Compiling HP BASIC 6.2 Programs

HP Part No. 98618-90001
Printed in USA

Notice

The information contained in this document is subject to change without notice.

Hewlett-Packard Company (HP) shall not be liable for any errors contained in this document. HP MAKES NO WARRANTIES OF ANY KIND WITH REGARD TO THIS DOCUMENT, WHETHER EXPRESS OR IMPLIED. HP SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. HP shall not be liable for any direct, indirect, special, incidental, or consequential damages, whether based on contract, tort, or any other legal theory, in connection with the furnishing of this document or the use of the information in this document.

Warranty Information

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

Restricted Rights Legend

Use, duplication or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013.

Use of this manual and magnetic media supplied for this product are restricted. Additional copies of the software can be made for security and backup purposes only. Resale of the software in its present form or with alterations is expressly prohibited.

Printing History

First Edition - April 1990
Second Edition - June 1991

Contents

1

Using This Manual

When you are using a new software product, one of the most important elements of success is the documentation. Being without a complete and readable user's manual is like being in a strange city without a map. Before you start using the BASIC Compiler, you should read this chapter and familiarize yourself with the organization and content of the manual, as well as the notation and terms that are used. Taking time to do this will, in the long run, save you some effort.

Manual Notation

This manual will utilize the following notational conventions:

`Computer Font`	is used in all examples.
Italics Font	is used to identify generic descriptions of items to be supplied by you.
Bold Font	is used to identify important terms which may be found in the Glossary (see appendix C). Each of these terms will be in bold font the first time you see them in this manual.
[ ]	are used to denote optional items.
( )	are used to specify parameters or arguments which are not optional and must be supplied by you.
/	is used to separate two items and means that one or the other (but not both) secondary keywords may be used.
UPPER CASE LETTERS	are used to denote BASIC keywords and secondary BASIC keywords which must appear in the *same form* as they appear in the manual.

Key caps are shown in appropriate key envelopes (e.g. [A]).

The following is an example of notational conventions:

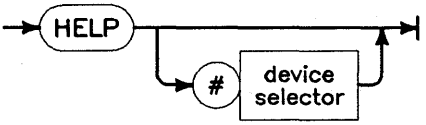
```
10  ! $CONFIGCHECK [ON/OFF] [$ [comments]]
```

where:

`10 ! $CONFIGCHECK`	is in computer font because it is an example.
`ON/OFF`	are two secondary keywords separated by a "/". The "/" tells you that one or the other but not both secondary keywords may be used.
`[ON/OFF]`	are optional as indicated by the square brackets.
comments	is in italics to indicate that the text is to be supplied by you.

Syntax Drawings Explained

Statement and directive syntax are represented pictorially as shown in the following example. All characters enclosed by a rounded envelope must be entered exactly as shown. Words enclosed by a rectangular box are names of items used in the statement. A description of each item is given either in the table following the drawing, in another drawing, or in the Glossary.



Statement elements are connected by lines. Each line can be followed in only one direction as indicated by the arrow at the end of the line. Any combination of statement elements that can be generated by following the lines in the proper direction is syntactically correct. An element is optional if there is a path around it. Optional items usually have default values. The table or text

following the drawing specifies the default value that is used when an optional item is not included in a statement.

Comments may be added to any valid line. A comment is created by placing an exclamation point after a statement or after a line number or line label. For example:

```
100   PRINT "Hello"  ! This is a comment
110   ! This is also a comment.
```

The text following the exclamation point may contain any characters in any order.

Comments for directives are created by placing a dollar sign after the directive statement. For example:

```
10   ! $KEEP ON $ This is a comment.
```

The text following the second dollar sign may contain any characters in any order.

The drawings do not necessarily deal with the proper use of spaces (ASCII blanks). In general, whenever you are traversing a line, any number of spaces may be entered. If two envelopes are touching, it indicates that no spaces are allowed between the two items. However, this convention is not always possible in drawings with optional paths, so it is important to understand the rules for spacing given in the next section.

Keywords and Spaces

The computer uses spaces, as well as required punctuation, to distinguish the boundaries between various keywords, names, and other items. In general, at least one space is required between a keyword and a name if they are not separated by other punctuation. Spaces cannot be placed in the middle of keywords or other reserved groupings of symbols. Also, keywords are recognized whether they are typed in uppercase or lowercase. Therefore, to use the letters of a keyword as a name, the name entered must contain some mixture of upper-case and lower-case letters. The following are some examples of these guidelines.

Space Between Keywords and Names

The keyword NEXT and the variable `Count` are properly entered with a space between them, as in `NEXT Count`. Without the space, the entire group of characters is interpreted as the name `Nextcount`.

No Spaces in Keywords or Reserved Groupings

The keyword DELSUB cannot be entered as `DEL SUB`. The array specifier `(*)` cannot be entered as `( * )`. A function call to "A$" must be entered as `FNA$`, not `FN A $`. The I/O path name "@Meter" must be entered as `@Meter`, not as `@ Meter`. The "exceptions" are keywords that contain spaces, such as `END IF` and `OPTION BASE`.

Using Keyword Letters for a Name

Attempting to store the line `IF X=1 THEN END` will generate an error because `END` is a keyword not allowed in an `IF ... THEN` statement. To create a line label called "End," type `IF X=1 THEN ENd`. This or any other mixture of uppercase and lowercase will prevent the name from being recognized as a keyword.

Preview of the Chapters and Appendixes

Below is a summary of the organization of the material in this manual.

Chapter 1: Using This Manual

Unless you skipped right to this section, you know what this chapter is all about.

Chapter 2: The BASIC Compiler

This chapter is devoted to giving you an idea of what the compiler does, how it works, and what its limitations are.

Chapter 3: Overview of Compiler Tasks

This chapter is a quick reference and index into tasks described in the manual. It lists a number of common tasks that you may want to perform, gives a quick explanation of how to accomplish them, and tells you where in the manual to look for more detailed information on the tasks.

Chapter 4: Details of Using Your Compiler

This chapter gives you detailed instructions on using the compiler. It includes the methods for installing the compiler binary (BASIC/WS and BASIC/DOS only), and invoking the compiler and the options that are available to you.

Chapter 5: Details of Compiler Directives

This chapter lists and describes the compiler directives that are available to you.

Chapter 6: Improving Compiled Programs

This chapter provides information about compiled programs and offers tips on how to write programs that compile and run efficiently. Detailed instructions on using the EOL directive are also found in this chapter.

Appendix A: Error Messages and Warnings

This appendix describes error messages and warnings created by the compiler.

Appendix B: Troubleshooting

This appendix provides help with common problems encountered when using the compiler.

Appendix C: Glossary

This appendix contains a list of important terms found in this manual.

2

The BASIC Compiler

This chapter is devoted to giving you an idea of:

- what the compiler does
- how it works
- what its limitations are

If you are using HP BASIC/UX 6.2, the BASIC/UX compiler is included and installed with BASIC/UX. If you are using HP BASIC/WS 6.2 or HP BASIC/DOS 6.2, the HP 98618A BASIC Compiler must be installed to allow you to compile your BASIC programs.

Hardware and Software Requirements

The BASIC Compiler can be run on any HP Series 200 or 300 computer running the HP BASIC Language System of the appropriate version (e.g., 6.2 for the 6.2 compiler). If you are using BASIC/WS, you will also need a local disk drive to install the compiler binary or to transfer the compiler binary from the disk media to another storage location. A disk drive is not needed during the compilation process.

A different compiler is required for each new version of the BASIC Language System. For example, if you have compiled programs on a 1.0, then they will run on all 1.x revisions (such as 1.0, 1.1, etc.): a different compiler would be required to run such programs on revisions 2.x or 3.x.

The compiler compiles a program that is residing in memory. Since it needs room to hold both the source code and the compiled code, it may require available memory up to 3 times the size of the program to be compiled. Some programs require much less.

2 Compiler Capabilities

The BASIC Compiler provides you with the ability to translate a BASIC program into a series of Compiled Subprograms (**CSUBs**). Each CSUB is a machine code routine (written in the language of the MC68000/68010/68020 microprocessor) which will perform the same task as the original BASIC **subprogram.** BASIC SUBs are compiled into CSUBs, and BASIC **functions** are compiled into **CDEFs** (see the Glossary). The advantages of compiled BASIC include:

- Speed—because the compiled program needs fewer steps to complete the same task as an interpreted BASIC program, a compiled program runs faster.
- Security—as each CSUB is written in machine code, it is almost impossible to recover the original BASIC source code (the BASIC statements which make up your program).

Identifying the Compiler Revisions

The BASIC compiler is associated with a software revision number. The revision number will change each time you get a software update. To determine the revision number using BASIC, type:

```
SYSTEM$("VERSION:COMPILER")
```
Return

This returns a string containing the revision number. The number is of the form:

```
R.x
```

where `R` is the BASIC Language System revision (6, etc.) and `x` is the release number of the BASIC Compiler.

Compiler Overview

The BASIC compiler will compile from and to RAM. The program to be compiled must reside in RAM and the results of compilation will be resident in RAM. If you are using BASIC/WS or BASIC/DOS the BASIC Compiler is a BASIC binary that is loaded into the system via the LOAD BIN command. After a program has been compiled the COMPILER binary need not reside in memory.

The compiler can be invoked with a **compiler invocation command.** These commands will cause the compiler to begin compiling a BASIC program which is residing in memory. The simplest compiler invocation command is COMPILE. The following describes what occurs when the COMPILE command is executed.

With the compiler installed, you are ready to compile a program. Either develop a new program to be compiled or LOAD a program from an external location to be compiled. Let's suppose that your program to be compiled looks like this:

```
10   COM REAL D, INTEGER A
20   PRINT "NOW IN MAIN PROGRAM"
30   CALL Subone
40   CALL Subtwo
50   A=1
60   D=5.4
70   END
80   SUB Subone
90      PRINT "NOW IN SUB Subone"
100  SUBEND
110  SUB Subtwo
120     PRINT "NOW IN SUB Subtwo"
130  SUBEND
```

With this program in memory, you would enter the compiler invocation command:

```
COMPILE (Return)
```

If you EDIT this program, you will see:

```
1        OPTION BASE 0
3        COM REAL D, INTEGER A
4        Mainsub
5        END
6 CSUB Mainsub
7        SUB Mainsub
8        COM REAL D, INTEGER A
9        PRINT "NOW IN MAIN PROGRAM"
10       CALL Subone
11       CALL Subtwo
12       A=1
13       D=5.4
14       SUBEND
15 CSUB Subone
16       SUB Subone
17          PRINT "NOW IN SUB Subone"
18       SUBEND
19 CSUB Subtwo
20       SUB Subtwo
21          PRINT "NOW IN SUB Subtwo"
22       SUBEND
```

You should notice a few things about this program. First, the compiler renumbers your program as it compiles it. So, no matter how you number your program, it will be numbered starting with line 1 after it is compiled. Note that the line number sequence may be broken by one line number because the compiler reserved that line during compilation and later decided not to use it. Therefore, the line sequence in the above program is broken at line 2. This does not mean that there is a program line missing. Second, the original BASIC source code is still there. By default, the compiler will keep both the source code and the compiled code in memory. The interpreted parts of your program will be indented further than the compiled lines so that they are easily distinguishable.

When you want to RUN your program, simply press the RUN key (f3 in the System menu on the ITF Keyboard). When the system encounters the call to "Mainsub," it will execute the first "Mainsub" it finds, which will be the compiled version (CSUB Mainsub). In the above program, only the compiled subprograms will be executed when the RUN key is pressed.

The Compilation Process

The compiler binary first pre-runs your program to cause the system to build the necessary tables. It will then begin to compile your program. Each subprogram in your program will be transformed into a CSUB (Compiled subprogram). If your **main program** consists of an END statement, it will be left as is. If, however, your main program is more than an END statement, it will be transformed into a subprogram called "Mainsub." This subprogram created by the compiler will then be compiled into a CSUB of the same name. Each function will be transformed into a CDEF (Compiled function). Each CSUB or CDEF is written in machine code.

The BASIC compiler will keep the source code of the original subprograms and functions, along with the compiled CSUBs and CDEFs. After compilation, both the compiled code and the source code will reside in memory. This enables you to modify your original program if you wish. Each subprogram and function will have its associated CSUB or CDEF inserted immediately before it. The name of each CSUB/CDEF will be the name of the subprogram or function from which it was created. This will not cause any problems when the program is run, since the BASIC Language System always executes the first subprogram it finds when a name is encountered more than once. Hence, the compiled version (the CSUB or CDEF) of each routine will be executed when the program is run.

You may choose not to keep the source code either by deleting it after the program is compiled or by using the KEEP OFF compiler switch or directive (see the section "Switches in the Options List" and the chapter "Details of Compiler Directives"). However, once the source code has been removed you can no longer use the UNCOMPILE command to remove the compiled code. The UNCOMPILE command and KEEP OFF directive are covered in subsequent chapters. To invoke the compiler using the KEEP OFF directive, type:

```
COMPILE: KEEP OFF
```

The source code will be deleted from memory as the compiled code is produced. If you want to delete the source code after the program has been compiled, you can use the DEL command. You *cannot* delete the source code with the DELSUB command, because it deletes the compiled code.

Since the main context of your program is compiled into a CSUB, your main program "disappears" during the compilation process. To adjust for this, the Compiler will create a **dummy main program** (see the section in this chapter titled "A Dummy MAIN Program and Mainsub"). This dummy main program will consist of a call to Mainsub (the CSUB created from your original main program) and an END statement. If your program contains any COM statements, the dummy main program will also contain a duplicate of each COM statement in your program, as well as the appropriate OPTION BASE statement.

