	Relax line numbering constraints
	/R	Store multidimensional arrays in row-major order
	/S	Write quoted strings to .OBJ file on diskette and not to data area in memory
	/O	Do not use BASRUN.EXE runtime module

When the Compiler Finishes

As soon as you enter the command line, or answer the last prompt, the compiler begins its work. If the program contains any syntax errors, these errors are displayed on the screen as well as in the listing file.

When the compiler is finished, it displays a message with the number of errors it has found. The message looks something like this, and appears on the screen as well as in the source listing:

If the compiler shows only warnings, you can generally continue with the next step, linking. You should check "Appendix A. Messages" to see what the actual message meant.

If the compiler has indeed found errors, you must locate and fix those problems in your source program before you go on to the linking step. You must change your program, save it again (in ASCII format), and rerun the BASIC Compiler.

When the compiler is finished processing, control is returned to DOS. You may interrupt the compiler run prior to its normal completion by pressing Ctrl-Break.

COMPILING

Offset	Data	Source Line	IBM Personal Computer BASIC Compiler V1.00
001A	0002	10 ' $TITLE:'Sales Report' $SUBTITLE:'Jones Co.' $PAGESIZE:70	
001A	0002	20 ' PRICES ARE STORED IN ARRAY M	
001A	0002	30 ' NUMBER OF ITEMS SOLD IS STORED IN ARRAY N	
001A	0002	40 ' TOTAL SALES FOR EACH MODEL ARE STORED IN ARRAY T	
001A	0002	50 PRINT TAB(13);"Sales Report":PRINT :PRINT	
0050	0002	60 PRINT "Model #1","Model #2","Model #3":PRINT	
006E	0002	70 DEFINT I,J	
006E	0002	80 DATA 99.99,62,150,8,275.50,12,25,22,61.99,82,70,31,125.75, 20,265,7,315.50,2	
006F	0002	90 DIM T(3,3),M(3,3),N(3,3)	
006F	00C2	100 FOR I=1 TO 3	
0075	00C2	110 FOR J=1 TO 3	
007B	00C2	120 READ M(I,J),N(I,J)	'read data into the arrays
009D	00C6	130 REM $OCODE+	'turn on object code listing
009D	00C6	140 T(I,J)=M(I,J)*N(I,J)	'compute total sales
0026	**	L00130:	
009D	**	L00140: MOV DI,J%	
00A1	**	SAL DI,1	
00A3	**	SAL DI,1	
00A5	**	ADD DI,I%	
00A9	**	SAL DI,1	
00AB	**	SAL DI,1	

Sample Compiler Listing

The format of the listing is affected by several of the BASIC Compiler metacommands. Refer to "Compiler Metacommands" in the "Differences Between the Compiler and Interpreter" section for a complete description of all the metacommands.

Every page of the BASIC Compiler source listing has a header at the top. In the upper left-hand portion of the page, the first two lines contain your choice of title and subtitle, set with the $TITLE and $SUBTITLE metacommands, respectively.

The first three lines in the upper right-hand portion of the page contain the page number, the date, and the time, respectively. The name and version number of the compiler appear on the line below the time, aligned with the right margin. The column labels also appear on that line.

The Offset and Data Columns

The numbers in the columns labeled "Offset" and "Data" are in hexadecimal. "Offset" is the relative address of the code associated with the source line, using 0 as the start of the program. "Data" is the cumulative data area needed so far during the compilation. Asterisks (**) appear in the Data column opposite object code.

Source Line

Your BASIC source program is listed in the "Source Line" column. You can turn the source code listing off or back on with the $LIST- and $LIST+ metacommands (errors are always listed).

If you requested an object code listing, either by specifying the /A parameter to the compiler, or by including the $OCODE+ metacommand in your source program, the object code is also shown in this column, indented underneath the associated source line. You can turn the object code listing on and off with the $OCODE+ and $OCODE- metacommands.

```
OOAD    **                              MOV     BX,DI
OOAF    **                              ADD     DI,OFFSET T!
OOB3    **                              MOV     DX,BX
OOB5    **                              ADD     BX,OFFSET M!
OOB9    **                              MOV     SI,BX
OOBB    **                              ADD     DX,OFFSET N!
OOBF    **                              MOV     CX,DI
OOC1    **                              MOV     DI,DX
OOC3    **                              INT     93H
OOC5    **                              DB      65H
OOC6    **                              MOV     DI,CX
OOC8    **                              INT     93H
OOCA    **                              DB      4BH
OOCB    OOC6    150 REM $OCODE-                         'turn off object code listing
OOCB    OOC6    160 PRNT T(I,J),                        'print total sales
                         ^ SN
OOCB    OOC6    170 NEXT J
OOD9    OOC6    180 PRINT
OOE2    OOC6    190 NEXT I
OOFO    OOC6
OOF3    OOC6


19742 Bytes Available
19015 Bytes Free

    0 Warning Error(s)
    1 Severe  Error(s)
```

Compiler Messages

The compiler gives messages to you in the listing in the form of two-character codes. The compiler shows the error by pointing to the place in the line where the error occurred with a caret ($\wedge$) beneath the source line, along with showing the two-character code for the error. Appendix A of this book lists all the two-character codes along with their explanations.

Some of the two-character codes are only warning messages; warnings do not need to be corrected before you go on to the linking step. If a message is a warning, it is noted in Appendix A. If the explanation does not say that the message is only a warning, then the message indicates a severe error which must be corrected.

Summary

The last lines of the compiler source listing show the number of bytes available, the number of bytes free, along with the total number of warning and severe error messages. This is the same information that is displayed on the screen when the compiler finishes.

Bytes Available is the initial amount of compiler workspace available for storing the symbol table and the line number table, and for working storage for optimization and code generation.

Bytes Free is the size of unused compiler workspace after the compiler has finished. If this number is less than 1024, then you are approaching the maximum program size for the amount of memory in your computer. You cannot increase the size of your program without creating an error. We suggest you try compiling with the /S parameter if your program uses many string constants. When your program approaches maximum size, the compiler may not be performing as many optimizations as possible. To increase optimization, you may need to increase the amount of memory in your computer.

Linking

The .OBJ file created by the compiler is not executable, and needs to be linked to the appropriate library. Linking is the process of:

• Combining separately produced object (.OBJ) modules

• Searching the appropriate library file(s) for definitions of unresolved external references

• Resolving external cross-references

• Computing absolute addresses for local references within modules

• Producing an executable file on diskette

To link a compiled program, use the IBM Personal Computer Linker Version 1.10, which is the file named LINK.EXE on the BASIC diskette. The version 1.10 linker is an updated version of the IBM Personal Computer Linker Version 1.00, and should be used in place of the linker (LINK) program on the DOS version 1.00 diskette.

LINK produces as output an executable object file with .EXE as the filename extension. The modules you link together need not all have been written in BASIC; some may be assembled modules created with the Macro-Assembler.

You will probably want to refer to Appendix B of this book for information on how to use the linker before you read this section.

Preliminary Steps

The linker needs to have the following files available on diskette:

- Your object file (created by the compiler)

- The BASIC library being used:

  - The BASCOM.LIB library, if you used the /O compiler parameter during the compile step
  - The BASRUN.LIB library, if you did *not* use the /O compiler parameter (that is, if you elected to use the BASRUN.EXE runtime module)

  The two libraries and the /O parameter are discussed in more detail earlier, in the section called "Compiler Parameters."

- Any other modules you may be linking to your program

  - If you use communications files and compiled with the /O parameter, you need IBMCOM.OBJ

If your system has a single diskette drive, you must copy the library needed by the linker (BASCOM.LIB or BASRUN.LIB) from the LIBRARY diskette to the work diskette, where your object file is.

If you have a two-drive system and want to start the linker with the single command line, you will find it easiest if you have created a Library-Linker diskette as described under "Diskette Setup for a Two-Drive System," earlier in this book. When you enter the command line, the linker must be in a drive, in addition to the files listed above. The suggestions that follow for a two-drive system assume you have created such a diskette with the appropriate library.

Starting the Linker

Like the compiler, you can start the linker in either of two ways:

- You can let the linker prompt you for the information it needs. This lets you change diskettes before you proceed with answering the prompts.

- Entering the command line is a fast way to start the linker. Remember that to do this you must have the linker itself (LINK.EXE) available on diskette.

In this section we will explain how to answer the linker prompts for a compiled BASIC program. Once you know that, you can refer to Appendix B of this book for an explanation of starting the linker using the command line.

Answering the Linker Prompts

The linker is started as follows:

Start DOS if it's not already active.

If your system has a single diskette drive, insert the BASIC diskette in drive A: and enter:

If you have a two-drive system, the default drive should be B:. Insert your Library-Linker diskette in drive A:, and your work diskette in drive B:. Enter:

(The linker looks for the library(s) on drive A:, regardless of the default diskette drive. That is why it is convenient to have B: as your default drive, and use the Library-Linker diskette in drive A:.)

54

Once started, the linker program asks for the following:

Object Modules refers to the program(s) you want to link together. Enter the name of your object file (not the listing file). The linker will add the default extension .OBJ if you don't include a filename extension of your own. For example:

If you are linking more than one module (for example, if you are using the CALL statement), you enter the names of additional modules on the same line, separated by plus signs (+). Refer to Appendix B of this book for details.

The next prompt will be:

Run File is the name you want to give to the file containing the executable code for your program. If you want the linker to use the default name (the name given in the brackets – MYPROG.EXE in this example), just press Enter. If you want to give the run file a different name, enter the filename next to this prompt. This filename will be given the extension .EXE, even if you specify another extension.

The next prompt is:

List File is the name of the linker listing file. If you don't want a listing, just press Enter. If you do want a listing, enter the filename you want it to have after this prompt. The default extension is .MAP.

The last prompt from the linker is:

You may simply press Enter. The version 1.10 linker knows which library it needs (BASCOM.LIB or BASRUN.LIB). You may enter the library name here if you wish, but it is not necessary.

You should let all the optional linker parameters (discussed in Appendix B of this book) default to their normal settings.

When the Linker Finishes

After you enter the complete command line or respond to the last prompt, the linker begins to link the program. When linking has been completed, control returns to DOS. If any errors occurred during linking you should not try to execute your program. See "Appendix B. The Linker (LINK) Program" for a list of messages from LINK.

If linking was successful, you should have the executable run file stored on your work diskette. We recommend that you display the diskette directory for the work diskette to confirm that the run filename is there (it will have the .**EXE** filename extension). Using our example filename, you should see **MYPROG.EXE** in the directory.

Running a Program

The executable object file can be executed, like any file with an extension of .EXE, by simply entering the file's base name (the filename without the .EXE extension).

For example:

The above command executes the program DEMO.EXE.

The executable file can also be executed from within a program, as in the following statement:

The default extension is .EXE. The .EXE file can be an executable file created in any programming language. The CHAIN command is used in a similar fashion. In either case, an executable binary file is loaded.

Considerations When Using the BASRUN.EXE Runtime Module: If the program was compiled without using the /O compiler parameter, then the file BASRUN.EXE (from the LIBRARY diskette) must be accessible on diskette when you run your program. Your executable program first looks on the default drive for BASRUN.EXE, then on drive A:. If the file is not found, the following message is displayed:

You should respond with the letter for the drive where the BASRUN.EXE module is. If you respond incorrectly, the message will be displayed again. You can press Ctrl-Break to exit the prompt.

When you run any compiled program, the executable file created by the linker is loaded into memory. If the runtime module is required, BASRUN.EXE is also loaded and both files reside in memory at the same time. That is, 30K-bytes of memory are taken up by the runtime module itself.

If you chain to another program using the CHAIN statement, the runtime module (BASRUN.EXE) is not reloaded. However, the runtime module is reloaded when you use RUN.

Using a Batch File

The Batch command facility of the Disk Operating System can be a very convenient way to automatically start the BASIC Compiler and the linker. See the *IBM Personal Computer Disk Operating System* manual for detailed information of the Batch command facility.

Several sample batch files are provided on the BASIC diskette. All the batch files have the extension .**BAT** and were created using EDLIN. They contain remarks to document how they work. Three of these batch files, SAMPLE.BAT, CREATE.BAT, and COM.BAT are explained in this section.

SAMPLE.BAT

SAMPLE.BAT can be used to compile, link, and run the DEMO.BAS program presented in "Sample Session." This batch file assumes a two-drive system with the BASIC diskette in drive A: and the work diskette in drive B:. Also, you need the BASRUN-Linker diskette to use for the link step, with the file BASRUN.EXE on the BASRUN-Linker diskette.

SAMPLE.BAT

To run this batch file with the demonstration program DEMO.BAS, enter the following in response to the DOS prompt:

The filename "DEMO" is used for the %1 replaceable parameter and "/E" for the %2 replaceable parameter. The compiler translates DEMO.BAS and produces DEMO.OBJ and DEMO.LST. After the directory of the work diskette is displayed, the source listing file (DEMO.LST) is copied to the printer and then erased from the diskette. The PAUSE command then displays a message, and the system waits until you press a key before continuing. This allows you to change the diskette in drive A:. The batch file then calls the linker, which produces the executable file DEMO.EXE. After the directory is listed, the runfile, DEMO.EXE, is executed.

> **Note:** When using batch files, the batch file must be accessible each time a statement in the batch file is to be executed. If the batch file is not found, DOS gives the following message:
>
>
> When changing diskettes, as in SAMPLE.BAT, it is convenient to have the batch file on both the BASIC and BASRUN-Linker diskette.

CREATE.BAT

CREATE.BAT can be used to create and compile a BASIC source program. The process is repeated until you want to stop, which would normally be when you get an error-free compilation. This batch file assumes a single-drive system. It needs a diskette containing the batch file, CREATE.BAT; the file BASCOM.COM from the BASIC diskette; the file EDLIN.COM from the DOS diskette; and space for the BASIC source program you want to create.