CSUBs and CDEFs generated by the BASIC compiler can be loaded at run time from any BASIC program (interpreted or compiled) by using the LOADSUB command. If you wish to LOADSUB a subprogram from a compiled program that you have previously stored, and that program consists of both compiled code and source code (i.e., the source code was not deleted), then the command:

```
LOADSUB subprogram FROM "program_name"
```

will always bring in the CSUB of the *subprogram* specified, since the CSUB occurs before the *subprogram.*

A Dummy MAIN Program and Mainsub

If your main program contains more than an END statement, at compile time the compiler converts your main program into a subprogram called "Mainsub" and then compiles it into a CSUB called "Mainsub." Since this means that you no longer have a main program, the compiler creates a dummy main program for you. The dummy MAIN program created by the compiler contains all COM declarations from your original main program, a call to "Mainsub," and an END statement (see the previous section titled "Compiler Overview"). Note that you can EDIT and change this dummy MAIN program if you wish but *do not* re-enter the END statement if you wish to keep this dummy MAIN program in interpreted form.

When the dummy MAIN is created, the END statement is "marked" by the compiler to prevent the compiler from re-compiling the dummy MAIN in subsequent compilations. The compiler scans your program while it is

compiling to find out if any subprograms with the name "Mainsub" already exist. If it finds that a subprogram with the name "Mainsub" exists, the compiler issues a warning (warning 22). If you choose to ignore this warning you may end up with an infinitely recursive program, or your program may display a totally different behavior from what you were expecting.

If you do not want the compiler to compile your main program, you should invoke the compiler via the command:

```
COMPILE SUBX TO END
```

where SUBX is the subprogram occurring after the main program.

If, for some reason, you wish to compile your main program, then you can force it to be compiled by one of two methods:

- Invoke the compiler with the command:

  ```
  COMPILE MAIN (Return)
  ```

- Re-enter the END statement. It is the END statement that is marked by the compiler to prevent subsequent regeneration of the dummy MAIN program. Re-entering the END statement erases this mark.

Compiler Limitations

The BASIC Compiler compiles programs written in Hewlett Packard's Series 200/300 BASIC Language. However, a few statements, commands, and keys in BASIC are not supported since they are not applicable to a compiled environment. Note that they should not affect the execution of compiled BASIC programs. If your program contains one or more of these statements or commands, you may remove them or put them in an interpreted section of your program (i.e., a section of your program which is not compiled). The following is a list of compiler limitations as they relate to certain statements, commands and keys.

- Programmable BASIC commands *are not* supported by the compiler. These commands are as follows:

  ```
  GET       RE-SAVE       SAVE
  ```

2

Note that a GET statement which is followed by a file name is allowed in a compiled program. However, a GET statement followed by a file name and a line number is not permitted.

- Non-programmable BASIC commands that cannot be used to access, modify, or locate the compiled part of a BASIC program (CSUBs) are given below:

```
CHANGE      COPYLINES     INDENT      REN
CONT        DEL           FIND        MOVELINES
```

- TRACE PAUSE does not work in compiled programs.
- The [STEP] key ([f1] key on the ITF Keyboard System menu) cannot be used to single step into a CSUB. Using the [STEP] key will have the same effect as pressing the [CONTINUE] key ([f2] key on the ITF Keyboard System menu).
- Caution must be exercised when performing DELSUB, LOADBIN, or INITIALIZE of a memory volume, whether from within a program or from the keyboard. In other words, *do not* cause any active CSUB to physically be moved at run time. Once a CSUB is activated it cannot be relocated in memory because this can cause the machine to hang or a run-time error 133 to occur. You should keep the following guidelines in mind:

  Do not DELSUB the CSUB you are currently executing;

  Do not DELSUB any CSUBs, SUBs, or DEFs which are located physically before any active CSUB. Attempting to delete a CSUB (at run time) which was physically loaded before the CSUB containing the DELSUB statement will generate run-time error 133 (DELSUB of non-existent or busy sub (or prior sub)).

  Do not INITIALIZE a RAM volume or LOADBIN any binary (for BASIC/WS or BASIC/DOS) while any CSUB is still active. An active CSUB is a CSUB which was called and is currently executing or which made a CALL to another subprogram and is waiting for a RETURN, SUBEXIT, or SUBEND to be executed.

- You cannot single-step a compiled program. However, you can use the [PAUSE] key ([Stop] key on the ITF Keyboard) and the PAUSE statement to pause the program while it is executing. The [PAUSE] key will be processed quicker if the program was compiled with EOL ON and an EOL is encountered.

- When using the BASIC function ERRL in a compiled program, the value returned by ERRL will always be zero (0).
- If you have a program that loads parts of another program, such as:

```
100   LOAD "P1", Lbl1
            .
            .
150   LOAD "P1", Lbl2
            .
            .
200   LOAD "P1", Lbl3
            .
            .
```

it will *not* work if the program to be loaded (P1 in this case) is compiled. This is because the labels are moved to the compiled subprogram "Mainsub" during compilation. A run-line label on a LOAD statement is valid only if the run-line label is in the main program (The above program segment would work fine if P1 were interpreted). As a solution for this problem, you could have the calling program structured like this:

```
100   COM /Loadloc/ Loc
110   Loc=1
120   LOAD "P1"
            .
            .
180   Loc=2
190   LOAD "P1"
            .
            .
240   Loc=3
250   LOAD "P1"
            .
            .
```

and the referenced program (P1 in this case) structured as:

```
100   COM /Loadloc/ Loc
            .
            .
150   ON Loc GOTO  Lbl1,Lbl2,Lbl3
            .
            .
```

- It is very important that the BASIC Compiler know the types of functions used during compilation, especially when a function is part of an expression, as in:

```
A = X + FNSample(Y) - Z
```

  where "FNSample" is a function in your program. The compiler directives REAL and COMPLEX determine which data type a function must return. The COMPLEX directive must be used to declare functions that are to return a COMPLEX value. REAL functions need not be declared this way (since REAL is the default), though your program may be more consistent if all of your functions are declared. These directives are explained in detail in the chapter "Details of Compiler Directives."

- When using a list of subprograms with the COMPILE command, the program is limited to only 2048 subprograms. This limitation does not exist if the COMPILE command is used without a list of subprograms when compiling the program.
- The LOADSUB FROM command will not load subprograms referenced in compiled subprograms (unless the interpreted version of the same subprogram is in memory).
- A GOTO will not have an EOL surrounding it even if EOL is ON (see the section "Toggling EOL" found in the chapter "Improving Compiled Programs").
- Caution must be exercised when compiling large programs with line increments of 1, especially when your last line is greater than 32700. The reason for this is that the compiler inserts lines to create a dummy MAIN program at the start and inserts a new line for each CSUB it creates. Therefore, it is likely that the maximum line number may be exceeded. To solve this problem, break your program up into several subprograms and compile and load them separately. If you are unsure whether this problem exists with your program, then it is a good idea to store your program before compiling it.

3

Overview of Compiler Tasks

This chapter acts as a quick reference guide and *is not* intended as the *only* reading material for first-time users. First-time users are advised to read the entire manual before using the BASIC Compiler. Depending on this chapter as your main source of information will cause you to miss important information. This chapter:

- lists common tasks that you may want to perform.
- gives a quick explanation of how to accomplish each task.
- tells you where in the manual to look for more detailed information related to the task.

Chapter Contents

This table of contents will help you find the task you want to perform:

3

How Tasks Are Described

The material in this section should be considered as an expanded index only. Each "how to" task is described in the following manner:

Task

Considerations

Lists some things that need to be considered before the task is done (when applicable).

Solution

Provides a quick "how to" for the listed task.

Reference

Refers you to the appropriate sections of the manual for more information.

Task Reference

There are 12 tasks covered in this section for BASIC/UX users, and 14 tasks for BASIC/WS users. Each task contains a solution and a reference section and in some cases a considerations section, as explained at the beginning of this chapter.

Install the Compiler (BASIC/WS and BASIC/DOS only)

(BASIC/UX includes the Compiler and does not require separate installation.)

Solution

3

Insert the disk containing the compiler into a disk drive connected to your computer, and type:

```
LOAD BIN "COMPILER [:msvs]" Return
```

where *msvs* is the mass storage volume specifier.

Reference

See the section "Language Extensions, Drivers, and Configuration" found in the *Installing and Maintaining HP BASIC* manual, chapter 5.

Compile a Simple Program

Solution

LOAD or GET the program to be compiled into memory (or develop a new program in memory). Then execute:

```
COMPILE (Return)
```

Reference

Chapter/Section Title	Page
Invoking the Compiler	4-3
COMPILE Command	4-3

Access On-line Help

Solution

Enter one of the following commands:

3

`HELP`	prints a HELP screen which gives you general instructions on how to invoke the compiler.
`HELP COMPILE`	prints a HELP screen which outlines the syntax of the COMPILE and UNCOMPILE commands and their parameters.
`HELP OPTIONS`	prints a HELP screen which provides information on each compiler switch and command that can be used in the **options list.** For information on switches and commands in the options list, read the chapter "Details of Using Your Compiler."

Reference

See the section titled "Compiler Command Reference" in chapter 4, page 4-20.

Remove the Compiler (BASIC/WS and BASIC/DOS only)

Considerations

The BASIC Compiler Binary can be removed (without disturbing any system binaries or your program) if it was the last binary loaded and the system was not stored with the STORE SYSTEM command.

Solution

Enter the command:

```
REMOVE COMPILER
```

Reference

See the section titled "Compiler Command Reference" in chapter 4, page 4-20.

Send Compiler Output to Printer

Considerations

No output will be generated by the compiler unless you invoke the compiler with the SHOW, DS or ST command options or unless you include the SYMBOLS directive in one or more of your subprograms. By default, compiler output (if any) will be routed to the current PRINTER IS device.

Solution

The compiler output can be re-routed by specifying a device selector in the invocation command. Enter this command:

```
COMPILE #701
```

This will route your output to the printer at device selector 701.

Reference

See the section titled "Compiler Output" in chapter 4, page 4-20.

Compile Program to Run as Quickly as Possible

Considerations

The speed of your program will depend upon several factors. Using integers instead of real numbers and constants instead of variables whenever possible will improve performance. Also, certain compiler directives or **compiler commands** can be used to enhance program speed.

Solution

In very general terms, to get the fastest code (while possibly increasing the size of your compiled code), you should use the BEST command option in your compiler invocation command:

```
COMPILE: BEST
```

Be aware that BEST will set a number of **compiler directive** defaults, such as EOL OFF and STATICARRAYS ON. These two directives in particular may adversely affect the way your program works. If you compile with BEST and encounter problems, try one of these invocation commands:

```
COMPILE: BEST, EOL ON
COMPILE: BEST, SA OFF
COMPILE: BEST, EOL ON, SA OFF
```

Reference

Chapter/Section Title	Page
STATICARRAYS (SA) Directive	5-40
Writing Efficient Programs	6-3
Optimizing Your Program	6-7
BEST Command	6-12

Optimize Program for Size and Speed

Considerations

Writing efficient programs entails establishing a balance between code size and program speed. This balance can be achieved through the judicious use of compiler directives.

Solution

In general, you will get the best compromise of space and speed by using the compiler invocation command:

```
COMPILE: BEST, OP OFF
```

The BEST command in the options list optimizes your program for speed, and the OP OFF switch optimizes your program for space. Be aware that the BEST command will set a number of compiler directive defaults, such as EOL OFF and STATICARRAYS ON. These two directives in particular may adversely affect the way in which your program works. If you compile with BEST and encounter problems, try one of these invocation commands:

```
COMPILE:  BEST, OP OFF, EOL ON
COMPILE:  BEST, OP OFF, SA OFF
COMPILE:  BEST, OP OFF, EOL ON, SA OFF
```

Reference

Chapter/Section Title	Page
STATICARRAYS (SA) Directive	5-40
Writing Efficient Programs	6-3
Optimizing Your Program	6-7
BEST Command	6-12

Use Compiler Directives

Considerations

Compiler directives are special commands inserted into your program to help control the compilation environment. Different directives may affect your program speed, size, or the degree to which error-checking occurs.

Solution

All compiler directives must appear inside a comment line, as the first sequence of characters following the "!". More than one directive may appear on a single line, if they are separated by semicolons or commas. The syntax is:

!$*directive* [;/, *directive*] ...] [$[*comments*]]

where *directive* is a compiler directive.

Reference

See the section titled "Compiler Directives" in chapter 4, page 4-2.

Use Conditional Compilation

Considerations

Conditional compilation is a useful tool if used carefully. Incorrect use may cause your compiled program to run much differently than your interpreted program, or it may not run at all.

Solution

Conditional compilation uses the directives $IF or $IFNOT. The directive $CONTROLVAR assigns values to **control variables.** For example:

```
10   !$CONTROLVAR A=1,B=0
20   !$IF A
       .
90   ! this segment will be compiled
       .
140  !$END
150  !$IF B
       .
190  ! this segment will not be compiled
       .
250  !$END
260  END
```

Though the code between lines 150 and 250 exists as interpreted code, it *does not* exist in compiled code.

Reference

Chapter/Section Title	Page
CONTROLVAR (CONTROL or C) Directive	5-11
IF and IFNOT Directives	5-16

Use $EOL OFF Correctly

Considerations

The EOL OFF compiler directive can be used to increase the speed and decrease the size of a compiled program by eliminating unnecessary event checking. However, when used improperly, it can cause many problems with your program.

Solution

In general, you should have EOL ON whenever your program will be waiting to acknowledge events such as ON ERROR GOSUB. With EOL OFF, you may encounter some problems, even if you are not processing any events. Read the section specified below before using EOL OFF.

Reference

See the section titled "Using the EOL Directive" in chapter 6, page 6-13.

Build CSUBs Library

Solution

A library of CSUBs consists of a collection of CSUBs which perform similar tasks. For example, you might have a need for a collection of CSUBs which perform specific input and output tasks.

You can build a library of CSUBs by compiling a program that consists of a number of SUBs, where the *main program* consists solely of an END statement. By compiling with the invocation command:

```
COMPILE: KEEP OFF
```

the compiler will compile your SUBs, and delete the source code as compilation progresses. This will leave you with a series of CSUBs.

Reference

Chapter/Section Title	Page
A Dummy MAIN Program and Mainsub	2-6
KEEP Directive	5-20

Set Up Programs with Event Processing

Considerations

The proper use of the EOL directive may be essential to the success of a program that uses event processing. In some cases, this directive must be ON in order for events to be processed correctly.

Solution

When your program depends on event processing, you should leave EOL ON throughout the program. If you want to toggle EOL in programs that process events, you *must* read the section listed below.

Reference

See the section titled "Using the EOL Directive" in chapter 6, page 6-13.

Compile Program without Error Prompts

Considerations

When an error occurs during compilation, the compiler will (by default) pause
3 compilation and ask the user whether to stop compilation so the error can be corrected immediately or continue. Some users may wish to "turn off" the prompt and have the compiler automatically continue compilation when errors occur.

Solution

Compile the program with ERROR OFF in every SUB, or use the compiler invocation command:

```
COMPILE: ERROR OFF
```

Reference

Chapter/Section Title	Page
Switches in the Options List	4-10
Correcting Compile-Time Errors	4-18
Correcting Run-Time Errors	4-19
ERROR (ERR) Directive	5-15

Access MC68020/30 Processor and MC68881/82 Coprocessor

Considerations

By default, the compiler generates code that checks for the presence of the MC68020 or MC68030 processor and the MC68881 or MC68882 coprocessor and uses them if they are present. You may wish to eliminate these checks and have the compiler generate code that is dependent upon the MC68020/30 and/or MC68881/82 (co)processor.

Solution

Use the MC68020 ON and/or MC68881 ON directives in any SUB that wants to eliminate these checks, or use the compiler invocation command:

```
COMPILE: MTP, MCP
```

The MC68020 directive accesses the MC68020 and MC68030 processors. This manual refers to these processors in common as MC68020/30. The MC68881 directive accesses the MC68881 and MC68882 coprocessors. This manual refers to these coprocessors in common as MC68881/82.

Reference

Chapter/Section Title	Page
Switches in the Options List	4-10
MC68020 (MTP) Directive	5-24
MC68881 (MCP) Directive	5-25

4

Details of Using Your Compiler

This chapter explains how to install and use the BASIC Compiler. (If you are using BASIC/UX, you won't need to install the compiler—this is done when you install BASIC/UX.) The following topics are covered:

- invoking the compiler
- using the compiler commands
- using the switches and commands in the options list
- correcting compile- and run-time errors
- producing compiler output

A command reference is included at the end of the chapter.

Installing the BASIC Compiler (BASIC/WS and BASIC/DOS only)

If you are using BASIC/WS or BASIC/DOS, you will need to install the BASIC Compiler binary before you can use it. Insert the disk containing the compiler into a disk drive connected to your computer and type:

```
LOAD BIN "COMPILER [:msvs]"
```

Next press (EXECUTE) ((ENTER), (EXEC) or (Return)). Be sure that the correct *msvs* (Mass Storage Volume Specifier) is included in the file name. For example:

```
COMPILER:HP82901,700,1
```

The default *msvs* will be used if none is specified.

Once the BASIC Compiler binary is installed, you can compile any BASIC program residing in memory. A list of **compiler invocation commands** appears later in this chapter.

Note The BASIC compiler binary is removable under certain conditions (see the section in this chapter titled "REMOVE COMPILER"). As long as the BASIC Compiler was the *last* binary loaded and the system was not stored with the STORE SYSTEM command, it can be removed using the REMOVE COMPILER command (without disturbing any system binaries or your program).

Compiler Directives

Compiler directives are special commands that can be included in your interpreted BASIC program to control the compiling environment. They do not affect the semantics of the program (unless you use them incorrectly). Compiler directives can be very helpful for debugging, optimizing the speed of your program, or optimizing the size of your compiled code. A full list of the available compiler directives and instructions on how to use them appears in the chapter "Compiler Directives."

The chapter titled "Improving Compiled Programs" contains a section describing how each directive affects the size and/or the speed of your compiled program. We recommend that you read this chapter, especially the section on the EOL directive.

Invoking the Compiler

Once the compiler has been installed, it is ready for use. When invoked, the compiler will compile a program residing in memory—so make sure there is one present. You can either LOAD a program residing on an external device or develop a new program in memory. The compiler is then invoked via a compiler invocation command. Topics covered in this section are:

- COMPILE command
- UNCOMPILE command
- Device Selector
- Compile List
- Options List

The COMPILE Command

The COMPILE command, with nothing following it, is the simplest of all the compiler invocation commands. When you enter the command:

```
COMPILE
```

the BASIC compiler will compile the entire program that is currently residing in memory. If the compiler was invoked with this command and it encounters a subprogram that is already compiled (for which compiled code already exists), that subprogram will only be re-compiled if it has been modified since the last compilation. The compiler "marks" each line of a program as it is compiled. If the compiler encounters a line that has no mark, it recognizes that the subprogram has had changes made to it. The compiler can also tell if a line was deleted and it recompiles the subprograms.

The COMPILE command may be followed by an options list, a **compile list**, and/or a device selector. These are described in detail later in this chapter.

Default Compiler Directives

When you invoke the compiler via the COMPILE command (without specifying an options list), you are producing the "safest" code. It may not be the fastest or most compact code possible, but it will have all error checking and a number of other "safe" features enabled. The compiler directives will have the following values when the compiler is invoked with the command COMPILE:

`CONFIGCHECK ON`	enables configuration checks.
`EOL ON`	generates end-of-line activity code.
`KEEP ON`	preserves both the source and the compiled code.
`LONGCODE OFF`	keeps extended addressing from being generated.
`MC68020 OFF`	keeps MC68020/30-dependent code from being generated. This directive's default state is ON if running on a computer with a MC68020/30 processor or MC68881/82 coprocessor.
`MC68881 OFF`	keeps MC68881/82-dependent code from being generated. This directive's default state is ON if running on a computer with a MC68881/82 coprocessor.
`OPTIMIZE OFF`	optimizes a program for space, not speed.
`OVERFLOWCHECK ON`	provides integer/string overflow checking.
`RANGECHECK ON`	provides checking for out-of-range values.
`STATICARRAYS OFF`	allows the use of REDIM.
`SAVELINENUMBER OFF`	prevents line numbers from being retained.
`STACKCHECK ON`	enables system stack checking.
`ERROR ON`	emits a user prompt when errors occur.

When the compiler finishes, both the compiled code and the source code will exist in memory. If you edit a compiled program, you will see something like:

```
1       Mainsub
2       END
3  CSUB Mainsub
4       SUB Mainsub
        .
21      SUBEND
22 CSUB Subone
23      SUB Subone
        .
30      SUBEND
31 CSUB Subtwo
32      SUB Subtwo
        .
53      SUBEND
```

Each BASIC SUB is preceded by a line of the form:

```
CSUB subprogram name
```

This line represents the compiled version of the subprogram. Remember that if the compiler encounters a subprogram for which compiled code exists, that subprogram will only be re-compiled if it has been modified (unless the COMPILE was followed by a compile list). To re-compile a SUB that has *not* been modified, you can:

- EDIT the program, and delete the line representing the compiled code. You can then type in COMPILE which will cause *all* uncompiled subprograms to be compiled.
- UNCOMPILE that SUB (or the entire program), and then COMPILE it again. See the next section titled "The UNCOMPILE Command."
- You can specify a compile list (after the keyword COMPILE) which contains the name of the subprograms you wish to compile. See the section "The Compile List."

The UNCOMPILE Command

The UNCOMPILE command removes compiled code from memory restoring your interpreted program to its original form *provided that the source code is still in memory.* To uncompile a compiled program which is currently in memory, type:

```
UNCOMPILE (Return)
```

This causes the compiler to go through your program and remove (delete) all of the compiled code (leaving the source code intact). With this command, only those subprograms *for which source code exists* will have the compiled code removed. If you have deleted the source code for a subprogram, or if you compiled a subprogram without keeping the source code, the UNCOMPILE command will have no effect on its associated CSUB.

The UNCOMPILE command may also be followed by an options list, a compile list, and/or a device selector. These are described in detail later in this chapter.

If you want to remove compiled code for a SUB whose source code is not present, you have two choices:

- EDIT the program, and delete the CSUB lines from the program
- Use the DELSUB command followed by the appropriate subprogram name

Note The UNCOMPILE command *cannot* change compiled code back into source code. It simply removes compiled code (provided that the source code is resident in memory).

Device Selectors

A device selector can be specified in the invocation of the COMPILE command. The purpose of the device selector is to route output produced by the compiler to a device other than the default set by the PRINTER IS command. If a device selector is included in the invocation command, it must immediately follow the keyword COMPILE or UNCOMPILE and be preceded by a **#**. For example:

```
COMPILE #701
```

will send the output to a device located at device selector 701 (usually the printer). If no device selector is specified, output will be routed to the current PRINTER IS device.

A device selector is either an interface select code or a combination of an interface select code and a primary address. To construct a device selector with a primary address, multiply the interface select code by 100 and add the primary address. For example, if you have an interface select code of 7 and a primary address of 1, your device selector would be 701.

By default, the compiler runs silently (does not produce any output or compilation messages) except when error messages occur. Therefore, unless you have included a directive in your program, or a command in the options list that will generate output, changing the device selector will have no effect.

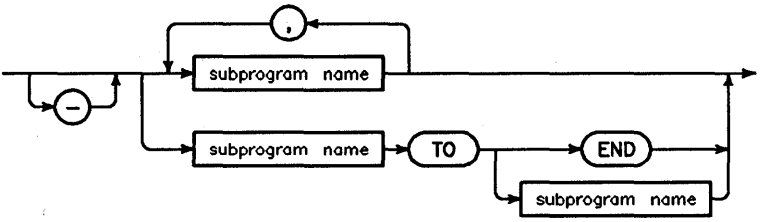
Compile List

A compile list can be specified after either the COMPILE or UNCOMPILE command. The compile list is a list of SUBs that are to be COMPILEd or UNCOMPILEd. When a compile list is specified, the action requested will be performed on all items in the list. In other words, if a compile list follows the COMPILE command, *all* subprograms in the list, and *only* those subprograms, will be compiled. Note that all subprograms in the list will be compiled, or re-compiled if they have already been compiled.

If a compile list follows the UNCOMPILE command, all of the subprograms in the list, and *only* those subprograms, will have their compiled code removed.

The compiled code will *only* be removed if the source code exists in memory along with the compiled code.

The compile list has the following syntax:



4

where *subprogram name* is the name of a subprogram in your program, or MAIN to designate the main program. The following are all valid forms:

[-] *subprogram name* TO *subprogram name*

[-] *subprogram name* TO END

[-] *subprogram name* [, *subprogram name*] ...

If the compile list is preceded by a minus sign (−), then all subprograms *except* those in the compile list will be compiled. Similarly, if you specify a compile list with the UNCOMPILE command, and if the compile list is preceded by a minus sign (−), all subprograms *except* those in the compile list will be uncompiled (have their associated CSUBs removed). Again, only those subprograms for which source code exists in memory will be uncompiled.

Compiler Invocation Examples

Here are some examples of valid compiler invocation commands using a compile list:

```
COMPILE MAIN
```

will compile the dummy main program.

This command:

```
COMPILE #701; -A,SUBTWO,FIRST
```

will compile (or recompile) *all* subprograms in the program, *except* for the subprograms A, SUBTWO and FIRST. The output for this compilation has been re-routed to the device located at device selector 701. This command:

```
UNCOMPILE THISSUB TO THATSUB
```

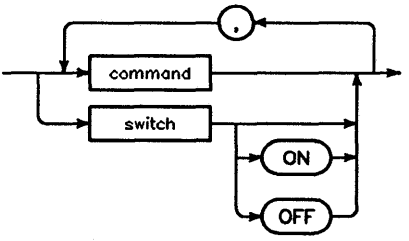
will remove the compiled code for the subprograms THISSUB, THATSUB, and all the subprograms in between them. (The compiled code will only be removed if the source code exists in memory). Finally, this command:

```
COMPILE SUBTWO TO END
```

will compile (or recompile) the subprogram SUBTWO and every subprogram that occurs *after* it.

Options List

The options list is a list of options which will affect how a program is compiled. The syntax is:



Items in the options list are identified as either *switches* or *commands.*

Note Only the short form of switches and commands can be used in the options list. For example, you would use CC OFF instead of CONFIGCHECK OFF.

Switches in the Options List

The switches are used to change the default values of compiler directives. Each compiler directive has a certain default value that is used if that directive does not appear in your program. The switches can be used to change these defaults. Switches will have no effect on subprograms in which the corresponding compiler directives appear. Switches affect only subprograms in which the corresponding directives *do not* appear. Switches are ignored if they are used in conjunction with UNCOMPILE.

Each switch can appear alone or followed by ON or OFF. A switch appearing alone is equivalent to the switch followed by ON. If a switch appears in the options list more than once, the *last* value will be used. Switches can be interspersed with commands in the options list.

The switches and their corresponding compiler directives are covered in the following sections. For a more detailed explanation of the action of a specific switch and how it may affect your code, refer to the description of its corresponding compiler directive in the chapter "Details of Compiler Directives."

CC (corresponds to CONFIGCHECK)

By default, the compiler will emit code to check for the existence of binaries, subprograms, and functions before they are called. Using the CC ON (or CC) switch leaves this default value unchanged. If CC OFF is specified in the options list, the compiler will emit such checks only in those subprograms containing the CONFIGCHECK ON directive.

EOL (corresponds to EOL)

By default, the compiler emits end-of-line activity code which allows the operating system to service pending events such as key presses and interrupts. Using the EOL ON (or EOL) switch leaves this default value unchanged. If EOL OFF is specified in the options list, the compiler will emit end-of-line activity code only in subprograms containing the EOL ON directive. For more information, read the section "Using the EOL Directive" in the chapter "Improving Compiled Programs."

ERR (corresponds to ERROR)

By default, when an error or warning is encountered during compilation, the compiler will give you the choice of stopping immediately to correct the error or continuing compilation. If ERR ON (or ERR) is in the options list, the default remains as is. If ERR OFF appears in the options list, the compiler will automatically continue compilation without the prompt unless the error occurs in a subprogram containing the ERR ON directive.