CREATE.BAT

To run this batch file, enter the following in response to the DOS prompt:

Here, *progname* is the name of the BASIC source program you want to create. Do not enter the extension **.BAS**, as it is already included in the batch file. *parms* are whatever compiler parameters you wish to include.

The first thing CREATE.BAT does is start EDLIN. Once it is started, you use it as you normally would to create your BASIC source program. For information on how to use EDLIN, refer to the *IBM Personal Computer Disk Operating System* manual. Remember to include line numbers!

After you exit EDLIN (by entering **E**), your program is saved as an ASCII file, and control is returned to the batch file. The BASIC Compiler then compiles the program you just created, but does not produce an object file. Any compiler errors are displayed on the screen. The PAUSE command then allows you to make a choice. If the compilation was successful, you can press Ctrl-Break to exit the batch file. Otherwise, CREATE.BAT is called again, allowing you to make corrections and recompile.

COM.BAT

COM.BAT can be used to compile, link, and run a
BASIC source program which uses the BASIC
communications features. This batch file assumes a
two-drive system with the BASIC diskette in drive A:
and the work diskette in drive B:. Also, you need the
BASCOM-Linker diskette to use for the link step, and
the file IBMCOM.OBJ copied from the LIBRARY
diskette to the BASCOM-Linker diskette.

COM.BAT

To run this batch file, enter the following in response
to the DOS prompt:

progname is the name of the BASIC source program
you want to compile. *parms* are whatever additional
compiler parameters you wish to include.

This batch file works similarly to SAMPLE.BAT, but
note that IBMCOM.OBJ is linked with *progname*.OBJ.
The communication module IBMCOM.OBJ is only
needed when your program was compiled with the
/O parameter. As in SAMPLE.BAT, after the link is
complete, the executable file is run.

62

Differences Between the Compiler and Interpreter

Differences between the languages supported by the BASIC Compiler and the BASIC interpreter must be taken into account when compiling existing or new BASIC programs. To help you become aware of these differences, we recommend that you first compile the demonstration program in the section called "Sample Session," then read the following sections, and only then begin compiling your own programs.

The differences between the languages supported by the BASIC Compiler and the BASIC interpreter are described on the following pages. The BASIC commands and functions for the interpreter are described in detail in Chapter 4 of the *IBM Personal Computer BASIC* manual. Where differences exist, the command or function is also discussed here.

Compiler Metacommands

The metacommands for the BASIC Compiler are:

$INCLUDE	$PAGEIF
$LINESIZE	$PAGESIZE
$LIST	$SKIP
$OCODE	$SUBTITLE
$PAGE	$TITLE

The metacommands are included in your source file as part of a remark statement, after the REM keyword or the single quote. You can have more than one metacommand in a remark statement, for example:

but if you use $INCLUDE, it must be the *last* meta-command on the line. Metacommands on one line may be separated by space, tab, or line feed characters; if the compiler sees any other character which is not part of a metacommand, it ignores the rest of the remark.

The header for the source listing is not printed until the compiler scans the first line of the program for metacommands; this way, metacommands such as $TITLE can affect the first page of the source listing.

$INCLUDE
Metacommand

Purpose:	Tells the compiler to include source code from another BASIC file.

Format: $INCLUDE: *'filespec'*

Remarks: *filespec*   is the file specification for the file to be included. The default extension is **.BAS**. The included file must be in ASCII format. The file specification must be enclosed in *single* quotation marks.

The compiler imbeds the specified file into the source file at the point where it encounters this metacommand. The included file may be a subroutine, a single line, or any type of partial program, but it must be written in IBM Personal Computer BASIC.

You should take care that any variables in the included files match their counterparts in the main program, and that included lines do not contain GOTOs to non-existent lines or similarly erroneous code.

Included files can be very useful for COMMON declarations existing in more than one program, or for subroutines that you might have in an external library of subroutines.

Considerations when using the BASIC program editor:
If you create the included file using the program editor from within the BASIC interpreter, you must remember to save it using SAVE with the **A** option.

Also, note that since the BASIC interpreter does not support the $INCLUDE metacommand, a program which contains a $INCLUDE metacommand may not run correctly if you try to run it under the interpreter.

$INCLUDE
Metacommand

Considerations when using another editor: Use an editor (such as EDLIN) that will save your file in ASCII format.

If you use a text editor other than the BASIC program editor, you can create a file of lines without line numbers. The compiler supports sequences of lines without line numbers if the /N parameter is specified when you start the compiler. This feature can make it very easy to include the same file in many different programs. But remember, line numbers must exist for any lines that are targets of GOTOs or GOSUBs.

Notes:

1. The included lines must be in ascending order.

2. The lowest line number of the included lines must be higher than the line number of the $INCLUDE metacommand in the main program.

3. The range of line numbers in the included file must numerically precede subsequent line numbers in the main program.

These preceding three restrictions are removed if the main program is compiled with the /N parameter, since line numbers need not be in ascending order in this case. For more information, see the section called "Compiler Parameters," earlier in this book.

4. An included file may not contain another $INCLUDE metacommand; that is, $INCLUDEs cannot be nested. This means that $INCLUDE can only be used in the file containing your main BASIC program.

However, the main program may have any number of $INCLUDEs.

5. The $INCLUDE metacommand must be the last statement on a line, as in the following statement:

Example: The first example includes the file named **SUBR.BAS**:

The second example uses the $INCLUDE metacommand in a remark beginning with a single quote. The included file is named **PROC.ASC**:

Also see the example under "COMMON Statement."

$LINESIZE
Metacommand

Purpose:	Tells the compiler to change the maximum line width in the listing file.
Format:	$LINESIZE: *number*
Remarks:	*number* is a constant in the range 40 to 255.
	The default line size is 80 characters.
	The $LINESIZE metacommand must appear in the first line of your program if you want the entire source listing to be the same width. If $LINESIZE appears anywhere else in the program, it only changes the width of the following lines.
Example:	

Purpose: Turns the listing of source code on and off.

Format: $LIST+

$LIST−

Remarks: The default setting is $LIST+.

$LIST+ turns the listing of source code on.
$LIST− turns the source code listing off.

Errors are always listed.

$LIST is useful, for instance, if you make a change to
a large program, and you want a listing of only the
change. You can cause a partial listing by using
$LIST− in the first line of the program to turn the
source code listing off, then place $LIST+ at the start
of the new code and $LIST− at the end of the new
code.

Example:

$OCODE
Metacommand

Purpose: Turns the listing of object code on and off.

Format: $OCODE+

$OCODE–

Remarks: The $OCODE metacommand controls listing the generated code in the same way $LIST controls the source listing: $OCODE+ turns the listing of object code on, $OCODE– turns the object code listing off.

$OCODE works independently of the setting of the /A parameter when you compile your program. /A includes all the object code (unless you turn it off with $OCODE–); $OCODE is used to list just parts of the object code.

The format of the object code listing is basically like an assembly listing, with code addresses and operation mnemonics.

Example:

Purpose: Tells the compiler to force a new page in the compiler
listing file.

Format: $PAGE

Remarks: The page is forced by putting the form feed character
(hex 0C) into the listing file and writing a new heading
for the page.

Example:

$PAGEIF
Metacommand

Purpose: Skips to the next page if there are less than n printable
                 lines left on the current page.

Format: $PAGEIF: n

Remarks: n is a numeric constant in the range 1 to 255.

                 The last six lines of each page are always blank. These
                 lines are not considered printable lines.

                 The default is no action.

Example:

$PAGESIZE
Metacommand

Purpose: Sets the number of lines per page in the compiler listing file.

Format: $PAGESIZE: *n*

Remarks: *n*   is a numeric constant in the range 15 to 255. The default page size is 66.

n specifies the number of lines which will fit on one piece of paper. Pages in the listing file are separated by form feed characters (hex 0C), and each page starts with a heading.

If *n* is 255, this has the effect of "infinite" page size; that is, no form feed characters are added to the listing file.

The $PAGESIZE metacommand must appear in the first line of your program if you want all the pages in your source listing to be the same length. If $PAGESIZE appears anywhere else in the program, it only changes the length of the following pages.

Example:

$SKIP
Metacommand

Purpose: Skips *n* printable lines, or to the end of the page, whichever occurs first.

Format: $SKIP: *n*

Remarks: *n*   is a numeric constant in the range 1 to 255.

               The last six lines of each page are always blank. These lines are not considered printable lines.

Example:

$SUBTITLE
Metacommand

Purpose: Sets compiler listing page subtitle to *string*.

Format: $SUBTITLE: '*string*'

Remarks: *'string'* is a character string constant, enclosed in *single* quotation marks. The maximum length of the string is 60 characters. If a program does not contain a $SUBTITLE command, the null string is used as a subtitle.

$SUBTITLE tells the compiler to head subsequent pages of the listing with the specified subtitle, until it is overridden by another $SUBTITLE metacommand. The $SUBTITLE metacommand must appear in the first line of your program if you want the subtitle to appear on the first page of the source listing.

The specified string appears as the subtitle on each page of the source listing in the upper left-hand corner on the second printed line.

Example:

$TITLE
Metacommand

Purpose: Provides a title for the compiler listing.

Format: $TITLE: '*string*'

Remarks: *'string'*   is a character string constant, enclosed in *single* quotation marks. The maximum length of the string is 60 characters.

The specified string is printed on each page of the listing. A long title string may be truncated if $LINESIZE is set to a value less than 80.

The $TITLE metacommand must appear in the first line of your program if you want the title to appear on the first page of the source listing. If $TITLE appears anywhere else in the program, it only affects the title on the following pages.

Example:

Operational Differences

Those BASIC commands and statements used to operate in the interpreter programming environment are not acceptable input to the compiler. These are:

AUTO	LOAD
CONT	MERGE
DELETE	NEW
EDIT	RENUM
LIST	SAVE
LLIST	

Also, while certain statements function similarly in the BASIC Compiler and the interpreter, they do require special parameters to be specified when you start the compiler.

- **Event trapping:** If you use any of the event trapping statements, you must specify either the /V or the /W parameter when you start the compiler. The event trapping statements are:

COM(n)	ON PEN
KEY(n)	ON STRIG(n)
ON COM(n)	PEN STOP
ON KEY(n)	STRIG(n)

- **Error trapping:** If you use an ON ERROR statement and some form of a RESUME statement, you must specify either the /E or the /X parameter when you start the compiler. If you use only the RESUME *line* form, you should specify /E; if you use RESUME NEXT, RESUME 0, just plain RESUME, or any combination of those with RESUME *line*, the /X parameter must be used instead.

These parameters tell the compiler to generate
special extra object code. This extra code is
required for your program to run correctly when
it has an error trapping routine. Note, however,
that using these parameters increases the size of
the .OBJ and .EXE files.

- **Debug code (TRON and TROFF):** In order to
use TRON and TROFF, the /**D** parameter must be
specified when you run the compiler. Otherwise,
TRON and TROFF are ignored and a warning is
generated.

See "Compiler Parameters," earlier in this book, for a
detailed explanation of each of the compiler parameters.

Language Differences

The BASIC Compiler does not support cassette I/O, and the following statements are not supported:

ERASE
MOTOR

Other than that, though, most programs that run under the BASIC interpreter will compile under the BASIC Compiler with little or no change. However, differences exist in the following commands, statements, and functions:

CALL	FOR...NEXT
CHAIN	FRE
CLEAR	KEY
COMMON	PLAY
DEF FN	REM
DEF*type*	RUN
DIM	STOP
DRAW	USR
END	WHILE...WEND

Also, the BASIC Compiler includes some statements and functions which do not exist in or have new function compared to the BASIC interpreter release 1.00:

OPEN	VARPTR$
OPEN "COM...	WIDTH
STRIG	

All the differences and the new statements and functions are described on the following pages.

CALL
Statement

CALL Macro-Assembler Routine

Purpose: Calls and transfers program control to an assembly language routine created with the Macro-Assembler.

> **Note:** The Macro-Assembler is a separate product available from IBM and is not part of the BASIC Compiler package. If you do not have the Macro-Assembler, then you can't use this form of CALL.

Format: CALL *subrtname* [(*variable*[,*variable*]...)] ·

Differences: *subrtname* is the name of the subroutine that you wish to call.

subrtname must be recognized by LINK as a global symbol. That is, the routine must be compiled separately and linked to your program during the linking step; and *subrtname* must be a name declared in a PUBLIC statement in Macro-Assembler language. The maximum length of a global name is 31 characters.

The starting address of the subroutine is determined by the linker; DEF SEG is unimportant when calling a Macro Assembler subroutine from a compiled program.

The *variables* are optional and are limited in number to 60. They are the arguments that are passed to the assembly language subroutine. They are passed as unsegmented addresses according to the usual

conventions, as discussed in Appendix C of the *IBM Personal Computer BASIC* manual. Also as mentioned in that appendix, the subroutine is a FAR procedure, so the return to BASIC must be by an inter-segment RET instruction.

Note, however, that since the BASIC Compiler allows strings to be up to 32767 bytes long, the string descriptor requires four bytes rather than three (low byte, high byte of the length, followed by low byte, high byte of address). If your assembly language routine uses string arguments, you may need to recode it to take this difference into account.

Example: The following program, REC.BAS, calls an assembly language routine called SORT, which sorts 50 integers:

In line 50, the addresses of the first and last integer to be sorted are passed as arguments to SORT. SORT was declared PUBLIC in the Macro-Assembler, and was linked to REC.BAS by responding to the **Object Modules** prompt from the linker as follows:

CALL
Statement (ABSOLUTE)

CALL ABSOLUTE

Purpose: Transfers control to a machine language subroutine.