KEEP (corresponds to KEEP)

By default, the compiler will keep the source code along with the compiled code as your program is compiled. Specifying KEEP ON (or KEEP) in the options list leaves this default value unchanged.

Specifying KEEP OFF in the options list will cause the compiler to save the source code for *only* those subprograms containing the KEEP ON directive.

LC (corresponds to LONGCODE)

By default, the compiler will not generate object code with extended addressing. Extended addressing is necessary for jumps and addresses to string constants that are more than 32K bytes away. Note that the compiler informs you if the LONGCODE directive is necessary for any of the **contexts.** If LC OFF is used in the options list, the default will remain as is. If LC ON (or LC) is used in the options list, the compiler will generate object code with extended addressing in all subprograms not containing the LONGCODE OFF directive. We recommend that you use LONGCODE ON in only those subprograms that require it.

MTP (corresponds to MC68020)

The default value for MC68020 is ON for computers with a MC68020 or MC68030 processor; otherwise, the default is OFF meaning that the compiler generates code that checks for the presence of the MC68020/30 processor and uses it if it is present. Compiling with MC68020 ON causes the compiler to generate code that runs only on the MC68020/30 processor.

MCP (corresponds to MC68881)

The default value for MC68881 is ON for computers with a MC68881 or MC68882 coprocessor; otherwise, the default is OFF meaning that the compiler generates code that checks for the presence of the MC68881/82 coprocessor and uses it if it is present. Compiling with MC68881 ON causes the compiler to generate code that runs only on the MC68881/82 coprocessor.

OP (corresponds to OPTIMIZE)

By default, the compiler will optimize your program for space. Using OP OFF (or OP) will keep the default as is. If OP ON is specified, your program will be optimized for speed rather than for space except in those subprograms containing the OPTIMIZE OFF directive.

OC (corresponds to OVERFLOWCHECK)

The compiler normally emits code to check for integer and string overflow after each integer and string computation. Using the OC ON switch will keep the default as is. Using the OC OFF switch will cause the compiler to *not* emit such checks except in those subprograms containing the OVERFLOWCHECK ON directive.

RC (corresponds to RANGECHECK)

The compiler normally emits code to check, at run time, for out of range values before they are used. The RC ON (or RC) switch will keep the default as is. If you specify RC OFF in the options list, range checking will be done in only those subprograms that contain the RANGECHECK ON directive.

SA (corresponds to STATICARRAYS)

By default, the compiler assumes that you may be using the REDIM statement in your program. Using the SA OFF switch will keep this default as is. If you specify SA ON (or SA) in the options list, the compiler will assume that a subprogram contains only static arrays (thus saving space and significantly increasing the speed of array access for local arrays) unless that subprogram contains the STATICARRAYS OFF directive.

SL (corresponds to SAVELINENUMBER)

The SL command is useful when you need to know the line number where a run-time error occurred. Compiling a program using the SL ON command causes all subprogram lines to be saved. When you run this compiled program and a run-time error occurs the correct line number where the error occurred will appear in the error message. If this same program had been compiled using the SL OFF command, you would have received only the line number of the line calling the compiled subprogram. This is not the line number where the error occurred.

Normally, the compiler *does not* emit code to save the line number before a line is executed. With SL OFF in the options list, the default remains the same. If SL ON (or SL) is specified in the options list, the compiler will save the line numbers in all subprograms except those containing the SAVELINENUMBER OFF directive.

ST (corresponds to SYMBOLS)

When this switch is included in your options list, the symbol table for each subprogram will be displayed after it is compiled. The symbol table produced will look something like:

```
-------  SYMBOL TABLE DUMP for  Subone

    NAME            TYPE       KIND        SCOPE
---------------  -------  --------  ---------------
Sub_x            REAL     SUB       Global
A                INTEGER  VARIABLE  Parameter
STR$             STRING   VARIABLE  Parameter
Lbl                --     LABEL     Local
FNWork$          STRING   SUB       Global
Intarray         INTEGER  ARRAY     Local
D_array          REAL     ARRAY     Local Dynamic
Com_one            --     COM LABEL Global
C_array          REAL     ARRAY     Common Variable
S$               STRING   VARIABLE  Local
@File            IO PATH  VARIABLE  Local
80                 --     LABEL     Local
-------------------------------------------------------
Number of Entries in Table = 17
Number of Used Entries     = 12
```

The **Used Entries** as seen on the above display are covered in the section titled "Unused Entries" found in the chapter "Debugging Programs" in the *HP BASIC Programming Techniques* manual.

STC (corresponds to STACKCHECK)

The compiler automatically emits code to check for overflow of the system stack before variable space is allocated. Using STC ON (or STC) in the options list keeps the default as is. Using STC OFF in the options list will cause these checks to be made only in those subprograms containing the STACKCHECK ON directive.

Commands in the Options List

Commands that appear in the options list will affect a certain feature of the compiler. A command is not followed by ON or OFF, like a switch—a command will just appear as a single word. Commands (with the exception of the SHOW command) will be ignored if they are used in conjunction with the UNCOMPILE command. The commands that are available and their effects are described in detail below.

BEST Command

If the BEST command appears in the options list, the compiler directives listed below are set to values that produce the fastest code. The compiler directives are set to:


```
CONFIGCHECK OFF     OPTIMIZE ON          SAVELINENUMBER OFF

EOL OFF             OVERFLOWCHECK OFF    STACKCHECK OFF

LONGCODE OFF        RANGECHECK OFF       STATICARRAYS ON
```

Keep in mind that BEST will set EOL OFF and STATICARRAYS ON, which may cause serious problems with your program. Make sure that all the directive values set with BEST are valid in your program. When using the BEST command, you may use switches to override one or more of the directives set by BEST. For example, either of these:

```
COMPILE : BEST, EOL ON
COMPILE:  EOL ON, BEST
```

will set all the BEST conditions, then enable the generation of end-of-line activity code. If you set a switch more than once, the *last* specified value will be used. For example:

```
COMPILE:  EOL ON, BEST, EOL OFF
```

will use EOL OFF, while this:

```
COMPILE:  EOL OFF, EOL ON, BEST
```

or this:

```
COMPILE:  EOL OFF, BEST, EOL
```

will use EOL ON.

DS Command

If the DS command appears in the options list, the compiler prints out (dumps) statistical information on each subprogram after it is compiled. As soon as the compiler finishes compiling a subprogram or function, a table like the following will be printed on the output device (specified as the device selector or current PRINTER IS device):

```
CSUB Header          :      60 bytes ==>   9 %
Symbol Tables        :      82 bytes ==>  12 %
CSUB Entry Code      :     404 bytes ==>  61 %
CSUB Body Code       :      84 bytes ==>  13 %
Event Lines          :       0 bytes ==>   0 %
Constant Pool        :       4 bytes ==>   1 %
Added Libraries      :       0 bytes ==>   0 %
Relocation Tables    :      26 bytes ==>   4 %
Data Statements Pool :       0 bytes ==>   0 %
Local DIM Table      :       0 bytes ==>   0 %
TOTAL CODE           :     660 bytes ==> 100 %
```

Note that the percentages in the far right column are rounded up to the nearest percent, so they may add up to more than 100.

If a program contains more than one SUB or DEF, a summary table will be printed at the end of compilation which generates the above information about the total code that was generated during compilation.

The DS command and the tables it produces are explained in detail in the chapter "Improving Compiled Programs."

HIDE Command

This command provides additional security to your program when used with the KEEP OFF switch.

In a compiled program, each subprogram becomes a CSUB and each function becomes a CDEF. Each CSUB or CDEF is followed by a list of the subprogram's parameters. So this header:

```
SUB Sample (String$,INTEGER Iarray(*),REAL Temp)
```

when compiled, will by default become:

```
CSUB Sample (String$,INTEGER Iarray(*),REAL Temp)
```

When the HIDE command appears in the options list the parameter lists for each of your compiled subprograms will be "hidden." Each parameter name will be replaced by a single blank character. So, the header shown above becomes:

```
CSUB Sample ( ,INTEGER (*),REAL )
```

after compilation with HIDE. Note that the parameter names are hidden, but the types are not.

If the program source was retained during compilation, the parameter list of the SUB source will not be affected by the use of this command.

SHOW Command

By default, the compiler is "quiet", meaning it does not display any messages while it is running. The SHOW command can be used to cause the compiler to produce output about the status of compilation. If SHOW is used, a message such as:

```
COMPILING X
```

will appear as each subprogram is compiled or uncompiled. After each subprogram is compiled, the compiler will display a list of all the compiler directives, and the values that were in effect during the compilation of that subprogram such as:

```
Config Check  :ON   Emit Long Code:OFF  EOL Set/Check :ON   MC68020 Code :OFF
MC68881 Code  :OFF  Optimize Code :OFF  Overflow Check:ON   Range Check  :ON
ROM Based Only:ON   Save Line Num.:OFF  Stack Check   :ON   Static Arrays:OFF
```

All of this output will go either to the current PRINTER IS device by default, or to the device you specified in the compiler invocation command.

Correcting Compile-Time Errors

If the compiler encounters an error during compilation, it will print the relevant line number, error message, and error number. If you did not specify a device selector in the invocation command, these messages will be printed to the current PRINTER IS device. If you *did* specify a device selector, the printout of the error message will go to that device.

After the error message is printed, the compiler will pause and ask you if you wish to continue. This prompt will *always* go to the screen, even if you have specified an alternate device selector for output. If your answer to the prompt is:

- YES (you would like to continue)—the compiler will "skip" the error, and the subprogram in which the error was found, and will continue compiling. The subprogram containing the error will remain uncompiled, but any subprograms that are error free will be compiled. You can, after compilation, go back and correct any errors that were detected.
- NO (you would not like to continue)—the compilation process will halt, and you can EDIT the program to correct the error immediately. The subprogram containing the error will not be compiled, but all preceding (error-free) subprograms will have been compiled.

The ERROR OFF compiler directive (or switch) can be used in your program if you don't want the compiler to prompt for error correction. If the compiler encounters an error in a subprogram which uses the ERROR OFF directive, the compiler will assume that you want to continue compilation. It will still print out the associated error message and number to the current PRINTER IS device (unless output was re-routed using a device selector) so you can correct any errors after compilation is complete.

Correcting Run-Time Errors

When you encounter an error at run time, remember these four rules:

- If the subprogram containing the error was compiled with SAVELINENUMBER OFF (or SL OFF which is the default), the line number reported with the error will be the line number of the CSUB in which the error occurred.
- If the subprogram containing the error was compiled with SAVELINENUMBER ON and KEEP ON, the line number reported with the error will be the line number within the SUB following the CSUB being executed (which is the source code form of the CSUB).
- If the subprogram containing the error was compiled with SAVELINENUMBER ON and KEEP OFF, the compiler will assume that you are keeping a copy of your source code elsewhere. Therefore, the line number that is reported with the error message will correspond to the line number of your original BASIC program (before the program was compiled).
- Separately compiled subprograms with SL ON will report the wrong line number if a run-time error occurs in the compiled subprogram. For example, you may compile a subprogram using COMPILE: SL with line numbers ranging from 1 to 10 and store it in a file. First make sure that this subprogram will cause a run-time error such as division by zero. Perform a LOADSUB of the compiled CSUB into another program which contains lines with line numbers greater than 10. The loaded CSUB will end up with a line number greater than 10. Running the program which calls the compiled CSUB will generate a run-time error which points to a line number less than 10. This line number should be in the CSUB, but when you EDIT the program you will be looking at the wrong line. This is due to the fact that when the CSUB was called it told the BASIC Language System that its line numbers are in the range of 1 to 10 since it was compiled that way. The CSUB had no knowledge of where it was loaded.

  This problem can be avoided by not separately compiling subprograms with SL ON, so when an error occurs the system points to the CSUB line.

Compiler Output

Normally the compiler will run silently (i.e., no printed output will be produced). The only way that you can tell the compiler is running is from the run light in the lower right hand corner of your screen. However, there are three compiler commands that can be used with the COMPILE command to produce output:

- SHOW
- DS
- ST

4

The command UNCOMPILE will produce output if it is used in conjunction with the command SHOW. See the previous sections titled "The UNCOMPILE Command" and "SHOW Command" for details on these commands.

The printed output produced by the compiler can be re-routed by specifying a device selector in the invocation command, such as:

```
COMPILE #701: SHOW
```

The command above would route all output to *device selector* 701. If you specify a device selector, the printed output generated by the compiler (such as symbol tables, statistical information, error messages, etc.) will go to the device specified. If, you do not specify a device selector, output will to go the current PRINTER IS device.

Compiler Command Reference

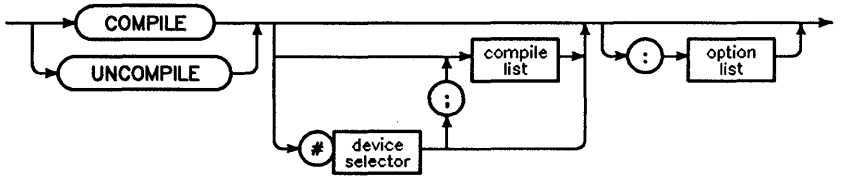
This section briefly describes the COMPILE and UNCOMPILE commands, as well as four more commands that can be used with the compiler, but which do not actually invoke the compiler. These commands are:

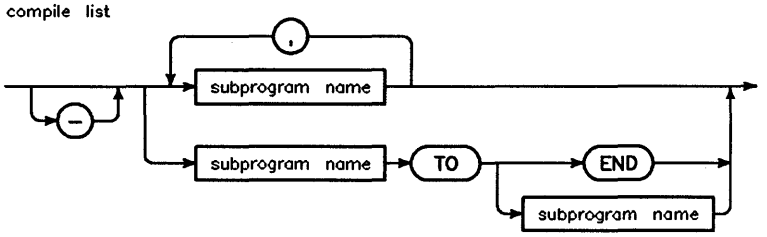
- HELP
- HELP COMPILE
- HELP OPTIONS
- REMOVE COMPILER

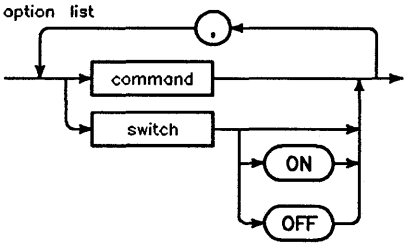
COMPILE and UNCOMPILE

Option Required	COMPILER
Keyboard Executable	Yes
Programmable	No
In an IF ... THEN	No

These commands compile and uncompile a program that is currently in memory.