Format: CALL ABSOLUTE ([*variable*[,*variable*]. . .,]*intvar*)

Differences: *intvar*   is the name of an integer variable.

The word ABSOLUTE must be used. It is a global symbol which is the name of a library routine that transfers control to your subroutine.

intvar must be included in the variable list. The value of *intvar* is the starting memory address of the subroutine as an offset into the current segment of memory as defined by the last DEF SEG statement. *intvar* is an argument to the ABSOLUTE routine, but is not passed as an argument to your machine language subroutine. If other variables are included in the list *intvar* must appear last. The value of *intvar* must be set to the offset value before the CALL statement is executed. A DEF SEG statement *must* be executed before the CALL ABSOLUTE statement is performed to ensure that the code segment is correct.

The *variables* are optional. They are the arguments that are passed to the machine language subroutine.

The BASIC interpreter allows you to load the machine language subroutine into several types of areas. If your program uses one of the following areas, then you don't need to make any change to the way you loaded the subroutine with the interpreter:

- An unused screen buffer

- A string variable area located by peeking at the string descriptor, which is found using VARPTR.

  Note: You cannot use an unused file buffer, since the compiler has no preallocated set of file buffers.

In a compiled program, you can put the routine into an integer variable array by following these steps:

1. Dimension an integer array so the number of elements in the array is the number of *words*, not bytes, in the subroutine.

2. Use a FOR...NEXT loop to read the hex values for the machine language code from DATA statements into the array. Remember that, since integers are stored low byte first, you must code the hex values "backwards."

3. Remember to call VARPTR (*arrayname*) to define the offset before you perform the CALL ABSOLUTE.

   To preserve the array across chaining, define the array in COMMON.

Example: For the interpreter:

For the compiler:

CHAIN
Statement

Purpose: Passes control to another program.

Format: CHAIN *filespec*

Differences: The BASIC Compiler does not support the ALL, MERGE, DELETE, and *line* options to CHAIN. If you wish to share variables between programs, we recommend that you use the COMMON statement. However, note that the BASIC Compiler only supports CHAIN with COMMON when you use the BASRUN.EXE runtime module; that is, when you compile without the /O parameter.

The CHAIN statement performs two different ways, depending on whether you are using the BASRUN.EXE runtime module or not (whether you omitted or used the /O compiler parameter).

When you use BASRUN.EXE, chaining works as in the interpreter. Files are left open, device status is preserved, and you can use COMMON to pass variables to the chained-to program. Both the chaining program and the chained-to program must have been compiled without the /O parameter.

In a program which doesn't use BASRUN.EXE, CHAIN works just like RUN *filespec*, with the exception that both the chaining program and the chained-to program must have been compiled with /O.

CLEAR
Command

Purpose: Performs the following actions:

- Closes all files

- Clears all COMMON variables

- Resets the stack and string space

- Releases all diskette buffers (that is, space allocated for file buffers is returned to string space)

- Resets all variables and arrays to zero or null

- Resets DEF SEG to the default and clears the definitions of any USR functions

Format: CLEAR [,[n] [,m]]

Differences: n and m must be integer expressions. If a value of 0 is given for either expression, the appropriate default is used.

n is a byte count which, if specified, sets maximum number of bytes available for BASIC to store programs and data. The default is the current size. m specifies stack space for BASIC. The default is 512 bytes.

Since type definitions are determined when the program is compiled, the type definitions set by DEFtype statements are not cleared. This differs from the interpreter.

DIFFERENCES

83

COMMON
Statement

Purpose: Passes variables to a chained or called program.

Format: COMMON *variable* [*,variable*] . . .

Differences: The COMMON statement must appear in a program
before any executable statements. The non-
executable statements for the BASIC Compiler are:

    COMMON
    DEF*type*
    DIM
    OPTION BASE
    REM
    all compiler metacommands

All other statements are executable.

Arrays in COMMON must be declared in preceding
DIM statements. Also, if there are any variables in
the COMMON statement whose types are defined
by a DEF*type* statement, the DEF*type* statement
must precede the COMMON statement.

When you use COMMON to communicate with a
chained-to program, you must use the BASRUN.EXE
runtime module (compile without the /O parameter).
Also, both the chaining program and the chained-to
program require a COMMON statement. The *order*
of variables in the two statements must be the same.
The common variables must be common to *all*
programs you're chaining to. If the size of the
COMMON in the chained-to program is smaller,
then the extra common variables will be ignored. If
the common size is larger, the additional common
variables will be initialized to zero or null.

84

COMMON
Statement

When you compile with the /O parameter,
COMMON may only be used to pass variables to
assembly language subroutines. To reference variables
in COMMON your assembly language subroutine
needs this segment definition;

COMMON SEGMENT COMMON 'BLANK'

and this group definition record;

DGROUP GROUP COMMON

A convenient way to share COMMON areas between
programs is to place COMMON declarations and
necessary preceding DIM statements in a single
included file and use the $INCLUDE metacommand
in each program. For example:

MENU.BAS

PROG1.BAS

COMDEF.BAS

COMDEF.BAS shows how arrays are passed by
adding ( ) to the array name.

DEF FN
Statement

Purpose: Defines and names a function that you write.

Format: DEF FN*name*[(*arg* [,*arg*]...)] =*definition*

Differences: A DEF FN statement must *physically*, not just
logically, precede a call to the defined function. As
it reads your program from the beginning to the end,
the compiler must actually see the function
definition before it sees any call to the function.

Also, the compiler allows a maximum of 60
arguments to the function.

DEFtype
Statements

Purpose:	Declares variable types as integer, single-precision, double-precision, or string.

Format: DEF*type letter*[*–letter*] [*,letter* [*–letter*]] . . .

where *type* is INT, SNG, DBL, or STR.

Differences: The compiler does not translate a DEF*type* statement into executable code as it does a PRINT statement, for example.

Instead, the compiler allocates memory for storage of the designated variables, and assigns them to the appropriate data type (integer, single-precision, double-precision, or string).

A DEF*type* statement takes effect as soon as it is encountered in your program *during compilation*. Once the type has been defined for the listed variables, that type remains in effect either until the end of the program or until another DEF*type* statement changes the type of the variable. Unlike the interpreter, you cannot make the compiler branch around the DEF*type* statement (or any statement) by using a GOTO.

As with the interpreter, variables named with a type declaration character (%, !, #, or $; as in A%=B) are not affected by the DEF*type* statement.

DIM
Statement

Purpose:	Specifies the maximum values for array variable subscripts and allocates storage accordingly.
Format:	DIM *variable(subscripts)* [,*variable(subscripts)*] . . .
Differences:	The value of the subscripts in a DIM statement must be integer *constants*; they may not be variables, arithmetic expressions, or single- or double-precision values. For example, each of the following DIM statements is *invalid*:

Also, the DIM statement is similar to the DEF*type* statement in that it affects the compiler but does not get translated into executable code. That is, DIM takes effect when the compiler first encounters it and remains in effect until the end of the program: it does not get executed again when the program is run. If the default dimension (10) has already been established for an array variable, and that variable is later encountered in a DIM statement, a "Duplicate Definition" error results. Therefore, the practice of putting a collection of DIM statements in a subroutine at the end of your program generates severe errors. This is because the compiler sees the DIM statement only after it has already assigned the default dimension to arrays encountered earlier in the program.

The compiler allows a maximum of 60 dimensions to an array.

Purpose: Draws an object as specified by *string*.

Format: DRAW *string*

Differences: *string* must be a string expression consisting of drawing commands as described in Chapter 4 of the *IBM Personal Computer BASIC* manual under "DRAW Statement." However, you may not use variable names in the string, either in the X drawing command or using the form *=variable*; for *n*.

You can get the identical function, however, by using VARPTR$(*variable*) instead of the variable name. For example:

    Interpreter   **Compiler Equivalent**

END
Statement

Purpose: Terminates program execution, closes all files, and returns to the system.

Format: END

Differences: When an END statement is encountered while a compiled program is running, control returns to the Disk Operating System, rather than BASIC command level. Files are closed, as in the interpreter.

The screen mode is reset to the initial screen mode (the mode and width the screen was in when you left DOS to begin the program).

FOR and NEXT
Statements

Purpose:	Performs a series of instructions in a loop a given number of times.
Format:	FOR *variable*=*x* TO *y* [STEP *z*] • • • NEXT [*variable*] [,*variable*] . . .
Differences:	The counter *variable* may be double-precision. Also, FOR...NEXT loops must be statically nested. Static nesting means that each FOR must have a single corresponding NEXT. Static nesting also means that each FOR...NEXT pair must *physically* (not just logically) reside within an outer FOR...NEXT pair. Therefore, the following construction is *not* allowed:

```
        FOR I ◄─────────────────┐
          FOR J ◄──────────────┐ │
            FOR K ◄──────────┐ │ │
                             │ │ │
          NEXT J ◄───────────│─┘ │
            NEXT K ◄─────────┘   │
        NEXT I ◄─────────────────┘
```
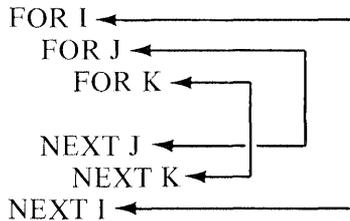
This construction is the correct form:

```
        FOR I ◄─────────────────┐
          FOR J ◄──────────────┐ │
            FOR K ◄──────────┐ │ │
                             │ │ │
            NEXT K ◄─────────┘ │ │
          NEXT J ◄─────────────┘ │
        NEXT I ◄─────────────────┘
```
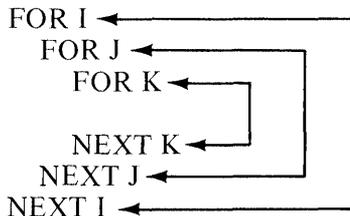
FOR and NEXT
Statements

Also, you should not direct program flow into a
FOR...NEXT loop with a GOTO statement. The
result of such a jump is undefined, as in the
following example:

FRE
Function

Purpose: Returns the size of the current block of free string space.

Format: $v = \mathrm{FRE}(x)$

$v = \mathrm{FRE}(x\$)$

Differences: FRE with a numeric argument returns the size of the current block of free string space. Normally, this is also the largest block of string space.

If a string allocation exceeds the size of the current block, then BASIC either finds a free block large enough to hold the string, or it does a housecleaning. This function can be used as an indicator of when a housecleaning may take place.

FRE with a string argument forces a housecleaning before returning the size of the current block of string space. After a housecleaning, all free space is collected in one block (the current block) of string space.

Note that the compiler maintains string space differently than the interpreter, so its housecleaning is faster than the interpreter.

KEY
Statement

Purpose:	Sets or displays the soft keys.
Format:	KEY ON
	KEY OFF
	KEY LIST
	KEY *n, x$*
Differences:	The function keys are initially disabled as soft keys. That is, their initial value is null.

OPEN
Statement

Purpose:	Allows I/O to a file or device.

Format: OPEN *filespec* [FOR *mode*] AS [#]*filenum*
[LEN=*recl*]

or:

OPEN *mode2*, [#]*filenum*, *filespec* [,*recl*]

Differences: In BASIC release 1.00, printer devices (LPT1:,
LPT2:, LPT3:) were opened for sequential output
even if random mode was specified.

In a compiled program, a printer may be opened in
random mode. Opening the printer in random mode
suppresses the automatic line feed after a carriage
return (CHR$(13)) is received, if the file width is
255. This allows all 256 ASCII characters to be
passed to the printer without change.

Also, there is no equivalent in the compiler for the
/S: option on the BASIC command used to start
Disk and Advanced BASIC from DOS. The buffer
size for random files is determined by the **LEN=**
parameter on the OPEN statement; there is no
maximum record length.

Example:

OPEN "COM...
Statement

Purpose:	Opens a communications file.

Format: OPEN "COM*n*: [*speed*] [,*parity*] [,*data*] [,*stop*]
[,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF]"
AS [#] *filenum* [LEN=*number*]

Differences: If you compile with the /O parameter and use
communications files, you need to include
IBMCOM.OBJ in your list of object modules when
you link.

The **RS**, **CS[*n*]**, **DS[*n*]**, **CD[*n*]**, and **LF** options are
not available in the interpreter release 1.00. These
options perform as follows:

RS	suppresses RTS (Request To Send)
CS[*n*]	controls CTS (Clear To Send)
DS[*n*]	controls DSR (Data Set Ready)
CD[*n*]	controls CD (Carrier Detect)
LF	sends a line feed following each carriage return

The CD (Carrier Detect) is also known as the RLSD
(Received Line Signal Detect).

n in each of the above options may range from 0 to
65535.

Note: The *speed*, *parity*, *data*, and *stop*
parameters are positional, but **RS**, **CS**, **DS**, **CD**,
and **LF** are not.

96

OPEN "COM...
Statement

The RTS (Request To Send) line is turned on when you execute an OPEN "COM. . . statement unless you include the **RS** option. When you specify **RS, CS0** is the default.

Normally I/O statements to a communications file will fail if the CTS (Clear To Send) or DSR (Data Set Ready) are off. The system waits one second before returning a "Device Timeout." The **CS** and **DS** options allow you to avoid this problem by ignoring these lines. If the n argument is included, it specifies the number of milliseconds to wait for the signal before returning a "Device Timeout" error. If n is omitted or is equal to zero, then the line status is not checked at all.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM. . . statement is executed. The **CD** option allows you to test this line by including the n parameter, in the same way as **CS** and **DS**. If n is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the **CD** option).

The **LF** parameter is intended for those using communication files as a means of printing to a serial line printer. When you specify **LF**, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). INPUT# or LINE INPUT#, when used with a communications file opened with the **LF** option, stop when they see a carriage return. The line feed is always ignored.