Item	Description	Range
device selector	numeric expression, rounded to an integer	see the "BASIC Language Reference" glossary
compile list	a list of one or more subprogram names	any valid name
option list	a list of one or more options representing compiler commands and switches	see the section "Options List"
subprogram name	name of a subprogram or function	any valid name
command	controls the compilation environment	see the section "Commands in the Options List"
switch	a directive which can change its own default value	see the section "Switches in the Options List"

Example Statements

```
COMPILE
UNCOMPILE
COMPILE MAIN
COMPILE #701
UNCOMPILE #705; Program_1
COMPILE:BEST
COMPILE #701; -SUBONE,SUB19,SUB_1
COMPILE #701; Srq_ck,Printer:KEEP ON
COMPILE Sub_here TO Sub_there
COMPILE Sub_seven TO END
UNCOMPILE Sub_second TO Sub_sixth
UNCOMPILE -Sub_one,Sub_two
COMPILE #701; -Sub_one,Sub_seven:BEST,MTP OFF
```

Semantics

The device selector part of the COMPILE statement determines where the output will be routed for printing. If no device selector is given, output will be routed to the current PRINTER IS device.

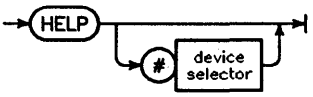
MAIN in a compile list is a valid name for a subprogram, and refers to the main program. Also, if the compile list is preceded by a minus sign (−), then the list is used to indicate the subprograms that will *not* be compiled/uncompiled. In other words, all subprograms except those appearing in the list will be compiled/uncompiled.

Once the source code has been removed, you can no longer use the UNCOMPILE command to remove the compiled code.

HELP

Option Required	COMPILER
Keyboard Executable	Yes
Programmable	No
In an IF ... THEN	No

This command prints a HELP screen, which gives you general instructions on how to invoke the compiler.



Item	Description	Range
device selector	numeric expression, rounded to an integer	see the "BASIC Language Reference" glossary

Example Statements

```
HELP
HELP #701
```

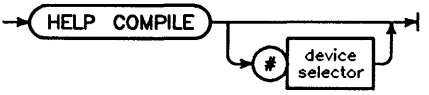
Semantics

If no device selector is specified, the output will go to the current PRINTER IS device.

HELP COMPILE

Option Required	COMPILER
Keyboard Executable	Yes
Programmable	No
In an IF ... THEN	No

This command will print out a HELP screen which outlines the syntax of the COMPILE and UNCOMPILE commands and their parameters.



Item	Description	Range
device selector	numeric expression, rounded to an integer	see the "BASIC Language Reference" glossary

Example Statements

```
HELP COMPILE
HELP COMPILE #701
```

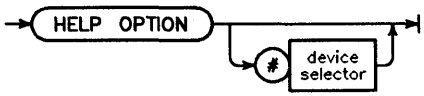
Semantics

If no device selector is specified, the output will go to the current PRINTER IS device.

HELP OPTIONS

Option Required	COMPILER
Keyboard Executable	Yes
Programmable	No
In an IF ... THEN	No

This command will print out a HELP screen which provides information on each **compiler switch** and command that can be used in the options list.



Item	Description	Range
device selector	numeric expression, rounded to an integer	see the "BASIC Language Reference" glossary

Example Statements

```
HELP OPTIONS
HELP OPTIONS #701
```

Semantics

If no device selector is specified, the output will go to the current PRINTER IS device.

REMOVE COMPILER (BASIC/WS and BASIC/DOS only)

Option Required	COMPILER
Keyboard Executable	Yes
Programmable	No
In an IF ... THEN	No

This command will remove the compiler binary from memory for BASIC/WS and BASIC/DOS. Do not use this command with BASIC/UX.

Item	Description	Range
device selector	numeric expression, rounded to an integer	see the "BASIC Language Reference" glossary

Example Statement

```
REMOVE COMPILER
```

Semantics

The COMPILER binary is not needed after compiling a program. Therefore, you may want to use the REMOVE COMPILER command to create extra space to load data such as user subprograms or other binaries.

The REMOVE COMPILER command removes the COMPILER binary only if it was the last one loaded and *is not* part of the system (STORE SYSTEM was *not* used to store the system and the COMPILER binary). If the REMOVE COMPILER command cannot be used, the compiler prints a message explaining the reason.

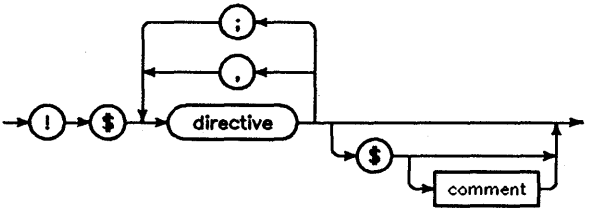
Using this command will not disturb any system binaries or your program.

5

Details of Compiler Directives

Compiler directives are special commands used to control the compilation process. They do not affect the semantics of the compiled program. Since directives are not part of the program, they must appear inside a comment line, on a line by themselves (though more than one directive may appear on the same line, separated by semicolons or commas). They are permitted only as the first sequence of characters following the "!". The general syntax of a compiler directive is:

! $*directive* [; / , *directive*] ... [$[*comments*]]



Compiler directive lines begin with a $ (after the !), to tell the compiler that this comment line contains one or more compiler directives, and end with an optional $. Characters appearing after the second $ are considered to be comments, and will be ignored by the compiler. For example:

```
10   ! $KEEP OFF $ Removes the source code from the compiled program.
```

Default Values of Compiler Directives

Each compiler directive has a certain default value that is used if the directive is not explicitly set. All of the directives, except the CONTROLVAR directive, are **context-scoped,** which means they are set (or re-set) to their default values at the beginning of each context. The CONTROLVAR is **global-scoped,** which means that it retains its value across each context boundary. A context is a section of code (a subprogram) beginning with a SUB or DEF statement and ending with a SUBEND or FNEND statement. The main program is also considered a context.

To change default values, you can use directives or options in the options list of the COMPILE command (the short form of the compiler directive). For example, to change the default value for range checking (default is RANGECHECK ON), you could include this statement at the beginning of a context:

5

```
10   ! $RANGECHECK OFF
```

The default value for range checking of an entire program can be changed using this command:

```
COMPILE : RC OFF
```

Note that this command will only eliminate range checking in those contexts which do not contain a RANGECHECK ON directive in them.

Set Location of Compiler Directives

Each compiler directive also has a certain "set location" where it is valid for that directive to appear. If a directive's set location is the beginning of each context, then it must appear before the first executable statement of that context or it will be ignored by the compiler. If a directive's set location is anywhere, it will be valid anywhere in a context. Directives that can appear anywhere within a context can be toggled ON or OFF as many times as is desired.

Compiler directives that are inserted into the dummy MAIN program (generated by the compiler) will be ignored unless the dummy main was

compiled using the COMPILE MAIN command. This restriction will only affect the CONTROLVAR directive in most cases.

For directives that can be ON or OFF (switches), the directive can be toggled ON by including the name of the directive (long or short form) optionally followed by ON. The directive name appearing alone is equivalent to toggling the directive ON. To toggle a directive OFF, the directive name (long or short) should be followed by OFF.

When using ON or OFF, you *must* include a space after the directive name.

A Detailed Description of Compiler Directives

The long and short form of each compiler directive appears in this section. The short form of a directive is simply an abbreviated form of the directive. For example, OC ON will have exactly the same effect as OVERFLOWCHECK ON. The paragraph immediately following the name of the directive gives a complete description of the function of the directive and outlines the syntax. Also included with the description of the directive are the topics given below.

Directive Topics

Default

Shows what default is used if the directive is not specified and if it has not been changed by using its associated switch in the options list of the compiler invocation command.

Scope

Tells whether the directive is context-scoped or global-scoped.

Set Location

Shows the set location of the directive which is either the beginning of the context or anywhere within a context.

Details

Provides any additional information that you may need to use the directive.

Directives Overview

The following table lists each directive with its short form, default value, set location, and a brief description of what feature of compilation may be affected by its use.

Directive	Short Form	Default	Set Location	Compiler Feature Affected
COMPLEX	none	none	Anywhere	Returns COMPLEX function results
CONFIGCHECK	CC	ON	Anywhere	User subs and system binaries check
CONTROLVAR	C	none	Anywhere	Control variable values
EOL	none	ON	Anywhere	End of line activity code
ERROR	ERR	ON	Anywhere	Prompt when errors occur
IF	none	none	Anywhere	Conditional compilation
IFNOT	none	none	Anywhere	Conditional compilation
KEEP	none	ON	Anywhere	Keep the Source code
LONGCODE	LC	OFF	Beginning	Extended addressing code
MC68020	MTP	[1]	Beginning	Code specific for the MC68020/30 processor
MC68881	MCP	[2]	Beginning	Code specific for the MC68881/82 math coprocessor

[1] If the computer used has a MC68020/30 processor then the default is ON.

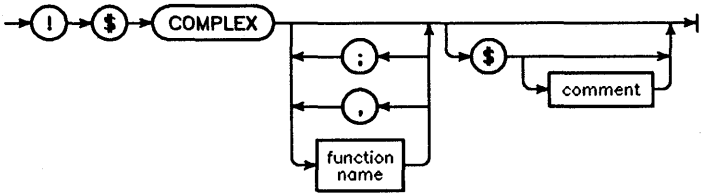
[2] If the computer used has a MC68881/82 coprocessor then the default is ON.

Directive	Short Form	Default	Set Location	Compiler Feature Affected
OPTIMIZE	OP	OFF	Beginning	Space (OFF)/speed (ON) optimization
OVERFLOWCHECK	OC	ON	Anywhere	Overflow checking
RANGECHECK	RC	ON	Anywhere	Out-of-range value check
REAL	none	none	Anywhere	Returns REAL function results
SAVELINENUMBER	SL	OFF	Anywhere	Save program line numbers
STACKCHECK	STC	ON	Anywhere	System stack bound check
STATICARRAYS	SA	OFF	Beginning	Use of static local arrays (REDIM not used)
SYMBOLS	ST	OFF	Anywhere	Symbol table listing

COMPLEX Directive

By default, the value returned by any function in your program will be of type REAL. The directive COMPLEX allows you to specify certain functions that will return a value of type COMPLEX. The syntax is:

! $COMPLEX *function name* [;/, *function name*] ... [$*comment*]



5

Each *function name* is the name of a function that is to return a COMPLEX value.

Default

By default, all functions return a REAL value.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

When using this directive, the first two letters of any specified function name must be FN or warning 21 (Improper ID on Declaration Directive) will be generated at compile time. If a specified function name ends with the character $, warning 21 will occur at compile time. Warning 21 will also be generated

any time a specified function is not found in the symbol table (i.e., if a function is declared using this directive but is not called within the context). If warning 21 is ignored (you continue compilation without correcting the error), the function will be assumed to be of type REAL.

This directive must appear once in every context (subprogram) that calls the function *before* the function is called and in the function itself. For example:

```
10   COMPLEX C
20   !$COMPLEX FNOne
30   C=FNOne(10)
40   END
50   !
60   DEF FNOne(X)
70   !$COMPLEX FNOne,FNTwo
80   RETURN CMPLX(X,0)+FNTwo(X,2*10)
90   FNEND
100  !
110  DEF FNTwo(R,I)
120  !$COMPLEX FNTwo
130  RETURN CMPLX(I,R)
140  FNEND
```


If a function was declared to be of a given type (using the directive COMPLEX or REAL) and the function returns a value of a different type, a run-time error will occur stating that the value returned is not of the correct type.

The directive in the calling context (as in line 20 in the previous program) indicates what type the calling context should expect to receive (the expected type). The directive in the function itself (as in line 120 in the previous program) indicates what type should be returned by the function (the returned type). The value to be returned from a function will be converted to the returned type (if necessary) before the value is returned. When the value is returned, the returned and expected types are compared: if they are different, error 19 (Improper value or value out of range) will be generated. In this program sample, the value of function `FNOne` (12, as indicated in line 80) will be converted to the COMPLEX value (12.,0.) before it is returned.

```
10   !$COMPLEX FNOne
20   REAL X
30   X=FNOne
40   END
50   !
60   DEF FNOne
70   !$COMPLEX FNOne
80   RETURN 12
90   FNEND
```

After it is returned the value will be converted to REAL, since X is a real value, and then assigned to X on line 30.

The REAL directive can also be used to declare the type of a function. These directives (REAL and COMPLEX) can be used in conjunction to toggle the type returned by a function. For example:

```
10   COMPLEX C
20   REAL R
          .
          .
100  !$COMPLEX FNOne
110  C=FNOne(1)
120  !$REAL FNOne
130  R=FNOne(2)
140  END
150  !
160  DEF FNOne(Parm)
170  COMPLEX C
180  REAL R
190  IF Parm=1 THEN
200     !$COMPLEX FNOne
          .
          .
240     RETURN C
250  ELSE
260     !$REAL FNOne
          .
          .
310     RETURN R
320  END IF
330  FNEND
```

Keep in mind that any time the returned and expected types differ, an error will be generated. So, the following code will generate an error:

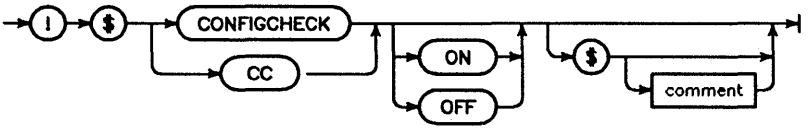
```
10   !$COMPLEX FNOne
20   R=FNOne
30   END
40   !
50   DEF FNOne
60   RETURN 2
70   FNEND
```

Line 10 indicates that the expected type of FNOne is COMPLEX. Since there is no directive in FNOne declaring its type, the function will return a REAL value by default. An error will occur on line 20 at run time when the expected and returned types do not match.