OPEN "COM...
Statement

number is the maximum number of bytes which can be read from the communication buffer when using GET or PUT. The default is 128 bytes. This option is included because the compiler has no equivalent for the /S: option on the BASIC command used to start Disk or Advanced BASIC from DOS.

Example:

Opens COM1: at 9600 bps with no parity and eight data bits. CTS, DSR, and RLSD are not checked.

Opens COM1: at 1200 bps with the defaults of even parity and seven data bits. RTS is sent, CTS is not checked, and "Device Timeout" is given if DSR is not seen within two seconds. Note that the commas are required to indicate the position of the *parity*, *start*, and *stop* parameters, even though they are omitted. This is what is meant by *positional* parameters.

OPEN "COM...
Statement

An OPEN statement may be used with an ON
ERROR statement to make sure a modem is working
properly before sending any data. For example, the
following program makes sure we get Carrier Detect
(RLSD) from the modem before starting. Line 20
is set to timeout after 10 seconds. TRIES is set to 6
so we give up if Carrier Detect is not seen within
one minute. Once communication is established,
we re-open the file with a shorter delay until
timeout.

The next example shows a typical way to use a
communication file to control a serial line printer.
The **LF** parameter in the OPEN statement ensures
that lines do not print on top of each other.

PLAY
Statement

Purpose:	Plays music as specified by *string*.
Format:	PLAY *string*
Differences:	*string* must be a string expression consisting of music commands as explained in Chapter 4 of the *IBM Personal Computer BASIC* manual under "PLAY Statement." However, you may not use variable names in the string, either in the X music command or using the form *=variable*; for *n*.
	You can get the identical function, however, by using VARPTR$(*variable*) instead of the variable name. For example:

Interpreter	Compiler Equivalent

Also, the interpreter allows a maximum of 32 notes to be buffered in "Music Background" mode. This is increased to 256 notes in a compiled program.

Example:

The delay loop in line 40 is needed for a compiled program. Otherwise, when program control returns to DOS, all music currently executing is terminated.

REM
Statement

Purpose: Inserts explanatory remarks in a program.

Format: REM *remark*

Differences: Remarks are significant when they contain compiler metacommands.

When the compiler sees a REM statement in your source file, it doesn't create any object code corresponding to the remark. So remarks don't take up time or space when your compiled program is running. Therefore, REM may be used as freely as you wish in your source program. Using REM statements is a good idea to improve the readability of your programs.

RUN
Command

Purpose:	Transfers control to another program, or restarts the current program at the given line number.

Format: RUN [*line*]

RUN *filespec*

Differences: The program to be run (specified by *filespec*) must be an executable (.EXE) file. The default extension .EXE is supplied. Any .EXE file can be executed using the RUN command, even if it was not created using the BASIC Compiler. It may be an .EXE file created in another language besides BASIC. However, you cannot run a BASIC *source file* from your compiled program.

It is okay for a program compiled using /O to use RUN *filespec* to transfer control to a program which was compiled *without* the /O parameter, and vice versa. But remember, BASRUN.EXE must be accessible whenever you run a program which uses the runtime module (was compiled without /O).

RUN *filespec* loads another executable file into memory and transfers control to it. The new program overlays the current program in memory. The BASIC Compiler does not support the **R** option with RUN. If you want the equivalent function, you should use the CHAIN statement.

RUN *line* restarts the program currently running at the specified line number. It is the same as CLEAR followed by GOTO *line*.

STOP
Statement

Purpose:	Terminates program execution and returns to the system.
Format:	STOP
Differences:	The STOP statement, like the END statement, closes all open files and returns control to DOS.
	Unlike the END statement, STOP does not restore the screen mode to where it was when you left DOS. It also displays a message telling you the hexadecimal address the program stopped at. If you compiled the program with the /D, /E, or /X parameter, then the line number the program stopped at is displayed instead.
	STOP is normally used for debugging purposes.

STRIG
Function

Purpose: Returns the status of the joystick buttons (triggers).

Format: $v = \text{STRIG}(n)$

Differences: The BASIC interpreter release 1.00 only reads two buttons from the joysticks. The compiler supports four buttons.

 n is in the range 0 to 7. The values supporting the additional buttons are:

  4 Returns –1 if button A2 was pressed since the last STRIG(4) function call, returns 0 if not.

  5 Returns –1 if button A2 is currently pressed, returns 0 if not.

  6 Returns –1 if button B2 was pressed since the last STRIG(6) function call, returns 0 if not.

  7 Returns –1 if button B2 is currently pressed, returns 0 if not.

USR
Function

Purpose:	Calls a machine language subroutine that was defined by DEF USR.

Format: $v = USR[n](x)$

Differences: Although the USR function is implemented in the compiler to call machine language subroutines, there is no way to pass parameters to the subroutine, except through the use of POKEs to memory locations that are later accessed by the machine language routine. That is, x is a dummy argument.

As with the interpreter, the starting address of the subroutine is determined by the addresses given in the DEF SEG and DEF USR statements. The DEF USR statement specifies the address as an offset into the code segment specified by the last DEF SEG statement executed. A DEF SEG statement *must* be executed prior to a USR call to assure that the code segment points to the subroutine being called.

The USR function returns the integer result in the BX register.

The machine language subroutine may be loaded into memory, as it was in the interpreter, using a BLOAD command. Refer to "CALL ABSOLUTE," earlier, for details.

USR
Function

If the subroutine is one of the modules linked during the linking step, the DEF SEG and DEF USR values must still be set, using the information given in the linker map.

As mentioned in Appendix C of the *IBM Personal Computer BASIC* manual, the subroutine is a FAR procedure, so the return to BASIC must be by an inter-segment RET instruction. However, note that since the BASIC Compiler allows strings to be up to 32767 bytes long, the string descriptor requires four bytes rather than three (low byte, high byte of length, followed by low byte, high byte of address). If your machine language routine uses string arguments, you may need to recode it to take this difference into account.

If you have the IBM Personal Computer Macro Assembler, it's probably easiest to assemble your subroutines and then link them directly to the compiled program. Then you can use the CALL statement to reference the routine. CALL does not require the segment and offset values to be identified in your BASIC program – the linker automatically takes care of that for you.

VARPTR$
Function

Purpose:	Returns a character form of the address of a variable in memory. It is primarily for use with PLAY and DRAW in compiled programs.
Format:	$v\$$ = VARPTR$(*variable*)
Differences:	VARPTR$ is not supported by the interpreter release 1.00.

variable   is the name of a variable existing in the program.

VARPTR$ returns a three-byte string in the form:

Byte 0	Byte 1	Byte 2
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2   integer
3   string
4   single-precision
8   double-precision

The returned value is the same as:

CHR$(*type*)+MKI$(VARPTR(*variable*))

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. Refer to "DRAW Statement" and "PLAY Statement" for examples.

WHILE and WEND
Statements

Purpose:	Executes a series of statements in a loop as long as a given condition is true.

Format: WHILE *expression*
     •
     •
     •
     (*loop statements*)
     •
     •
     WEND

Differences: WHILE...WEND constructions must be statically nested. Static nesting means when you nest WHILE...WEND pairs, the inner loop must reside completely inside the outer loop. The nesting must be *physical*, not just logical, as in the interpreter.

> **Note:** When FOR...NEXT loops are nested within WHILE...WEND loops, or vice-versa, the inner loop must still reside completely inside the outer loop. For example, the following construction is *not* allowed:

```
FOR I = 1 to 10
    A = COUNT
    WHILE A = 1
NEXT I
    A = A - 1
    WEND
```

> You should also not direct program flow into a WHILE...WEND loop without entering through the WHILE statement. See "FOR and NEXT Statements," earlier in this book, for an example of this restriction. It is okay, however, to branch *out* of a WHILE...WEND loop.

WIDTH
Statement

Purpose:	Sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.
Format:	WIDTH *size*
	WIDTH *filenum,size*
	WIDTH *device,size*
	WIDTH LPRINT,*size*
Differences:	The WIDTH LPRINT format is not available in the interpreter. WIDTH LPRINT,*size* does the same thing as WIDTH "LPT1:",*size*, and is included in the compiler for compatibility with some BASICs other than IBM Personal Computer BASIC.

Other Differences

Other differences between the BASIC interpreter and the BASIC Compiler include the following:

Double-Precision Arithmetic Functions

If you use double-precision operands for any of the arithmetic functions, including the transcendental functions (SIN, COS, TAN, ATN, LOG, EXP, and SQR), then the BASIC Compiler returns double-precision results. Only single-precision results are returned by the interpreter.

Double-Precision Loop Control Variables

The compiler, unlike the interpreter, allows the use of double-precision loop control variables. This allows you to increase the precision of increment in loops.

Expression Evaluation

Mathematical computations have been modified in the compiler for improved speed and accuracy, so there may be slight differences in the results of single- or double-precision operations compared to the interpreter.

Also, the BASIC Compiler performs optimizations, if possible, when evaluating expressions.

During expression evaluation, the BASIC Compiler converts operands of different types to the type of the more precise operand.

The above expression causes J% to be converted to single-precision and added to A!. This double-precision result is added to Q#.

The BASIC Compiler is more limited than the interpreter in handling numeric overflow. For example, when run on the interpreter, the following statements yield 40000 for M.

That is, J% is added to I%. Because the number is too large, it converts the result into a floating point number. The result, 40000, is found and saved as the single-precision number M.

The BASIC Compiler, however, must make type conversion decisions during compilation. It cannot defer until actual values are known. Thus, the compiler generates code to perform the entire operation in integer mode and arithmetic overflow may occur. If the /D debug parameter is set, the error is detected. Otherwise, an incorrect answer is produced. One possible way to avoid this problem is to use single-precision numbers instead of integers.

Besides the above type conversion decisions, the compiler performs certain valid optimizing algebraic transformations before generating code. For example, the following program could produce an incorrect result when run:

If the compiler actually performs the arithmetic in the order shown, no overflow occurs. However, if the compiler performs I%+K% first and then adds J%, overflow does occur. The compiler follows the rules of operator precedence, and parentheses may be used to direct the order of evaluation. *No other guarantee of evaluation order can be made.*

Input Statements

The compiler limits the number of variables read by an INPUT or INPUT # statement to 60.

Also, if you try to enter more than 255 characters in response to any INPUT or LINE INPUT statement, the compiler makes the computer sound a two-tone beep.

Integer Variables

The BASIC Compiler can make optimum use of integer variables as loop control variables. To help the compiler produce faster and more compact object code, you should use integer variables as much as possible. For example, the following program executes much faster by replacing I, the loop control variable, with I%, or by declaring I an integer variable with DEFINT.

Also, it is advantageous to use integer variables to compute array subscripts. The generated code is significantly faster and more compact.

Line Editor

When you respond to an input statement in a compiled program, you do not have all the facilities of the BASIC program editor to use. The BASIC Compiler does not allow you to change lines anywhere on the screen – you may only edit the current line. Therefore, the following special program editor keys are *not* supported by the compiler:

Home
Ctrl-Home
Cursor Up
Cursor Down
Next Word (Ctrl-Cursor Right)
Previous Word (Ctrl-Cursor Left)
Ctrl-Break

If you try to use any of these keys (with the exception of Ctrl-Break) in response to an input statement, the computer will sound a two-tone beep.

Pressing Ctrl-Break in response to an input statement returns you to DOS. All files are closed and the message *** Break *** is displayed. The DOS screen mode is not restored.

In the BASIC Compiler, you can read Ctrl-Break using INKEY$. The returned code is hex 03. This capability allows you to test for Ctrl-Break in a program, or to write a program which is "Ctrl-Break-proof."

Number of Files

For the Disk and Advanced BASIC interpreter, the **/F**: option on the BASIC command from DOS sets the maximum number of files that can be open at one time. There is no equivalent to the **/F**: option for the compiler; however, you may have as many files as you want, up to a maximum of 127. You are only restricted by the size of memory.

DIFFERENCES

PEEKs and POKEs

PEEKs and POKEs into the interpreter workarea (such as DEF SEG: POKE 106,0) are interpreter dependent and will not work for compiled BASIC.

String Length

Strings can be up to 32767 characters long rather than 255 characters long. Therefore, any string function parameters which identify the location in or length of a string (which may have a maximum value of 255 in the interpreter) may now range to 32767.

The internal storage format for the string descriptor requires four bytes rather than three (low byte, high byte of the length, followed by low byte, high byte of the address). If you use machine language subroutines with string arguments, you will have to recode the subroutine to account for this change.

String Space Implementation

To increase the speed of housecleaning, the implementation of the string space for the compiler differs from its implementation for the interpreter. Using PEEK, POKE, VARPTR, or assembly language routines to change string descriptors may result in a "String Space Corrupt" error.

APPENDIXES

Contents

Appendix A. Messages

During development of a BASIC program with the
BASIC Compiler, three different kinds of errors may
occur:

- BASIC Compiler long messages

- BASIC Compiler two-character codes

- BASIC runtime errors

The BASIC Compiler long messages and two-character
codes occur when you compile your program; the
BASIC runtime errors only occur at the last step in
the development process, when you actually run
your compiled program. All these messages are listed
in this appendix.

The first part of this appendix lists error codes and
messages for the errors detected by the BASIC
Compiler. They are separated into two groups: long
messages, where a whole message is displayed; and
two-character codes, which generally indicate an error
in your program.

Errors from the Compiler

Long Messages

The long messages from the compiler are severe errors; that is, they must be corrected before you can continue.

The following errors are compiler prompt errors. When they occur, the sequence of prompts to start the compiler is restarted, giving you a chance to correct the error.