CONFIGCHECK (CC) Directive

This directive causes the compiler to emit code to check (at run time) for the existence of binaries, subprograms, and functions before they are called. The syntax is:

```
! $CONFIGCHECK [ON/OFF] [$[comments]]
```



where CONFIGCHECK OFF suppresses these checks.

5

Default

The default for this directive is CONFIGCHECK ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere that it appears within a context.

Details

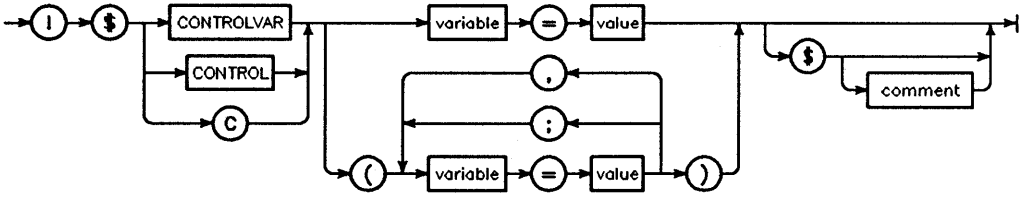
CONFIGCHECK OFF will disable checking for the existence of functions, subprograms or system binaries before allowing these calls to be made. If all associated system binaries, functions and subprograms are present at run time, checking for them is redundant and time wasting. If they are not present, however, the system will crash if CONFIGCHECK is OFF.

CONTROLVAR (CONTROL or C) Directive

This directive is used to set the values of any control variables in your program. A control variable is a variable name composed of multiple characters, with the first character being alphabetic (a-z, A-Z). The compiler uses only the first character to identify a control variable, so the first letter must be unique. The compiler will distinguish between upper and lower case letters, so there are 52 possible unique control variables. Control variables are used in conjunction with the IF and IFNOT compiler directives. Control variables are assumed to have a value of 0 (zero), unless their values are changed with this directive. The syntax is:

! **$CONTROLVAR** *variable=value* [$[*comments*]]

! **$CONTROLVAR** (*variable=value*[;/, *variable=value*] ...)



where *variable* is the name of the control variable and *value* is an integer value (positive or negative) you wish to have assigned to the variable (see "Details" below). The user can set control variables at the start of the main program, and keep their values for the entire compilation.

Default

Control variables are set to 0 (zero) at the start of compilation.

Scope

It is a global-scoped directive which retains its value across context boundaries. Note that it is the only directive which is global scoped.

Set Location

This directive is valid anywhere within a context except within the dummy MAIN program.

Details

The CONTROLVAR directive is not valid in the dummy main program generated by the compiler.

Control variables are used to control conditional compilation. They appear in the **IF ... END** directive pair. If the control variable following the IF has a non-zero value, the code between the IF and the END will be compiled; otherwise, it will not (i.e. it will be treated as comments). For example:

```
100   ! $CONTROLVAR  (Z=0;a=1) $
110   ! $IF Z$
        .
        .
150   ! $END$
160   ! $IF a$
        .
        .
210   ! $END$
```

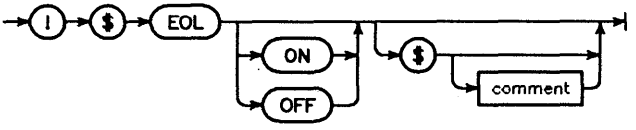
The first IF ... END segment would *not* be compiled because the control variable "Z" was set to zero by the CONTROLVAR directive. The second IF ... END segment is compiled because the control variable "a" was set to one by the CONTROLVAR directive.

With the IFNOT directive, if the control variable following the IFNOT has a value of zero, the code between the IFNOT and the END will be compiled. Otherwise it will not.

EOL Directive

This directive instructs the compiler to emit end-of-line activity code. This enables the operating system to service pending events such as key presses and interrupts. The absence of such code will cause the entire CSUB to be processed as a single line of code. In other words, events occurring during the execution of a compiled BASIC CSUB with EOL OFF will not be acknowledged until the end of the CSUB execution or an end-of-line check in the code, whichever comes first. This can be accomplished by setting EOL ON at some point or calling an interpreted or compiled SUB with EOL ON. The syntax is:

```
! $EOL [ON/OFF] [$[comments]]
```



where EOL OFF suppresses the generation of end-of-line activity code. Using EOL OFF produces much faster and smaller code than EOL ON.

Default

The default for this directive is EOL ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

Before you use this directive, you MUST read the section, "Using the EOL Directive" in the chapter "Improving Compiled Programs." Using this directive improperly can cause very serious side effects.

Errors will be processed by the operating system even with EOL OFF, but an ON *event* statement may not be processed properly unless it is compiled with EOL ON.

If a context is compiled with EOL OFF, events (such as key presses, pauses, etc.) will not be acknowledged until the next line where end-of-line checking is enabled.

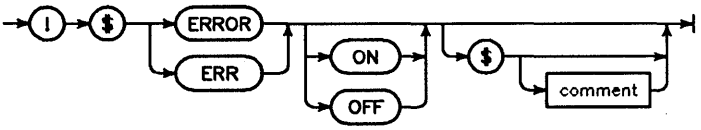
5

EOL OFF makes a section of code look (to the operating system) *and behave* like a single line of BASIC code.

ERROR (ERR) Directive

The compiler generates error messages when errors are encountered during compilation. By default, each time an error is encountered the compiler will pause and ask if you wish to stop and correct the error or continue with compilation. If you decide to continue compilation, only those subprograms containing errors will not be compiled—all the rest will. If you decide to stop and correct the error, all code generated for previously compiled subprograms will remain intact. The ERROR directive can be used to tell the compiler whether or not to give you this option. It has the syntax:

```
! $ERROR [ON/OFF] [$[comments]]
```



If ERROR OFF is specified, the compiler will assume that you want to continue compilation without correcting the error. The error message will still be displayed, but the compiler will not give you the option of immediately correcting the error.

Default

The default value for this directive is ERROR ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

IF and IFNOT Directives

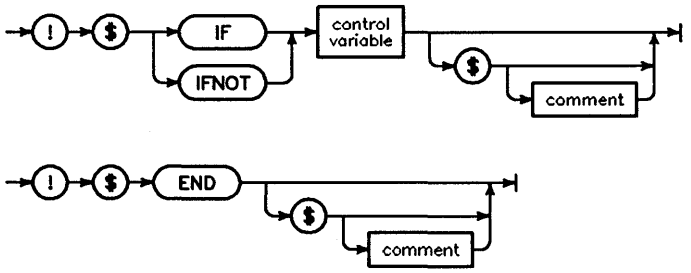
The IF and IFNOT directives are used to allow conditional compilation. They are always paired with a corresponding END directive. The text from (and including) the IF (or IFNOT) up to (and including) the END is the text affected by this directive. The syntax is:

```
120   ! $IF control variable [$[comments]]
        .
        .
180   ! $END [$[comments]]
```

or

```
120   ! $IFNOT control variable [$[comments]]
        .
        .
180   ! $END [$[comments]]
```

5



The *control variable* is a variable name composed of alphanumeric characters. The first character of a control variable name must be a unique alphabetic (a-z, A-Z) character. Only the first character of a control variable name is recognized by the compiler and all other characters are ignored. The control variable is used to decide whether or not the IF ... END segment will be included in compilation (see Details below).

Default

None

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

An IF ... END segment must be entirely contained within a single context.

Details

Control variables automatically have a value of 0 (zero) when they are encountered unless they have been reset using the CONTROLVAR directive.

Any variable name following an IF or IFNOT directive is assumed to be a control variable. The compiler can distinguish between control variables and program variables so you can use a control variable with the same name as a program variable.

When the compiler encounters the IF directive, the value of the control variable is checked. If the control variable has a nonzero value, the code between the IF and the END directives is compiled; otherwise, it is ignored. In the following sample:

```
230   ! $IF compile
          .
          .
270   ! $END$
```

the code between the IF and the END would be ignored, because the value of "compile" is zero. (We are assuming the "compile" has not appeared in a CONTROLVAR directive).

IF and IFNOT Directives

When the compiler encounters the IFNOT directive, the value of the control variable is checked. If the variable has a value of 0 (zero), the code between the IFNOT and the END directives is compiled; otherwise, it is ignored. In the following program sample:

```
100   ! $CONTROLVAR compile=1
110   ! $IFNOT compile
          .
          .
150   ! $END$
```

the text between the IFNOT and the END will be ignored by the compiler. The value of `compile` was set to 1 using the CONTROLVAR directive.

An IF ... END segment may appear anywhere within your program as long as it is entirely contained within a single context. Nested IF ... END segments are not allowed.

5 Care must be exercised when using the IF ... END construct. Block statements such as:

SELECT ... CASE ... END SELECT
WHILE ... END WHILE
LOOP ... END LOOP
IF ... ELSE ... END IF
REPEAT ... UNTIL
FOR ... NEXT

must not be broken by a conditional compilation. An error will be generated if any of the above constructs are only partially contained within an IF ... END segment. For example, if a conditional compilation causes a END, SUBEND or FNEND to be ignored, an error 1028 will be generated. The following program lines will produce this error:

```
10    ! $CONTROLVAR Z=0
20    SUB Sample
          .
100   ! $IF Z
          .
150   SUBEND
          .
190   ! $END
```

Conditional compilation is very useful for "removing" debug statements or statements that are not compilable without actually having to remove them from your program. For example, in the following program:

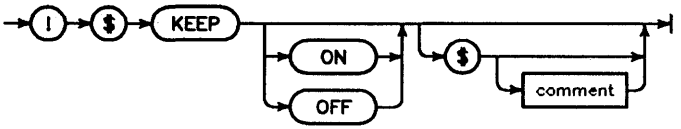
```
10   DIM Data_array(10)
20   Array_size=10
30   ! SAMPLE PROGRAM USING CONDITIONAL COMPILATION
40   ! THE FOLLOWING LINE WILL BE COMPILED:
50   PRINT "PROCESSING DATA ARRAY"
60   !$IF DEBUG
70   ! THE FOLLOWING 3 STATEMENTS WILL NOT BE COMPILED
80   PRINT "THE DATA ARRAY IS"
90   PRINT Data_array(*)
100  TRACE PAUSE
110  !$END
120  !
130  FOR I=1 TO Array_size
140  Data_array(I)=I
150  NEXT I
160  !$IF DEBUG
170  ! AGAIN, THE FOLLOWING STATEMENTS WILL NOT BE COMPILED
180  PRINT "THE RESULT IS"
190  PRINT Data_array(*)
200  TRACE PAUSE
210  !$END
220  END
```


the debugging code (lines 80-90, 180-190) would be present in the interpreted environment, as would the TRACE commands (lines 100 and 200). When your program is running correctly and you are ready to compile it, the debugging lines will be "skipped" since the value of the control variable "DEBUG" was never set to 1 (0 is the default). This is also a good way to support the usage of non-compilable commands (such as TRACE) in your program.

KEEP Directive

By default, the compiler will keep the source code along with the compiled code when it compiles a program. This feature can be turned off with the KEEP OFF directive, so the source code will be discarded (only if it is successfully compiled). The syntax is:

```
! $KEEP [ON/OFF] [$[comments]]
```



5 where KEEP OFF discards your source code as the program is compiled.

Default

The default value for this directive is KEEP ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

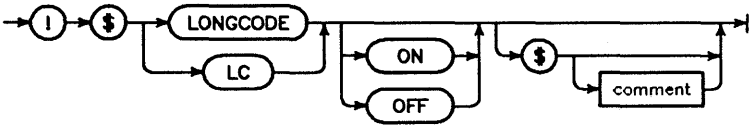
Make sure that you have saved a copy of your program on an external device before you compile it with KEEP OFF. You will not be able to recover the original source code of a program that is compiled with the KEEP OFF directive.

When you compile your program copies of both the compiled code and source code remain in memory. If you have a limited amount of memory to use, the KEEP directive is useful for discarding the unneeded source code. To discard the source code selectively, you can place the KEEP OFF directive at the beginning of each subprogram you wish to discard.

LONGCODE (LC) Directive

The LONGCODE directive permits the user to selectively enable or disable the generation of object code which uses extended addressing. This directive must be ON when forward addressing exceeds 32K bytes of code (see Details below). The syntax is:

```
! $LONGCODE [ON/OFF] [$[comments]]
```



5 where LONGCODE OFF inhibits the generation of extended addressing.

Default

The default value for this directive is LONGCODE OFF.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive *must* appear before the first executable statement of a context or it will be ignored by the compiler.

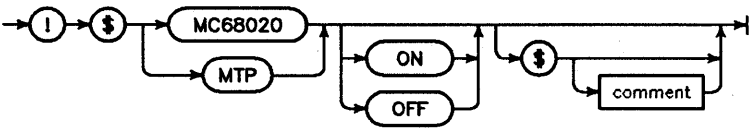
Details

You will not know prior to compilation whether or not the LONGCODE directive should be used. The compiler emits "short" code by default and informs you if the LONGCODE directive is necessary for any of the contexts. Instead of using LONGCODE, you could separate long contexts into two smaller ones. You should use LONGCODE only in those context(s) where it is necessary as this directive produces longer and slower code.

MC68020 (MTP) Directive

The default value for MC68020 is ON for computers with a MC68020/30 processor; otherwise, the default is OFF meaning that the compiler generates code that checks for the presence of the MC68020/30 processor and uses it if it is present. Compiling with MC68020 ON causes the compiler to generate code that runs on only the MC68020/30 processor. The syntax is:

```
! $MC68020 [ON/OFF] [$[comments]]
```



5

Default

The default value for this directive is MC68020 OFF on a computer without a MC68020/30 processor and ON on a computer with a MC68020/30 processor.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive must appear before the first executable statement of a context or it will be ignored by the compiler.

Details

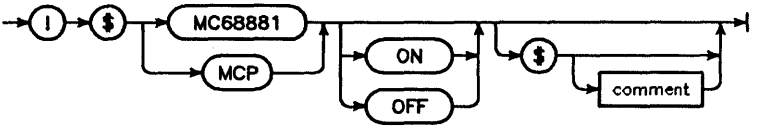
If a program (or subprogram) is compiled with MC68020 ON, the resultant code *will not* run if the MC68020/30 is not present (*a system error will occur*).

The speed increases that result from this directive will be most apparent in **compute-bound** subprograms. Code that is **I/O bound** will not yield a speed improvement when this directive is used.

MC68881 (MCP) Directive

The default value for MC68881 is ON for computers with a MC68881/82 coprocessor; otherwise, the default is OFF meaning that the compiler generates code that checks for the presence of the MC68881/82 coprocessor and uses it if it is present. Compiling with MC68881 ON causes the compiler to generate code that runs on only the MC68881/82 coprocessor. The syntax is:

```
! $MC68881 [ON/OFF] [$[comments]]
```



Default

The default value for this directive is MC68881 OFF on a computer without a MC68881/82 coprocessor and ON on a computer with a MC68881/82 coprocessor.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive must appear before the first executable statement of a context or it will be ignored by the compiler.

Details

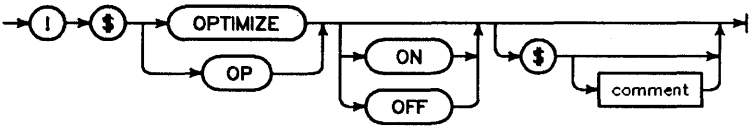
If a program (or subprogram) is compiled with MC68881 ON, the resultant code *will not* run if the MC68881/82 is not present (*a system error will occur*).

The speed increases that result from this directive will be most apparent in compute-bound code. Code that is I/O bound will not yield a speed improvement with this directive.

OPTIMIZE (OP) Directive

When the compiler is invoked, the default value of this directive is OFF—which means that your program is optimized for space rather than speed. In other words, the compiler may be sacrificing a small amount of speed to reduce the size of your compiled program. This feature can be controlled using the OPTIMIZE directive. The syntax is:

```
! $OPTIMIZE [ON/OFF] [$[comments]]
```



where OPTIMIZE OFF optimizes for space, and OPTIMIZE ON optimizes for speed.

Default

The default value for this directive is OPTIMIZE OFF (space optimization).

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive must appear before the first executable statement of a context or it will be ignored by the compiler.

Details

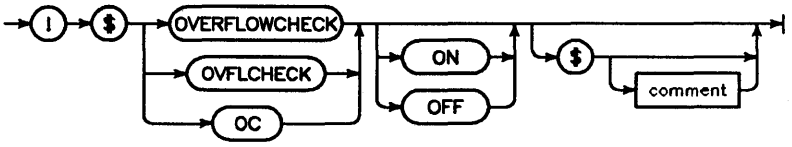
To get the smallest possible code, possibly at the expense of some speed, you should use OPTIMIZE OFF. To get the fastest code, which may increase the size of your compiled program, use OPTIMIZE ON. To get the best compromise of space and speed, you should use BEST and OPTIMIZE OFF in the options list of your compiler invocation command.

Space optimization will affect only your I/O and graphics speeds. Computations, logical operations, and looping are not affected by space optimization. For more details, see the chapter "Improving Compiled Programs."

OVERFLOWCHECK (OVFLCHECK or OC) Directive

The compiler will normally emit code to check for integer and string overflow after each integer and string computation. This capability can be enabled or disabled using the OVERFLOWCHECK directive. The syntax is:

```
! $ OVERFLOWCHECK [ON/OFF] [$[comments]]
```



where OVERFLOWCHECK OFF suppresses the check for integer and string overflow.

5

Default

The default value for this directive is OVERFLOWCHECK ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

Integer overflow will occur if computations cause an integer variable to be greater than 32767 (the largest allowable integer value) and smaller than −32768 (the smallest allowable integer value). The compiler emits code to check for overflow after each integer computation. This increases memory requirements and decreases speed. You may wish to disable such checks when you are confident that overflow is not possible.

OVERFLOWCHECK OFF will also disable checking for string overflow in the following cases:

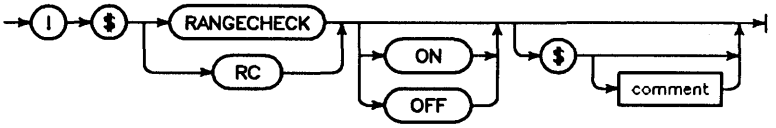
string_var$=*string_var*$

string_var$=*string_constant*

RANGECHECK (RC) Directive

The compiler automatically emits code to check for out of range values (for example, in the indexing of array subscripts) before they are used. This feature is controlled using the RANGECHECK directive. The syntax is:

```
! $RANGECHECK [ON/OFF] [$[comments]]
```



where RANGECHECK OFF suppresses generation of range checking code.

5

Default

The default value for this directive is RANGECHECK ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

RANGECHECK OFF will suppress the following checks:

- tests for missing OPTIONAL parameters when parameters are accessed
- testing for a division by zero (INTEGER) with DIV, MOD and MODULO
- testing for a RETURN statement that does have a corresponding GOSUB to return to
- indexing of array subscripts

The compiler emits code to check for out of range values in applicable cases which will increase memory requirements and decrease speed. You may wish to disable such checks when you are confident that your program works.

When the RANGECHECK compiler directive is ON, range checking for local static arrays is done differently in interpreted BASIC and compiled BASIC. In
5 interpreted BASIC, each separate index of an array is checked to make sure that it is in bounds. In compiled BASIC, only the overall index is checked. As an example, take a 3-dimensional array called `Array_1` which has the dimensions (assuming OPTION BASE 1):

$$6 \times 3 \times 2$$

This array has a total of 36 elements. If you try to access the element (2,7,1) in this array, it will be rejected as out of range by interpreted BASIC because the second dimension has only three elements. If you tried to access this same element in compiled BASIC, the error would go undetected. This is because the compiler does range checking on the overall array index, and:

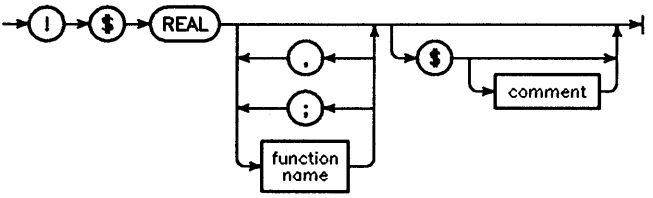
$$(2 \times 7 \times 1) = 14$$

which is *not* more than the overall index of 36 elements representing the entire array.

REAL Directive

By default, the value returned by any function in your program will be of type REAL. The COMPLEX directive allows you to specify functions that will return a COMPLEX value. If this directive has been used to specify the type of a function, REAL can be used to change the type back to REAL. REAL can also be used to declare a function that will be returning a real value, though this is redundant since REAL is the default. The syntax is:

!$REAL *function name* [;/, *function name*] ...



Each *function name* is the name of a function that is to return a REAL value.

Default

By default, all functions return a REAL value.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

When using this directive, the first two letters of any specified function name must be FN or warning 21 (Improper ID on Declaration Directive) will be generated at compile time. If a specified function name ends with the character $, warning 21 will occur at compile time. Warning 21 will be generated any time a specified function is not found in the symbol table (i.e., if a function is declared, but it is not called within the context). If warning 21 is ignored (if you continue compilation without correcting the error) the function in question will be assumed type REAL.

The rules for this directive are not as strict as those for the COMPLEX directive since the default type of any function is already REAL. To follow the rules exactly, the directive should appear:

- once in every context (subprogram) that calls the specified function
- before the function is called
- within the function itself

5

For example:

```
10   REAL X
20   !$REAL FNOne
30   X=FNOne(3)
40   END
50   !
60   DEF FNOne(X)
70   !$REAL FNOne,FNTwo
80   RETURN ((X/4)+FNTwo(2*X))
90   FNEND
100  !
110  DEF FNTwo(X)
120  !$REAL FNTwo
130  RETURN X-3
140  FNEND
```

If a function was declared to be of a given type (using one of the directives REAL or COMPLEX) and the function returns a value of a different type, a run-time error will occur stating that the value returned is not of the correct type.

The directive in the calling context (as in lines 20 and 70 in the previous program) indicates what type the calling context should expect to receive

(the expected type). The directive in the function itself (as in lines 70 and 120 in the previous program) indicates what type should be returned by the function (the returned type). The value to be returned from a function will be converted to the returned type (if necessary) before the value is returned. When the value is returned, the returned and expected types are compared. If the returned and expected types are different, error 19 (Improper value or value out of range) will be generated.

In this program sample:

```
10   !$REAL FNOne
20   INTEGER X
30   X=FNOne
40   END
50   !
60   DEF FNOne
70   !$REAL FNOne
80   RETURN 12.7
90   FNEND
```


The returned and expected types are both REAL. The value of the function FNOne (12.7, as indicated in line 80) is already a REAL value, so no conversion will take place before it is returned. After it is returned, the value will be converted to INTEGER (the value will be rounded to 13), since X is an integer value. The directives in the program sample above are unnecessary and redundant because REAL is the default condition anyway. The REAL directive is most useful when you are toggling the type of a function, as described below.

REAL Directive

The COMPLEX directive can be used to declare the type of a function. The REAL and COMPLEX directives can be used in conjunction to toggle the type returned by a function. A sample program illustrating this ability is given below:

```
10   COMPLEX C
20   REAL R
       .
100  !$COMPLEX FNOne
110  C=FNOne(1)
120  !$REAL FNOne
130  R=FNOne(2)
140  END
150  !
160  DEF FNOne(Parm)
170  COMPLEX C
180  REAL R
190  IF Parm=1 THEN
200     !$COMPLEX FNOne
       .
       .
240     RETURN C
250  ELSE
270     !$REAL FNOne
       .
310     RETURN R
320  END IF
330  FNEND
```

5

Keep in mind that any time the returned and expected types differ, an error will be generated. So, the following code will generate an error:

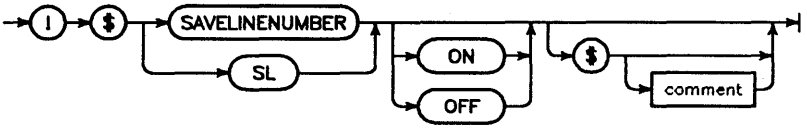
```
10   !$COMPLEX FNOne
20   R=FNOne
30   END
40   !
50   DEF FNOne
60   !$REAL FNOne
70   RETURN 3
80   FNEND
```

Line 10 indicates that the expected type of FNOne is COMPLEX, and line 50 indicates that the returned type of FNOne is REAL.

SAVELINENUMBER (SL) Directive

This directive can be used to indicate whether or not the compiler should emit code to save each line number in a program when the line is executed. If a run-time error occurs in a context with SAVELINENUMBER OFF, the line number reported with the error will be the first line of the context rather than the line where the error actually occurred. The syntax is:

! $SAVELINENUMBER [ON/OFF] [$[*comments*]]



where SAVELINENUMBER OFF suppresses the saving of line numbers.

Default

The default value for this directive is SAVELINENUMBER OFF.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

Details

The SAVELINENUMBER directive will have the most impact when run-time errors are encountered. When you encounter an error at run time, remember these four rules:

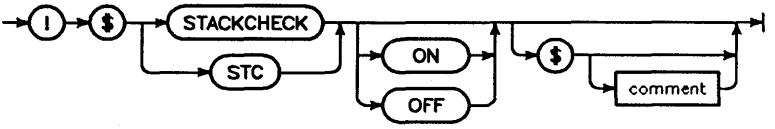
- If the subprogram containing the error was compiled with SAVELINENUMBER OFF (or SL OFF), the line number reported with the error will be the line number of the CSUB in which the error occurred.
- If the subprogram containing the error was compiled with SAVELINENUMBER ON and KEEP OFF, the compiler assumes that you are keeping a copy of your source code elsewhere. Therefore, the line number that is reported with the error message will correspond to the line number of your original BASIC program (before the program was compiled).
- If the subprogram containing the error was compiled with SAVELINENUMBER ON and KEEP ON, the line number reported with the error will be the line number within the SUB following the CSUB being executed (the source code form of the CSUB).
- Separately compiled subprograms with SL ON will report the wrong line number if a run-time error occurs in the compiled subprogram. For example, you may compile a subprogram using COMPILE: SL with line numbers ranging from 1 to 10 and store it in a file. First make sure that this subprogram will cause a run-time error such as division by zero. Perform a LOADSUB of the compiled CSUB into another program which contains lines with line numbers greater than 10. The loaded CSUB will end up with a line number greater than 10. Running the program which calls the compiled CSUB will generate a run-time error which points to a line number less than 10. This line number should be in the CSUB, but when you EDIT the program you will be looking at the wrong line. This is due to the fact that when the CSUB was called it told the BASIC Language System that its line numbers are in the range of 1 to 10 since it was compiled that way. The CSUB had no knowledge of where it was loaded.

  This problem can be avoided by not separately compiling subprograms with SL ON. So when an error occurs the system points to the CSUB line.

STACKCHECK (STC) Directive

During program execution, space may be needed for temporary variables. The compiler automatically emits code to check for the availability of such space before it is allocated. The STACKCHECK directive can be used to control this feature of the compiler. The syntax is:

```
! $STACKCHECK [ON/OFF] [$[comments]]
```



where STACKCHECK OFF inhibits the check for available space and suppresses error messages for memory overflow.

Default

The default value for this directive is STACKCHECK ON.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere within a context.