Message   Meaning

Bad filename

You entered a file specification which was not properly formed.

Enter the correct file specification.

Bad switch: /z

The character indicated by z is not a valid compiler parameter.

Omit the parameter or enter the correct parameter.

Can't create file

A compiler output file cannot be opened because there are no free directory entries on the diskette.

If there are any files on the diskette that you don't need, erase them; otherwise, use another diskette and retry the operation.

Message   Meaning

Command error: 'z'

> An error is in the command line at the
> character specified by z.
>
> Correct the command line.

The following long messages are compilation error
messages. When they occur, you must correct the
problem and restart the compiler from the
beginning.

Message   Meaning

Binary source file

> The source file you specified to the compiler
> was not in ASCII format.
>
> Make sure you specified the right file. If
> necessary, start the interpreter, load the
> file, and save it again with the **A** option.

Disk z full

> The diskette in the drive specified by z has
> no more disk storage space. If z is blank,
> then the drive is the DOS default drive.
>
> If there are any files on the diskette that you
> don't need, erase them. Otherwise, use a new
> formatted diskette and retry the operation.

File not found

> The source file you named does not exist on
> the drive specified.
>
> Check the file specification for the source file.
> If it is correct, insert the correct diskette and
> retry the operation.

Internal Error

An internal malfunction occurred in the
BASIC Compiler.

Recopy your diskette. Check the hardware
and retry the compile. If the error reoccurs,
report the conditions under which the message
appeared to your computer dealer.

Line *nnnnn* is undefined

A statement or command in the program refers
to a line which doesn't exist.

Check the line references in your program so
they all refer to actual program lines.

Memory Overflow

The compiler working memory is exhausted.
The program is too large to compile
successfully.

Try compiling the program again with the /S
parameter to reduce compiler working
memory requirements. Or break the program
up into smaller programs.

Missing NEXT for *z*

No NEXT statement was found for the
variable *z*.

Correct the static nesting of your FOR and
NEXT statements.

Two-Character Codes

The compiler lists all the two-character errors it finds
in your source listing file as follows: the compiler
outputs the line containing the error with an arrow
beneath that line pointing to the place in the line
where the error occurred, and the two-character code
for the error. In some cases, the compiler reads
ahead on a line to determine whether an error has
actually occurred. In those cases, the arrow points a
few characters beyond the error, or to the end of
the line.

Some of the two-character codes are only warning
messages; warnings do not need to be corrected before
you go on to the linking step. If a message is a
warning, it is noted in the explanation for the message.
If the explanation does not say that the message is
only a warning, then the message indicates a severe
error which must be corrected.

Code   Message and Meaning

BS   **Bad subscript**
An array reference had an invalid dimension
value (such as a non-integer value) or the
wrong number of subscripts.

CD   **COMMON duplication**
A variable appeared more than once in the
COMMON statement(s) in the program.

CN   **COMMON array not dimensioned**
An array in a COMMON statement had not
been dimensioned when the COMMON
statement was encountered.

An array passed in a COMMON statement must
be defined in a DIM statement which precedes
the COMMON statement.

Code   Message and Meaning

CO   COMMON out of order
The COMMON statement was found after executable statements in the program.

COMMON must precede any executable statements.

DD   Duplicate Definition
You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.
- The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default.

FD   Function already defined
You used DEF FN to define a function with the same name as a function previously defined in your program.

FN   FOR...NEXT error
The counter variable on a FOR statement is already in use, or a FOR statement does not have a corresponding NEXT, or a NEXT was encountered without a corresponding FOR.

IN   INCLUDE error
The file specified in the $INCLUDE meta-command could not be found.

LL   Line too long
A line has too many characters.

The line must have 253 characters or less.

Code   Message and Meaning

LS   String constant too long
You tried to create a string constant that is more than 255 characters long.

MC   Metacommand error
The format of a metacommand was invalid, or included an invalid argument. The metacommand is ignored. This message is only a warning.

ND   Array not dimensioned
Default dimensions were assigned to the array. This message is only a warning.

OM   Out of memory
A program has too many statement numbers, or program memory is too full; or data memory may be overflowing, possibly because an array is too big.

OV   Overflow
A constant was not within the range expected by the compiler. Or an expression containing constants was calculated by the compiler and resulted in an overflow. One way to correct this may be to use single-precision constants instead of integer constants.

SI   Statement ignored
The statement was ignored by the compiler. It may be that the command is unimplemented. This message is only a warning.

Code   Message and Meaning

SN   Syntax error
Caused by one of the following:

- Invalid argument name
- Invalid assignment target
- Invalid constant format
- Invalid DEF*type* character specification
- Invalid expression syntax
- Invalid function name
- Invalid function formal parameter
- Invalid separator
- Invalid format for statement number
- Invalid character
- Missing AS
- Missing equal sign
- Missing GOTO or GOSUB
- Missing comma
- Missing INPUT
- Missing line number
- Missing left parenthesis
- Missing minus sign
- Missing operand in expression
- Missing right parenthesis
- Missing semicolon
- Name too long
- Expected GOTO or GOSUB
- String assignment required
- String expression required
- String variable required
- Invalid syntax
- Variable required
- Wrong number of arguments
- Formal parameters must be unique
- Single variable only allowed
- Missing TO
- Invalid FOR loop index variable
- Missing THEN
- Missing BASE
- Invalid subroutine name

SQ **Sequence error**
The line numbers in your program were not
in sequence, or contained a duplicate statement
number.

TC **Too complex**
An expression is too complex, there are too
many arguments in a function call, there are too
many dimensions in an array, there is more than
one variable for LINE INPUT, there are too
many variables for INPUT, or the memory limit
was exceeded.

The compiler has a limit of 60 arguments for a
function call, and 60 variables in an INPUT
statement.

TM **Type mismatch**
The variable is not of the required type
(numeric or string).

UC **Unrecognizable command**
The statement is unrecognizable or the
command is not implemented. It may be that
you used a built-in function on the left side of
an equal sign.

UF **Undefined user function**
You called a function before defining it with
the DEF FN statement.

WE **WHILE...WEND error**
A WHILE does not have a matching WEND, or
a WEND was encountered before a matching
WHILE.

Code   Message and Meaning

/0   Division by zero
You tried to divide by zero, or you had a divide
overflow.

/E   Missing /E parameter
Your program included a RESUME *line*
statement.

It must be recompiled with the /E parameter.
If the listing also contains a /**X** error, recompile
using /**X** instead of /**E**.

/V   Missing /V parameter
The program contains event trapping statements.

Recompile the program using either of the event
trapping parameters, /**V** or /**W**.

/X   Missing /X parameter
Your program included a RESUME 0, RESUME,
or RESUME NEXT statement.

It must be recompiled with the /**X** parameter.

Errors while Running a Program

The following errors may occur when you run your compiled and linked program. The first group of errors can be trapped by using an ON ERROR statement. The error numbers match those issued by the BASIC interpreter. When an untrapped error occurs, the message is displayed followed by an address. If the /D, /E, or /X parameter was specified to the compiler, then the number of the line in which the error occurred is displayed instead.

Number   Message

2   **Syntax error**
         A string item was encountered in a DATA
         statement when the program wanted a
         numeric value.

         Correct the DATA statement or the READ
         statement.

         Or, you may have the wrong number of
         arguments in a COLOR, LOCATE, or
         SCREEN statement.

3   **RETURN without GOSUB**
         A RETURN statement needs a previous
         unmatched GOSUB statement.

         Correct the program. You probably need to
         put a STOP or END statement before the
         subroutine so the program doesn't "fall"
         into the subroutine code.

4   **Out of DATA**
         A READ statement is trying to read more
         data than is in the DATA statements.

         Correct the program so that there are enough
         constants in the DATA statements for all the
         READ statements in the program.

5   **Illegal function call**
     A parameter that is out of range is passed to
     a system function. The error may also occur
     as the result of:

   - A negative or unreasonably large subscript
   - Trying to raise a negative number to a
     power that is not an integer
   - Calling a USR function before defining
     the starting address with DEF USR
   - A negative record number on GET or
     PUT (file)
   - An improper argument to a function or
     statement (such as one that is out of the
     expected range for the parameter)
   - Trying to concatenate strings where the
     result is more than 32767 characters
     long

     Correct the program. Refer to Chapter 4 of
     the *IBM Personal Computer BASIC* manual
     for information about the particular
     statement or function.

6   **Overflow**
     The magnitude of a number is too large to be
     represented in the required number format.
     Unlike the interpreter, execution will always
     stop when this error occurs.

     You may be able to change the order of
     operations in a calculation so the overflow
     doesn't occur; or you may have to restrict
     the range of numbers in the program to avoid
     the overflow. To correct integer overflow,
     you may try changing to single- or double-
     precision variables.

          Note: As with the interpreter, if under-
          flow occurs, the result is zero and
          execution continues without an error.

7   **Out of memory**
        There is not enough free memory to allocate
        file buffers, communications buffers, and/or
        the music background buffer. Or you may
        be doing complex painting, or you may have
        too many GOSUBs.

9   **Subscript out of range**
        You used an array element with a subscript
        that is outside the dimensions of the array.

        Check the usage of the array variable.

11   **Division by zero**
        In an expression, you tried to divide by zero,
        you tried to raise zero to a negative power, or
        you had an integer divide overflow.

13   **Type mismatch**
        You gave a string value where a numeric value
        was expected, or you had a numeric value in
        place of a string value. This may occur in
        DRAW or PLAY with VARPTR$, or in a
        PRINT USING statement.

14   **Out of string space**
        String variables exceed the amount of
        remaining free string space after housecleaning.

16   **String formula too complex**
        A string expression is too long or too
        complex.

        The expression should be broken into smaller
        expressions.

        Or more than 15 string variables were
        requested in an input statement.

19   **No RESUME**
The physical end of the program was
encountered while the program was in an
error trapping routine.

Correct the error trapping routine so a
RESUME statement is executed. Or you
may want to add an ON ERROR GOTO 0
statement to the error trapping routine so
BASIC will display the message for any
uncaught error.

20   **RESUME without error**
The program has encountered a RESUME
statement without having trapped an error.
The error trapping routine should only be
entered when an error occurs or an ERROR
statement is executed.

You probably need to include a STOP or
END statement before the error trapping
routine to prevent the program from "falling
into" the error trapping mode.

24   **Device Timeout**
BASIC did not receive information from an
input/output device within a predetermined
amount of time.

For a communications file, this indicates that
one of the signals tested by OPEN "COM...
is off.

25   **Device Fault**
A hardware error indication was returned by
an interface adapter.

For communications files, this error may also
occur when one of the signals tested by
OPEN "COM... is lost.

27   **Out of Paper**
        The printer is out of paper, or the printer is
        not switched on.

        You should insert paper (if necessary), verify
        that the printer is properly connected, and
        make sure that the power is on. Then restart
        the program or continue the error trapping
        routine.

50   **FIELD overflow**
        A FIELD statement is attempting to allocate
        more bytes than were specified for the record
        length of a random file in the OPEN
        statement. Or, the end of the FIELD buffer
        is encountered while doing sequential I/O
        (PRINT #, WRITE #, INPUT #) to a random
        file.

        Check the OPEN statement and the FIELD
        statement to make sure they correspond. If
        you are doing sequential I/O to a random file,
        make sure that the length of the data read or
        written does not exceed the record length of
        the random file.

51   **Internal error**
        An internal malfunction occurred in the
        BASIC Compiler runtime.

        Report to your computer dealer the conditions
        under which the message appeared.

52   **Bad file number**
        A statement uses a file number of a file that
        is not open, or the file number is not in the
        range 1 to 127. Or, the device name in the file
        specification is too long or invalid, or the
        filename was too long or invalid.

Make sure the file you wanted was opened
and that the file number was entered
correctly in the statement. Check that you
have a valid file specification (refer to
"Naming Files" in Chapter 3 of the *IBM
Personal Computer BASIC* manual for
information on file specifications).

53   File not found
A KILL, NAME, FILES, or OPEN references
a file that does not exist on the diskette in
the specified drive.

Verify that the correct diskette is in the drive
specified, and that the file specification was
entered correctly. Then retry the operation.

54   Bad file mode
You tried to use PUT or GET with a
sequential file or a closed file; or to execute an
OPEN with a file mode other than input,
output, append, or random.

Make sure the OPEN statement was entered
and executed properly. GET and PUT require
a random file.

55   File already open
You tried to open a file for sequential output
or append, and the file is already opened; or,
you tried to use KILL on a file that is open.

Make sure you only execute one OPEN to a
file if you are writing to it sequentially. Close
a file before you use KILL.

57   Device I/O Error
An error occurred on a device I/O operation.
DOS cannot recover from the error.

This error may occur with communications
files, from overrun, framing, break, or parity
errors. If you are communicating using 7
or less data bits, the eighth bit will be turned
on in the byte in error.

58   File already exists
The filename specified in a NAME
statement matches a filename already in use
on the diskette.

Retry the NAME command using a different
name.

61   Disk full
All diskette storage space is in use. Files are
closed when this error occurs.

If there are any files on the diskette that you
no longer need, erase them; or, use a new
diskette. Then rerun the program.

62   Input past end
This is an end of file error. An input statement
is executed for a null (empty) file, or after all
the data in a sequential file was already input.

To avoid this error, use the EOF function to
detect the end of file.

This error also occurs if you try to read from
a file that was opened for output or append.
If you want to read from a sequential output
(or append) file, you must close it and open it
again for input.

63   Bad record number
In a PUT or GET statement, the record
number is equal to zero.

Correct the PUT or GET statement to use a
valid record number.

MESSAGES

64   Bad file name
An invalid form is used for the filename with
BLOAD, BSAVE, KILL, OPEN, NAME, or
FILES.

Check "Naming Files" in Chapter 3 of the
IBM Personal Computer BASIC manual for
information on valid filenames, and correct
the filename in error.

67   Too many files
An attempt is made to create a new file (using
OPEN) when all directory entries on the
diskette are full, or when the file specification
is invalid.

If the file specification is okay, use a new
formatted diskette and retry the operation.

68   Device Unavailable
You tried to open a file to a device which
doesn't exist. Either you do not have the
hardware to support the device (such as printer
adapters for a second or third printer), or
you have disabled the device.

69   Communication buffer overflow
A communication input statement was
executed, but the input buffer was already
full.

You should use an ON ERROR statement to
retry the input when this condition occurs.
Subsequent inputs attempt to clear this fault
unless characters continue to be received
faster than the program can process them. If
this happens there are several possible
solutions:

- Increase the size of the communications buffer using the /C parameter when you start the BASIC Compiler.

- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. (See the example in Appendix F of the *IBM Personal Computer BASIC* manual.)

- User a lower baud rate to transmit and receive.

70   Disk Write Protect
You tried to write to a diskette that is write-protected.

Make sure you are using the right diskette. If so, remove the write protection, then retry the operation.

71   Disk not Ready
The diskette drive door is open or a diskette is not in the drive.

72   Disk Media Error
The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad.

Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.

—   Unprintable error
This message occurs whenever an error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code.

Check your program to make sure you handle all error codes which you create.

Errors That Cannot Be Trapped

The following additional runtime error messages are unrecoverable and cannot be trapped:

Message   Meaning

Error in EXE file
> The indicated file is in the wrong format. It should be an executable (.EXE) file. This may happen with RUN, CHAIN, and when loading BASRUN.EXE.
>
> This error also occurs when a program which uses the BASRUN.EXE runtime module tries to chain to an executable program which does not use BASRUN.EXE.

Internal Error — No Line Number
> This error occurs when the error address cannot be found in the line number table during error trapping.

Internal Error — String Space Corrupt

Internal Error — String Space Corrupt during G.C.
> These two errors usually occur because a string descriptor has been improperly modified. (G.C. stands for garbage collection, which is the same thing as housecleaning.)

Program too large
> The file is too large to load into memory. This may happen when chaining from a program which uses the runtime module, and when loading BASRUN.EXE.

Appendix B. The Linker (LINK) Program

Introduction

The linker (LINK) program is a program that:

- Combines separately produced object modules

- Searches library files for definitions of unresolved external references

- Resolves external cross-references

- Produces a printable listing that shows the resolution of external references and error messages

- Produces a relocatable load module

In this appendix, we show you how to use LINK. You should read all of this appendix before you start LINK.

Note that this appendix contains general information on the IBM Personal Computer Linker version 1.10 product; some of the information contained here may not be applicable to programs compiled with the BASIC Compiler.

Files

The linker processes the following input, output, and temporary files.

Input Files

Type	Default .ext	Override .ext	Produced by
Object	.OBJ	Yes	Compiler[1] or Macro Assembler
Library	.LIB	Yes,	Compiler
Automatic Response	(None)	N/A[2]	User

Notes:

1. One of the optional compiler packages available for use with IBM Personal Computer DOS.

2. N/A — Not applicable.

Output Files

Type	Default .ext	Override .ext	Used by
Listing	.MAP	Yes	User
Run	.EXE	No	Relocatable loader (COMMAND .COM)

VM.TMP (Temporary File)

LINK uses as much memory as is available to hold the data that defines the load module being created. If the module is too large to be processed with the available amount of memory, the linker may need additional memory space. If this happens, a temporary diskette file called VM.TMP is created on the DOS default drive.

When the overflow to diskette has begun, the linker displays the following message:

Once this temporary file is created, you should not remove the diskette until LINK ends. When LINK ends, it erases the VM.TMP file.

If the DOS default drive already has a file by the name of VM.TMP, it will be deleted by LINK and a new file will be allocated; the contents of the previous file are destroyed. Therefore, you should avoid using VM.TMP as one of your own filenames.

Definitions

Segment, *group*, and *class* are terms that appear in this appendix and in some of the messages at the end of this appendix. These terms describe the underlying function of LINK. An understanding of the concepts that define these terms provides a basic understanding of the way LINK works.

Segment

A *segment* is a contiguous area of memory up to 64K-bytes in length. A segment may be located anywhere in memory on a *paragraph* (16-byte) boundary. Each of the four segment registers defines a segment. The segments can overlap. Each 16-bit address is an offset from the beginning of a segment. The contents of a segment are addressed by a segment register/offset pair.

The contents of various portions of the segment are determined when machine language is generated.

Neither size nor location is necessarily fixed by the machine language generator because this portion of the segment may be combined at link time with other portions forming a single segment.

A program's ultimate location in memory is determined at load time by the relocation loader facility provided in COMMAND.COM, based on whether you specified the /HIGH parameter. The /HIGH parameter is discussed later in this appendix.

Group

A *group* is a collection of segments that fit together within a 64K-byte segment of memory. The segments are named to the group by the assembler or compiler. A program may consist of one or more groups.

The group is used for addressing segments in memory. The various portions of segments within the group are addressed by a segment base pointer plus an offset. The linker checks that the object modules of a group meet the 64K-byte constraint.

Class

A *class* is a collection of segments. The naming of segments to a class affects the order and relative placement of segments in memory. The class name is specified by the assembler or compiler. All portions assigned to the same class name are loaded into memory contiguously.

The segments are ordered within a class in the order that the linker encounters the segments in the object files. One class precedes another in memory only if a segment for the first class precedes all segments for the second class in the input to LINK. Classes are not restricted in size. The classes are divided into groups for addressing.

Command Prompts

After you start the linker session, you receive a series
of four prompts. You can respond to these prompts
from the keyboard, respond to these prompts on the
command line, or you can use a special diskette file
called an *automatic response file* to respond to the
prompts. An example of an automatic response file is
provided in this appendix. Refer to the section called
"How to Start the Linker Program" in this appendix
for information on how to start the linker session.

LINK prompts you for the names of the object, run,
list, and library files. When the session is finished,
LINK returns to DOS and the DOS prompt is displayed.
If linking is unsuccessful, LINK displays a message.

The prompts are described in order of their appearance
on the screen. Defaults are shown in square brackets
([ ]) after the prompt. In the response column of the
table, square brackets indicate optional entries. **Object
Modules** is the only prompt that requires a response
from you.

PROMPT	RESPONSES
Object Modules [.OBJ] :	*filespec*[+*filespec*] . . .
Run File [*filespec*.EXE] :	[*filespec*] [/P]
List File [NUL.MAP] :	[*filespec*]
Libraries [.LIB] :	[*filespec*[+*filespec*] . . .]

Notes:

1. If you enter a file specification without
   specifying the drive, the default drive is
   assumed. The libraries prompt is an excep-
   tion – the default drive for the libraries is
   determined by the compiler.

2. You can end the linker session prior to
   its normal end by pressing Ctrl-Break.

Detailed Descriptions of the Command Prompts

The following detailed descriptions contain information about the responses that you can enter to the prompts.

Object Modules [.OBJ]:

Enter one or more file specifications for the object modules to be linked. Multiple file specifications must be separated by single plus (+) signs or blanks. If the extension is omitted from any filename, LINK assumes the filename extension .OBJ. If an object module has a different filename extension, the extension must be specified. Object filenames may not begin with the @ symbol (@ is reserved for using an automatic response file.)

LINK loads segments into classes in the order encountered.

If you specify an object module, but LINK cannot locate the file, it displays the following prompt:

You should insert the diskette containing the requested module. This permits .OBJ files from several diskettes to be included. On a single-drive system, diskette exchanging can be done safely *only* if VM.TMP has *not* been opened. As explained in the discussion of the VM.TMP file earlier in this appendix, a message will indicate if VM.TMP has been opened.

IMPORTANT: If a VM.TMP file has been opened, you should *not* remove the diskette containing the VM.TMP file. Remember, once a VM.TMP file is opened, the diskette it resides on cannot be removed.

LINK

After a VM.TMP file has been opened, if you specified an
object module on the same drive that VM.TMP is on and
LINK cannot find it, the linker session ends with the
message:

Run File [*filespec*.EXE] :

The file specification you enter is created to store the
run (executable) file that results from the LINK session.
All run files receive the filename extension .EXE, even
if you specify another extension. If you specify another
extension, your specified extension is ignored.

The default filename for the run file prompt is the first
filename specified on the object module prompt.

List File [NUL.MAP] :

The linker list file is sometimes called the linker *map*.

The list file is not created unless you specifically
request it. You can request it by overriding the default
with a file specification or a device name. If you do
not include a filename extension, the default extension
.MAP is used. If you do not enter a file specification,
the DOS reserved filename NUL specifies that no list
file will be created.

The list file contains an entry for each segment in the
input (object) modules. Each entry also shows the
offset (addressing) in the run file.

Note: If the list file is allocated to a file on diskette, that diskette must not be removed until the LINK has ended.

If you specify an object module on the same drive as the list file is allocated to, and LINK cannot find it, the linker session ends with the message:

To avoid generating the list file on a diskette, you can specify the display or printer as the list file device. For example:

If you direct the output to your display, you can also print a copy of the output by pressing Ctrl-PrtSc.

Libraries [.LIB]:

You may either list the file specifications for your libraries, or just press the Enter key. If you press the Enter key, LINK defaults to the library provided as part of the Compiler package. The Compiler package also provides the location of the library. For linking objects from just the Macro Assembler, there is no automatic default library search.

If you answer the library prompt, you specify a list of drive IDs and file specifications separated by plus signs (+) or spaces. You can enter from one to eight library file specifications. A drive ID tells the linker where to look for all subsequent libraries on the library prompt. The automatically searched library file specifications are conceptually placed at the end of the response to the library prompt.

LINK

LINK searches the library files in the order in which they are listed to resolve external references. When LINK finds the module that defines the external symbol, the module is processed as another object module.

If two or more libraries have the same filename, regardless of the location, only the first library in the list is searched.

When LINK cannot find a library file, it displays the following message:

The drive that the indicated library is located on must be entered.

When you link an object module produced by the IBM Personal Computer BASIC Compiler, the linker looks on drive A: for the appropriate library (BASCOM.LIB or BASRUN.LIB). You can affect this with your response to **Libraries**. For example, suppose you compiled with the /O parameter, so the linker uses the BASCOM.LIB library. The following library prompt responses may be used:

Libraries [.LIB]: B:

> Look for BASCOM.LIB on drive B.

Libraries [.LIB]: B:USERLIB

> Look for USERLIB.LIB on drive B and BASCOM.LIB on drive A.

Libraries [.LIB]: A:LIB1+LIB2+B:LIB3+A:

> Look for LIB1.LIB and LIB2.LIB on drive A, LIB3.LIB on drive B, and BASCOM.LIB on drive A.

Linker Parameters

At the end of any of the four linker prompts, you may specify one or more parameters that instruct the linker to do something differently. Only the / and first letter of any parameter are required.

/DSALLOCATION

The /DSALLOCATION (/D) parameter directs LINK to load all data defined to be in DGROUP at the *high end* of the group. If the /HIGH parameter is specified (module loaded high), this allows any available storage below the specifically allocated area within DGROUP to be allocated dynamically by your application and still be addressable by the same data space pointer.

> Note: The maximum amount of storage which can be dynamically allocated by the application is 64K (or the amount actually available) minus the allocated portion of DGROUP.

If the /DSALLOCATION parameter is not specified, LINK loads all data defined to be in the group whose group name is DGROUP at the *low end* of the group, beginning at an offset of 0. The only storage thus referenced by the data space pointer should be that specifically defined as residing in the group.

All other segments of any type in any GROUP other than DGROUP are loaded at the low end of their respective groups, as if the /DSALLOCATION parameter were not specified.

For certain compiler packages, /DSALLOCATION is automatically used.

/HIGH

The /HIGH (/H) parameter causes the loader to place
the run image as high as possible in storage. If you
specify the /HIGH parameter, you tell the linker to
cause the loader to place the run file as high as possible
without overlaying the transient portion of
COMMAND.COM, which occupies the highest area of
storage when loaded. If you do not specify the /HIGH
parameter, the linker directs the loader to place the run
file as low in memory as possible.

The /HIGH parameter is used with the /DSALLOCATION
parameter.

/LINE

For certain IBM Personal Computer language processors,
the /LINE (/L) parameter directs LINK to include the
line numbers and addresses of the source statements in
the input modules in the list file.

/MAP

The /MAP (/M) parameter directs LINK to list all
public (global) symbols defined in the input modules.
For each symbol, LINK lists its value and segment-
offset location in the run file. The symbols are listed
at the end of the list file.

/PAUSE

The /PAUSE (/P) parameter tells LINK to display a
message to you as follows:

This message allows you to insert the diskette that is
to contain the run file.

/STACK:*size*

The *size* entry is any positive decimal value up to 65536 bytes. This value is used to override the size of the stack that the assembler or compiler has provided for the load module being created. If you specify a value greater than 0 but less than 512, the value 512 is used.

If you do not specify /STACK (/S), the original stack size provided by the assembler or compiler is used.

If the size of the stack is too small, the results of executing the resulting load module are unpredictable.

At least one input (object) module must contain a stack allocation statement. This is automatically provided by compilers. For the assembler, the source must contain a SEGMENT command that has the combine type of STACK. If a stack allocation statement was not provided, LINK returns a **Warning: No Stack statement** message.

LINK

How to Start the Linker Program

Before You Begin

- Make sure the files you will be using for linking are on the appropriate diskettes.

- Make sure you have enough free space on your diskettes to contain your files and any generated data.

You can start the linker program by using one of three options:

Option 1 — Console Responses

From your keyboard, enter:

The linker is loaded into memory and displays a series of four prompts, one at a time, to which you must enter the requested responses. (Detailed descriptions of the responses that you can make to the prompts are discussed in this appendix in the section called "Command Prompts.")

If you enter an erroneous response, such as an incorrectly spelled file specification, you must press Ctrl-Break to exit LINK, then restart LINK. If the response in error has been typed but you haven't pressed Enter yet, you may delete the erroneous characters (on that line only).

An example of a linker session using the console response option is provided in this appendix in the section called "Example Linker Session."

As soon as you have entered the last filename, the linker begins to run. If the linker finds any errors, it displays the errors on the screen as well as in the listing file.

> Note: After any of these responses, before pressing Enter, you may continue the response with a comma and the answer to what would be the next prompt, without having to wait for that prompt. If you end any with the semicolon (;), the remaining responses are all assumed to be the default. Processing begins immediately with no further prompting.

Option 2 – Command Line

From your keyboard, enter:

LINK *objlist,runfile,mapfile,liblist* [*parm*] . . .;

objlist is a list of object modules separated by spaces or plus signs (+).

runfile is the name you want to give the run file.

mapfile is the name you want to give the linker map.

liblist is a list of the libraries to be used, separated by plus signs (+) or spaces.

parm is an optional linker parameter. Each parameter must begin with a slash (/).

The linker is loaded and immediately performs the tasks indicated by the command line.

When you use this command line, the prompts described in Option 1 are not displayed if you specified an entry for all four files or if the command line ends with a semicolon.

LINK

If an incomplete list is given and no semicolon is used, the linker prompts for the remaining unspecified files. The *parms* are never prompted for, but may be added to the end of the command line or to any file specification given in response to a prompt. Each prompt displays its default, which may be accepted by pressing the Enter key, or overridden with an explicit filename or device name. However, if an incomplete list is given and the command line is terminated with a final semicolon, the unspecified files default without further prompting.

Certain variations of this command line are permitted.

Examples:

LINK module
> The object module is MODULE.OBJ. A prompt is given, showing the default of MODULE.EXE. After the response is entered, a prompt is given showing the default of NUL.MAP. After the response is given, a prompt is displayed showing the default extension of .LIB.

LINK module;
> If the semicolon is added, no further prompts are displayed. The object module of MODULE.OBJ is linked, the run file is put into MODULE.EXE, and no list file is produced.

LINK module,,;
> This is similar to the preceding example, except the list file is produced in MODULE.MAP.

LINK module,,

>Using the same example, but without the semi-colon, MODULE.OBJ is linked, and the run file is produced in MODULE.EXE, but a prompt is given with the default of MODULE.MAP.

LINK module,,NUL;

>No list file is produced. The run file is in MODULE.EXE. No further prompts are displayed.

Option 3 — Automatic Responses

It is often convenient to save responses to the linker for use at a later time. This is especially useful when long lists of object modules need to be specified.

Before using this option, you must create the automatic response file. It contains several lines of text, each of which is the response to a linker prompt. These responses must be in the same order as the linker prompts that were discussed earlier in this chapter. If desired, a long response to the object module or libraries prompt may be contained on several lines by using a plus sign (+) to continue the same response onto the next line.

To specify an automatic response file, you enter a file specification preceded by an @ symbol in place of a prompt response or part of a prompt response. The prompt is answered by the contents of the diskette file. The file specification may not be a reserved DOS filename.

From your keyboard, enter:

Use of the filename extension is optional and may be any name. There is no default extension.

Use of this option permits the command that starts
LINK to be entered from the keyboard or within a
batch file without requiring any response from you.

Example

Automatic Response File — RESP1

Automatic Response File — RESP2

Command line

Notes:

1.   The plus sign at the end of the first line in
     RESP1 causes the modules listed in the first
     two lines to be considered as the input object
     modules. After reading RESP1, the linker
     returns to the command line and sees
     +mymod, so it includes MYMOD.OBJ in
     the list of object modules as well.

2.   Each of the above lines ends when you press
     the Enter key.

Example Linker Session

This example shows you the type of information that is displayed during a linker session.

Once we enter:

in response to the DOS prompt, the system responds with the following messages and prompts, which we answer as shown:

Notes:

1. By specifying /**map**, we get both an alphabetic listing and a chronological listing of public symbols.

2. By responding **prn** to the list file prompt, we send our output to the printer.

3. By specifying the /LINE parameter, LINK gives us a listing of all line numbers for all modules. (The /LINE parameter can generate a large amount of output.)

4. By just pressing Enter in response to the Libraries prompt, an automatic library search is performed.

Once LINK locates all libraries, the linker map displays
a list of segments in the relative order of their appear-
ance within the load module. The list looks like this:

Start	Stop	Length	Name	Class
00000H	00028H	0029H	MAINQQ	CODE
00030H	000F6H	00C7H	ENTXQQ	CODE
00100H	00100H	0000H	INIXQQ	CODE
00100H	038D3H	37D4H	FILVQQ_CODE	CODE
038D4H	04921H	104EH	FILUQQ_CODE	CODE
.				
.				
.				
074A0H	074A0H	0000H	HEAP	MEMORY
074A0H	074A0H	0000H	MEMORY	MEMORY
074A0H	0759FH	0100H	STACK	STACK
075A0H	07925H	0386H	DATA	DATA
07930H	082A9H	097AH	CONST·	CONST

The information in the **Start** and **Stop** columns shows a
20-bit hex address of each setment relative to location
zero. Location zero is the beginning of the load
module. The addresses displayed are not the absolute
addresses of where these segments are loaded. To
find the absolute address of where a segment is
actually loaded, you must determine where the seg-
ment listed as being at relative zero is actually loaded;
then add the absolute address to the relative address
shown in the linker map. The procedure you use to
determine where relative zero is actually located is
discussed in this appendix, in the section called "How
to Determine the Absolute Address of a Segment."

Now, because we specified the /MAP parameter, the public symbols are displayed by name and by value. For example:

```
Address              Publics by Name

0492:0003H      ABSNQQ
06CD:029FH      ABSRQQ
0492:00A3H      ADDNQQ
06CD:0087H      ADDRQQ
0602:000FH      ALLHQQ
   .
   .
   .
0010:1BCEH      WT4VQQ
0010:1D7EH      WTFVQQ
0010:1887H      WTIVQQ
0010:19E2H      WTNVQQ
0010:11B2H      WTRVQQ

Address              Publics by Value

0000:0001H      MAIN
0000:0010H      ENTGQQ
0000:0010H      MAINQQ
0003:0000H      BEGXQQ
0003:0095H      ENDXQQ
   .
   .
   .
F82B:F31CH      CRCXQQ
F82B:F31EH      CRDXQQ
F82B:F322H      CESXQQ
F82B:F5B8H      FNSUQQ
F82B:F5E0H      OUTUQQ
```

The addresses of the public symbols are also in the *segment:offset* format, showing the location relative to zero as the beginning of the load module. In some cases, an entry may look like this:

F8CC:EBE2H

This entry appears to be the address of a load module that is almost one megabyte in size. Actually, the area being referenced is relative to a segment base that is pointing to a segment below the relative zero beginning of the load module. This condition produces a pointer that has effectively gone negative. The memory map which follows illustrates this point.

When LINK has completed, the following message is displayed:

Load Module Memory Map

Low Memory

```
                        |  .  |
                        |  .  |
Data Segment            |  .  |    (Relative to the load
    Base                |-----|    module, this location
                        |     |    is below zero, or
                        |  .  |    negative)
64K Segment             |  .  |
                        |  .  |
                        |-----|    Relative Zero
(Data elements          |     |
have large              | Data Area |
offsets from            |     |
the data seg-           |-----|
ment bases)             |     |    Load Module
                        | Code |
                        |     |
                        |-----|
                        |  .  |
                        |  .  |
                        |  .  |
```

High Memory
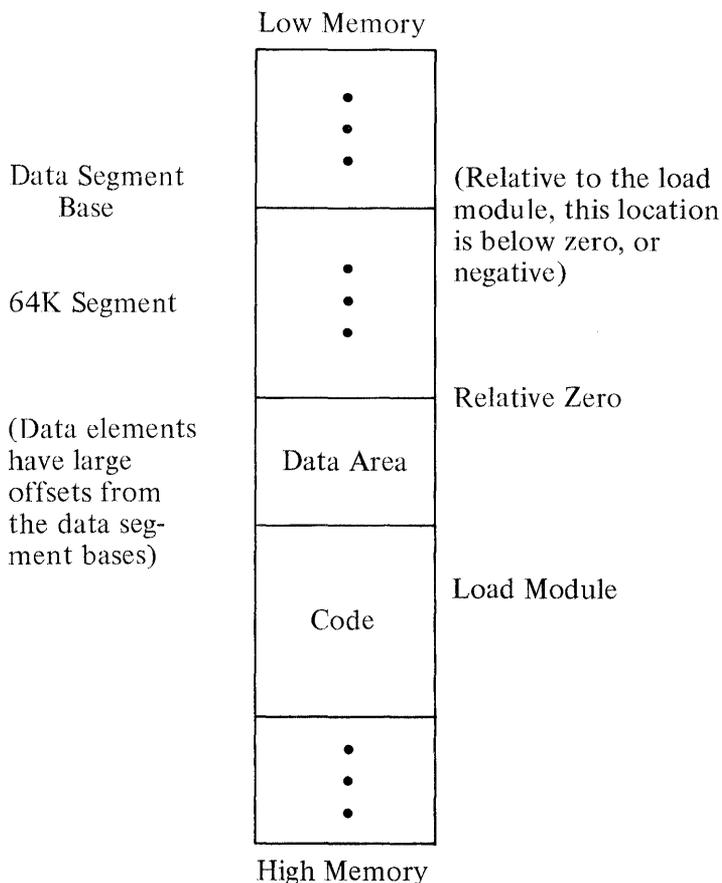
How to Determine the Absolute
Address of a Segment

The linker map displays a list of segments in the
relative order of their appearance within the load
module. The information displayed shows a 20-bit
hex address of each segment relative to location zero.
The addresses that are displayed are not the absolute
addresses of where these segments are actually located.
To determine where relative zero is actually located,
we must use DEBUG. DEBUG is described in detail
in the *IBM Personal Computer DOS* manual.

Using DEBUG,

1. Load the application.

    Note the segment value in CS and the offset
    within that segment to the entry point as shown
    in IP. The last line of the linker map also describes
    this entry point, but uses relative values, not the
    absolute values shown by CS and IP.

2. Subtract the relative entry as shown at the end of
    the map listing from the CS:IP value. For example,
    let's say CS is at 05DC and IP is at zero.

    The linker map shows the entry point at 0100:0000.
    (0100 is a segment ID or paragraph number; 0000
    is the offset into that segment.)

    In this example, relative zero is located at
    04DC:0000, which is 04DC0 absolute.

If a program is loaded low, the relative zero location is
located at the end of the Program Segment Prefix, in
the location DS plus 100H.

LINK

Messages

All messages, except for the warning messages, cause the LINK session to end. Therefore, after you locate and correct a problem, you must rerun LINK.

Messages appear both in the list file and on the display unless you direct the list file to CON, in which case the display messages are suppressed.

A complete list of linker messages follows:

About to generate .EXE file
Change disks <hit ENTER>
This message is displayed when you specify the /PAUSE parameter. Insert your Runfile diskette into the appropriate drive and press Enter.

Ambiguous switch: *z*
The characters specified by *z* do not uniquely identify a linker parameter. Use more characters from the parameter name.

An internal failure has occurred
An error has occurred in the linker program. Report the conditions under which the message appeared to your computer dealer.

Attempt to access data outside of segment bounds
The object module is probably bad.

Bad numeric parameter
The value you specified with the **/STACK** parameter is not a valid numeric constant.

Cannot find file *filespec*
Change diskette <hit ENTER>
The linker could not locate the specified object module on the drive. Insert the diskette with the specified module on it and press Enter.

Cannot find library *libname*
Enter new drive letter:
The specified library could not be found on the drive.
Enter the letter for the drive the library is on.

Cannot nest response file
You used *@filespec* within an automatic response file.
Automatic response files cannot be nested.

Cannot open list file

Cannot open overlay
The preceding two messages occur because the
directory is full.

Cannot open response file
The automatic response file could not be found.

Cannot open temporary file
The directory is full.

DUP record too complex
A problem exists in an object module created from
an assembler source program. A single DUP
requires 1024 bytes before expansion.

Fixup offset exceeds field width
A machine language processor instruction refers to an
address with a NEAR attribute instead of a FAR
attribute.

Invalid format file
A library is in error.

Invalid object module
An object module is incorrectly formed or incomplete
(as when the language processor is stopped in the middle).

Invalid switch: *z*
The characters indicated by *z* do not form a valid
linker parameter.

LINK

No object modules specified
You did not name any object modules in the command
line or in response to the prompt. The linker needs
some files to link.

Out of space on list file

Out of space on run file

Out of space on VM.TMP
The preceding three messages say that no more diskette
space remains to expand the indicated file. You should
retry the operation with a different diskette.

Program size exceeds capacity of linker
The load module is too big for processing.

Segment size exceeds 64K
In an attempt to combine identically named segments,
the resulting segment required more than 64K.
64K-bytes is the addressing limit.

Stack size exceeds 65535 bytes
The size specified for the stack must be less than or
equal to 65535.

Symbol defined more than once
The linker found two or more modules that define
a single symbol name. This message is only a warning.

Symbol table capacity exceeded
The limit is about 30K. Use shorter and/or fewer
names.

There was/were *number* errors detected
This message is displayed for your information at the
end of the link session.

Too many external symbols in one module
The limit is 256 external symbols per module.

Too many groups
The limit is 10, including DGROUP.

Too many libraries specified
The limit is 8 libraries.

Too many overlays
The limit is 64.

Too many public symbols in one module
The limit is 1024 public symbols.

Too many segments or classes
The limit is 256 (segments and classes taken together).

Unexpected end-of-file on library
This is probably caused by an error in the file.

Unexpected end-of-file on VM.TMP
The diskette containing VM.TMP has been removed.

Unresolved externals:
This message is followed by a list of the unresolved
external references. If this error occurs, do not
attempt to run the executable file created by the linker.

VM.TMP is an illegal file name and has been ignored
VM.TMP cannot be used for an object filename.
This message is only a warning.

Warning: No Stack Statement
When combining object modules from the Macro
Assembler, at least one source file must have a
SEGMENT command with the combine type of
STACK (compilers automatically provide stack
allocation statements).

Appendix C. Memory Maps

Segment Map

The segment maps for compiled programs under BASIC is almost the same for versions with and without the runtime module.

Address	Non Runtime Module Segment	Class	Runtime Module Segment	Class
low CS	BC_CODE	CODE	BC_CODE	CODE
	CODE	CODE	CODE	CODE
	BC_ICN	INIT	BC_ICN	INIT
	BC_IDS	INIT	BC_IDS	INIT
	INIT	INIT	INIT	INIT
low DS	CONST	RT_DATA	CONST	RT_DATA
	DATA	RT_DATA	DATA	RT_DATA
	COMMON	BLANK	COMMON	BLANK
	CONST	CONST	CONST	CONST
	DATA	DATA	DATA	DATA
	BC_DATA	DATA	BC_DATA	DATA
	BC_FT	DATA	BC_FT	DATA
	BC_CN	DATA	BC_CN	DATA
	BC_DS	DATA	BC_DS	DATA
	RUN	DATA (optional)	RUN	DATA
high DS	STACK	STACK	STACK	STACK
highest memory			BASRUN.EXE Runtime Module Code	

The segments BC_ICN and BC_IDS are block transferred to the segments BC_CN and BC_DS at program initialization. Just before the user program itself executes, the DS segment is moved down in physical memory over the segments of class INIT. If the runtime module is used, then the data segment is moved to high memory under the runtime module.

The contents of the segments are as follows:

BC_CODE	Compiled user program
CODE	BASIC and assembler runtime routines
BC_ICN	User program constants (moved to BC_CN)
BC_IDS	User program data statements (moved to BC_DS)
INIT	Disposable runtime initialization code
CONST	Runtime initialized data values
DATA	Runtime uninitialized data values
COMMON	User program COMMON area
CONST	User initialized data variables (assembly lang.)
DATA	User data variables (assembly lang.)
BC_DATA	User program data variables
BC_FT	User program floating point temporaries
BC_CN	Program constants
BC_DS	User program data statements
RUN	Relocatable data segment used by RUN statement
STACK	Stack segment required by loader (not used)

The string space and stack space are set up at
initialization time. All the available space after
MEMORY is used for the string space (up the 64K total
for DS segment) except for 512 bytes reserved for the
stack. In general, while a BASCOM program is running
the segment registers DS, ES, and SS are the same. CS
varies depending on whether the program or runtime
code is executing.

Memory Map (with Runtime Module)

low memory

Interrupt vectors	◄── 0000:0000
DOS area	
BC_CODE (compiled program)	◄── Bottom of user area compiled BASIC code (64K maximum)
CODE	◄── additional code (no size restriction)

free memory ◄── (if available)

DATA CONST (runtime – RT_DATA)	◄── DS,ES,SS:0000 runtime module data and constants (fixed size – approx. 3K)
COMMON	◄── COMMON area (variable per program) ◄──
DATA CONST (assembly language)	Optional data area for assembly language subroutines
BC_DATA (user variables)	◄── BASIC program variables
BC_DS BC_CN BC_FT	◄── Data statements Numeric and string constants Floating point temporaries
string space	◄── Bottom of string space
file buffers	◄── (dynamic boundary)
user stack	◄── Top of string space (default–512 bytes) ◄── Top of DS,ES,SS (64K maximum)
BASRUN.EXE	
space for .EXE loader	◄── Top of user area

high memory

Memory Map (without Runtime Module)

low memory

Interrupt vectors	← 0000:0000
DOS area	
BC_CODE (compiled program)	← Bottom of user area compiled BASIC code (64K maximum)
CODE	← additional code (no size restrictions)

free memory   (if available)

DATA CONST (runtime — RT_DATA)	← DS,ES,SS:0000 runtime module data and constants (fixed size — approx. 3K)
COMMON	← COMMON area (variable per program)
DATA CONST (assembly language)	← Optional data area for assembly language subroutines
BC_DATA (user variables)	← BASIC program variables
BC_DS BC_CN BC_FT	← Data statements Numeric and string constants Floating point temporaries
RUN (opt.)	← .EXE file loader data (64 bytes)
string space	← Bottom of string space ← (dynamic boundary)
file buffers	
user stack	← Top of string space (default — 512 bytes) ← Top of DS,ES,SS (64K maximum)
space for .EXE loader	← Top of memory

high memory

GLOSSARY

This part of the book explains many of the technical terms you may run across while programming in BASIC.

application program: A program written by or for a user which applies to the user's work. For example, a payroll application program.

argument: A value that is passed from a calling program to a function.

arithmetic overflow: Same as overflow.

ASCII: American National Standard Code for Information Interchange. An ASCII file is a text file that uses ASCII codes to represent each character.

backup: Pertaining to a system, device, file, or facility that can be used in case of a malfunction or loss of data.

binary: Pertaining to a condition that has two possible values or states. Also, refers to the Base 2 numbering system.

buffer: An area of storage which is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read or from which data is written.

bug: An error in a program.

byte: The representation of a character in binary. Eight bits.

call: To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

code: To represent data or a computer program in a symbolic form that can be accepted by a computer; to write a routine. Also, loosely, one or more computer programs, or part of a program.

comment: A statement used to document a program. Comments include information that may be helpful in running the program or reviewing the output listing.

compile: To translate a computer program written in problem-oriented language into machine-oriented language.

compiletime: That period of time during which the compiler is executing, during which it compiles a BASIC source file and creates an object file.

debug: To find and eliminate mistakes in a program.

default: A value or option that is assumed when none is specified.

delimiter: A character that groups or separates words or values in a line of input.

directive: A compiler metacommand.

directory: A table of identifiers and references to the corresponding items of data. For example, the directory for a diskette contains the names of files on the diskette (identifiers), along with information that tells DOS where to find the file on the diskette.

disabled: A state that prevents the occurrence of certain types of interruptions.

DOS: Disk Operating System. In this book, refers only to the IBM Personal Computer Disk Operating System.

dummy: Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.

dynamic: Occurring at the time of execution.

echo: To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit: To enter, modify, or delete data.

enabled: A state of the processing unit that allows certain types of interruptions.

end of file (EOF): A "marker" immediately following the last record of a file, signaling the end of that file.

event: An occurrence or happening; in IBM Personal Computer Advanced BASIC, refers particularly to the events tested by the COM(n), KEY(n), PEN, and STRIG(n).

execute: To perform an instruction or a computer program.

external reference: A variable name or label in one module that is referenced in another module. External labels are the entry points into modules.

folding: A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.

format: The particular arrangement or layout of data on a data medium, such as the screen or a diskette.

form feed (FF): A character that causes the print or display position to move to the next page.

function: A procedure which returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing, or its characteristic action.

garbage collection: Synonym for housecleaning.

global reference: Same as external reference.

hard copy: A printed copy of machine output in a visually readable form.

housecleaning: When BASIC compresses string space by collecting all of its useful data and frees up unused areas of memory that were once used for strings.

instruction: In a programming language, any meaningful expression that specifies one operation and its operands, if any.

integer: One of the numbers $0, \pm1, \pm2, \pm3, \ldots$

integrity: Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.

interface: A shared boundary.

interpret: To translate and execute each source language statement of a computer program before translating and executing the next statement.

interrupt: To stop a process in such a way that it can be resumed.

invoke: To activate a procedure at one of its entry points.

K: When referring to memory capacity, two to the tenth power or 1024 in decimal notation.

keyword: One of the predefined words of a programming language; a reserved word.

library: A set of routines in a file on diskette.

linking: The process by which the linker program combines separate modules, resolves all external references by searching the appropriate library, and creates a single executable (.EXE) file on diskette.

linktime: That period of time during which the linker is running.

location: Any place in which data or machine instructions may be stored.

loop: A set of instructions that may be executed repeatedly while a certain condition is true.

M: Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.

metacommand: A statement that supplies information to the compiler but which usually does not directly result in executable code. For the BASIC Compiler, some of the metacommands are $INCLUDE, $LINESIZE, $PAGE, $PAGESIZE, and $TITLE.

minifloppy: A 5-1/4 inch diskette.

mnemonic: A symbol chosen to assist the human memory; for example, an abbreviation such as "MPY" meaning "multiply."

module: A fundamental unit of code. There are several types of modules, including object and executable modules. The compiler creates object modules that are later manipulated by the linker. Your final executable program is an executable module.

nest: To incorporate a structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.

notation: A set of symbols, and the rules for their use, for the representation of data.

null: Empty, having no meaning. In particular, a string with no characters in it.

object file: Output from a compiler which is itself executable machine code or is suitable for processing to produce executable machine code.

offset: The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.

on-condition: An occurrence that could cause a program interruption. It may be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.

operating system: Software that controls the execution of programs; often used to refer to DOS.

operation: A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.

overflow: When the result of an operation exceeds the capacity of the intended unit of storage.

parameter: A name in a procedure that is used to refer to an argument passed to that procedure.

prompt: A question the computer asks when it needs you to supply information.

range: The set of values that a quantity or function may take.

real-time: Pertaining to the manipulation of data that are required or generated by some process while that process is in operation; usually the results are used to influence the process while it is occurring.

record: A collection of related information, treated as a unit. For example, in stock control, each invoice might be one record.

recursive: Pertaining to a process in which each step makes use of the results of earlier steps, such as when a function calls itself.

relocatable: A module is relocatable if the code within it can be "relocated" and run at different locations in memory. Relocatable modules contain labels and variables represented as offsets relative to the start of the module. These labels and variables are said to be "code relative." During the linking step, the linker associates an address with the start of the module, and then computes an absolute address that is equal to the associated address plus the code relative offset for each label or variable. These new computed values become the absolute addresses that are used in the executable file.

routine: Executable code residing in a module. More than one routine may reside in a module.

runtime: That period of time during which a previously compiled and linked program is executing. By convention, runtime refers to the execution time of your program and *not* to the execution time of the compiler or the linker.

runtime module: BASRUN.EXE – a module containing most of the routines needed to implement the BASIC language. The runtime module contains a majority of the library routines needed to implement the BASIC language. A library routine usually corresponds to a feature or sub-feature of the BASIC language.

It is a peculiarity of the runtime module that it is an executable .EXE file, and must be accessible on disk when you execute your final .EXE file.

runtime support: The body of routines that may be linked to your compiled object file. These routines implement various features of the BASIC language. The BASCOM.LIB and BASRUN.LIB libraries, along with the runtime module, all contain runtime support routines.

source: A BASIC program in ASCII format. Usually this term is used to refer to the program which is input to the compiler. The compiler translates this source and creates as output a new file called an "object" file.

stack: A method of temporarily storing data so that the last item stored is the first item to be processed.

statement: A meaningful expression that may describe or specify operations and is complete in the context of the BASIC programming language.

syntax: The rules governing the structure of a language.

trap: A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

unbound global reference: Same as unresolved external reference.

unresolved external reference: An external reference in a module that is not declared in that module. The linker tries to "resolve" this situation by searching for the declaration of that reference in other modules. These other modules are usually library modules. If the variable or label is found, the address associated with it is substituted for the reference in the first module, and is then said to be "resolved." When a variable is not found, it is said to be "undefined."

External references that are not resolved as part of the linking process can cause unpredictable results when you run your program.

INDEX

Special Characters

A

B

X

4

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC Compiler

6172246

Your comments assist us in improving our products.
IBM may use and distribute any of the information
you supply in anyway it believes appropriate without
incurring any obligation whatever. You may, of
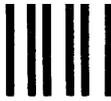course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in
this space.

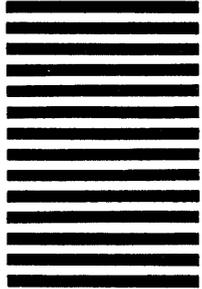Name _____

Address _____

City_____ State _____

Zip Code _____

BUSINESS REPLY MAIL

FIRST CLASS   PERMIT NO. 123   BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

· ·
Fold here

Please do not staple   Tape

IBM

The Personal Computer
Software Library

Product Comment Form

BASIC Compiler

6172246

Your comments assist us in improving our products.
IBM may use and distribute any of the information
you supply in anyway it believes appropriate without
incurring any obligation whatever. You may, of
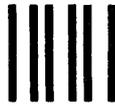course, continue to use the information you supply.

Comments:

If you wish a reply, provide your name and address in
this space.

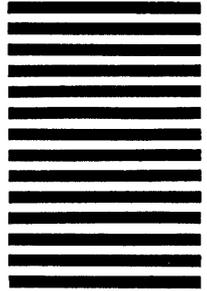Name _____

Address _____

City_____ State _____

Zip Code _____

BUSINESS REPLY MAIL

FIRST CLASS   PERMIT NO. 123   BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

· ·

Fold here

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or

2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBLITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328-W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

IBM ®

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

6172246