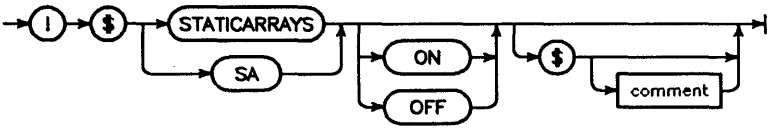
Details

STACKCHECK ON causes the compiler to emit extra code, which decreases speed and increases the memory requirements. You may wish to disable such checks when you are confident that memory overflow cannot occur.

STATICARRAYS (SA) Directive

The STATICARRAYS directive affects arrays declared with DIM, INTEGER, REAL, or COMPLEX statements (in other words, all arrays *except* those declared with an ALLOCATE or COM statement, and those that appear as parameters). If you do not use the REDIM statement on any of your local arrays within a context, you should take advantage of the speed increases that are produced with the STATICARRAYS directive. The syntax is:

```
! $STATICARRAYS [ON/OFF] [$[comments]]
```



5

where STATICARRAYS OFF allows the use of REDIM statement.

Default

The default value for this directive is STATICARRAYS OFF.

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

STATICARRAYS ON must appear before the first executable statement of the context or it will be ignored by the compiler.

Details

This directive tells the compiler that the local array bounds are static (i.e., do not change during program execution by using the REDIM statement or a copy subarray does not cause an auto-REDIM). The compiler will then perform most of the array indexing calculations during compile time, saving program execution time. The speed of execution will be greatly enhanced when multi-dimensional arrays are being used.

If you set STATICARRAYS ON, and a REDIM is encountered, warning 20 will be generated at compile time. This warning can be ignored if the REDIM is for:

- an array in COM
- a single-dimensional array
- a multi-dimensional array whose number of dimensions is changed

You *must* pay attention to this warning if you are:

- REDIMming a local multi-dimensional array
- REDIMming some dimension other than the last

When the STATICARRAYS compiler directive is ON, range checking for local static arrays is done differently in interpreted BASIC and compiled BASIC. In interpreted BASIC, each separate index of an array is checked to make sure that it is in bounds. In compiled BASIC, only the overall index is checked. As an example, take a 3-dimensional array called `Array_1` which has the dimensions:

$6 \times 3 \times 2$

This array has a total of 36 elements. If you try to access the element (2,7,1) in this array, it will be rejected as out of range by interpreted BASIC because the second dimension has only three elements. If you tried to access this same element in compiled BASIC, the error would go undetected. This is because the compiler does range checking on the overall array index, and:

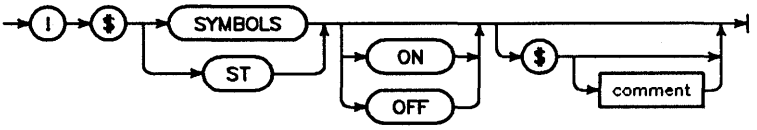
$(2 \times 7 \times 1) = 14$

which is *not* more than the overall index of 36 elements representing the entire array.

SYMBOLS (ST) Directive

This directive will dump a symbol table (at compile time) for the context in which it appears. The syntax is:

```
! $SYMBOLS [ON/OFF] [$[comments]]
```



Default

By default, a symbol table is not dumped.

5

Scope

It is a context-scoped directive which resets to a default value at the start of each context.

Set Location

This directive is valid anywhere that it appears within a context.

Details

The symbol table includes the name of the symbol, its type (REAL, INTEGER, etc.), what it is (variable, SUB, array, etc.), and its scope (local or global). If the directives REAL or COMPLEX are used in a program, the corresponding type will be in the symbol table for that function. If you use directives to toggle the type of a function (see the sections "REAL Directive" and "COMPLEX Directive"), the type declaration will be used in the symbol table. For example:

```
10   !$ST
20   !$COMPLEX FNComplex
30   !$REAL FNReal
40   COMPLEX Complex_num
50   REAL Real_num
60   Complex_num=FNComplex
70   Real_num=FNReal
80   END
```


will produce the following symbol table:

```
-------  SYMBOL TABLE DUMP for Mainsub

    NAME           TYPE      KIND      SCOPE
---------------  -------  --------  ----------------
Complex_num      COMPLEX  VARIABLE  Local
Real_num         REAL     VARIABLE  Local
FNComplex        COMPLEX  SUB       Global
FNReal           REAL     SUB       Global
Mainsub          REAL     SUB       Global
-------------------------------------------------------

Number of Entries in Table =11
Number of Used Entries   =7
```

For this program:

```
10   !$ST
20   !$COMPLEX FNSome_function
30   COMPLEX Complex_num
40   REAL Real_num
50   Complex_num=FNSome_function
60   !$REAL FNSome_function
70   Real_num=FNSome_function
80   END
```

the symbol table entry for FNSome_function will be listed as type REAL, as that is the last type the function was set to.

6

Improving Compiled Programs

Previous chapters in this manual explained how the BASIC compiler works and how to use it. This chapter provides information about compiled programs, and offers tips on how to write programs that compile and run efficiently. Topics covered are:

- Storing Your Program
- Compiled Program Compatibility
- Interacting with Compiled Programs
- Writing Efficient Programs
- Optimizing Your Program
- Using the EOL Directive
- Using the DS Command

Storing Your Program

Compiled programs must never be SAVEd or RE-SAVEd. The SAVE command converts a file into an ASCII file. Trying to SAVE your program would cause only the readable part of the compiled program (the CSUB and CDEF statements) to be copied into the ASCII file, not the machine code which is really the compiled program. Subsequent attempts to GET the SAVEd file will fail, since the BASIC Operating System cannot syntax CSUB or CDEF statements.

Note *Always* use STORE or RE-STORE to move a compiled program from the computer memory to a disk file.

Compiled Program Compatibility

You can write programs consisting of both interpreted and compiled BASIC. It is perfectly acceptable to have a BASIC program which consists of a interpreted BASIC main program which calls compiled functions and subprograms, as well as interpreted BASIC functions and subprograms and user-written CSUBs. Compiled SUBS follow the same compatibility rules as interpreted subprograms. These are outlined below.

- SUBs and CSUBs may call user-written CSUBs generated from Pascal, Assembly, or FORTRAN using the BASIC CSUB utility.
- SUBs and CSUBs may be called by interpreted BASIC routines.
- Compiled and interpreted BASIC routines can communicate through parameters or COMmon statements.
- SUBs and CSUBs may call interpreted BASIC routines or those BASIC routines compiled with the BASIC Compiler.

Interacting with Compiled Programs

6

When running a compiled program, the keyboard works much the same way as it does when running an interpreted program. However, the following "rules" must be kept in mind:

- Pressing the [PAUSE] key (or [Stop] key on an ITF Keyboard) will work only if the context was compiled with EOL ON. Otherwise, the entire context will be executed before the pause is acknowledged.
- The following are ok: [RUN] key (or [f3] key on the System menu of an ITF Keyboard), RUN keyword without a line number or label, [CONTINUE] key (or [f2] key on the System menu of an ITF Keyboard), CONT keyword without a line number or label. If you use the CONT or RUN keywords, you may not specify a line number or label.
- All other command keys such as [EXECUTE], [SHIFT]-[CLR I/O], etc. will work only if the EOL ON compiler directive was in effect when your program was compiled. This directive assures that the interrupts from these keys will be serviced.

- The PAUSE statement may be used in your program, even if EOL OFF was in effect. It will work in the same manner as it would in an interpreted program.
- The STEP key (which is in the System menu on an ITF keyboard) cannot be used to step into a CSUB. Using the STEP key will have the same effect as pressing the (CONTINUE) key.

With the exception of the keys or commands mentioned above, all other commands can be executed while a compiled program is running.

Writing Efficient Programs

Writing efficient programs entails establishing a balance between code size and program speed. A program that runs very quickly may not be worthwhile if it occupies 8 megabytes of memory. On the other hand, a very compact program may be useless if it takes all day to execute.

Before you can begin making decisions about the efficiency of a program, you need to understand some differences between compilers and interpreters. This section covers these differences as well as the following topics:

- Compute-bound code
- I/O-bound code

Interpreted vs. Compiled Code

In an interpreted BASIC program, each BASIC line is represented as a series of "intermediate codes" (referred to as I-codes):

Line Header	I-code	I-code	...	EOL

Each I-code represents an activity that must be carried out by the interpreter. The EOL represents the end-of-line I-code, and it occupies 1 byte. The "line header" occupies 6-8 bytes, depending upon whether the line is labeled.

An interpreted BASIC program is fairly compact as far as code size goes. Its instructions do not occupy a lot of space. The "expense" of interpreted BASIC occurs at run time. Each I-code has to be fetched and recognized by the interpreter. After this, an interpretation of the I-code semantics is made, then the I-code instruction is carried out.

In a compiled program, every BASIC line is represented directly in machine code. Machine code is bit patterns that can be understood by the computer. It is a direct and exact representation of the semantics of the original BASIC line. Since the statements have been "expanded" into machine code, a compiled program will be larger than an interpreted one.

Even though the code is larger, a compiled program usually executes much faster than the interpreted version because the "overhead" of fetching and recognizing instructions at run time is eliminated.

Compute-Bound Code

Compute-bound code consists largely of arithmetic expressions, string expressions, logical expressions, and looping statements. When code is compute bound, it will execute much faster when compiled.

6

In an interpreted BASIC program, the statement "A = B + C" is represented as:

Line Header	12	13	14	232	144	EOL

where the I-code:

12 means push the address of A onto the BASIC stack

13 means push the address of B onto the BASIC stack

14 means push the address of C onto the BASIC stack

232 means add B and C, and push the result on the stack

144 means store the value on the top of the stack in A

These I-codes occupy about 5 bytes of storage. The interpreter will execute at least 200 machine code instructions to carry out the operation.

In a compiled program, the statement "A = B + C" could be represented in machine code as:

```
move.w -26(A4), D0
add.w -28(A4), D0
move.w D0. -24(A4)
```

This machine code occupies 12 bytes. However, this operation is executed by 3 machine code instructions.

For this assignment statement, the "code expansion ratio" (the amount the code expands when compiled) is 12/5, or 2.4. This means that the compiled code is about 2.4 times the size of the interpreted code, but it should be about 40 to 70 times faster.

On the average, the code expansion ratio for compute-bound code will be between 1.5 and 2.0. The speed increases for compute-bound code, however, are very great. Computations with real numbers can be expected to run 1.15 to 5.0 times faster than interpreted code; computations with integers will run between 5 and 10 times faster, with some operations as great as 40 times faster.

The speed of programs that make extensive use of system functions such as SIN, COS, or SQR may be improved approximately 10% by compilation because these functions are implemented in firmware even when the program is compiled.

I/O-Bound Code

I/O-bound code is code that consists mainly of I/O or graphics operations such as PRINT, READ, or MOVE. The speed increases for I/O-bound code are not as great as for compute-bound code.

In interpreted BASIC, the statement PRINT A is represented as:

Line Header	155	0	164	EOL

where the I-code:

155 initializes the print operation

0 pushes the address of A onto the BASIC stack

164 performs an I/O operation

These I-codes occupy about 3 bytes.

In a compiled program, the statement "PRINT A" could be represented in machine code as:

```
lea         global_pntr, A5
move.l      A2, B_TOS
jsr         Print_init
movea.l     Valptr, A4
movea.l     B_TOS, A2
movem.l     -8(A4), D0_D1
movem.l     D0_D1, -(A2)
clr.w       -(A2)
movea.w     #443, A3
move.l      A3, IPC
movea.l     36(A4), A0
jsr         Goto_OS
```

6 This machine code occupies 64 bytes.

The code expansion ratio for this example is 21. The form shown above is one of the worst case examples for compiled BASIC. On the average, the code expansion ratio for I/O-bound code will be between 2.5 and 4.0. If you had a print statement with 20 items listed after it, the code expansion ratio would be much smaller. Note that the compiled program has no additional overhead after the first item is printed. With an interpreted program, the overhead would continue for each item in the list to be printed.

On the average, the speed increases that can be expected for I/O-bound code are from 20% to 50% (i.e., 1.2 to 1.5 times faster), depending upon the device and operating system routines that you are using.

The BASIC Compiler will use the above code generation method when the program is being optimized for speed (OPTIMIZE ON). When optimizing for space (OPTIMIZE OFF), the compiler uses a completely different method of code generation that reduces the code size considerably, while sacrificing a small amount of speed for I/O statements (not more than 10%).

Optimizing Your Program

There are a number of programming techniques that can affect code size and speed. This section explains what effect different compiler directives and programming techniques will have on the speed and size of your compiled program.

Compiler Directives

Compiler directives are special commands that may be used in your program to enable or disable certain features of the compiled environment. A full explanation of each directive described in this section can be found in the chapter "Compiler Directives."

The directives listed in this section are the ones that could have an effect on the size or speed of your program. Note that the ERROR and SYMBOLS directives will never have an effect on the size or speed of your program, so they are not listed. Also note that the IF, IFNOT and CONTROLVAR directives will only affect your program size and speed when blocks of code are not compiled due to conditional compilation. If you have blocks of code that are not compiled (due to conditional compilation), your code will be smaller and faster. The same thing would happen in interpreted BASIC if you deleted sections of your program.

CONFIGCHECK

With CONFIGCHECK ON, the compiler emits code (at run time) to check for the presence of user subprograms and system binaries before they are used. Since the code emitted to make these checks takes up space, and the actual checking takes time, your program will be both smaller and faster if you set the CONFIGCHECK directive to OFF. If you have very few calls to user SUBs or system binaries, the speed and size differences will be minimal.

EOL

By default, the compiler generates end-of-line activity code for each line in a program. This allows ON *events* to be processed properly. End-of-line activity code adds a certain amount of code to each line of a program, so turning EOL OFF can decrease your code size considerably. Note that the longer your program is, the more EOL OFF will help. EOL OFF will also increase the

speed, as the line pointers will not have to be moved after the execution of each line. Leaving EOL OFF is ok if you *do not* mind the increased amount of time it takes to service interrupts. However, if this is a problem you may want to use this directive only in subprograms where the time delay for servicing interrupts is minimal. Before you use the EOL directive, be sure to read the section on "Using the EOL Directive," later in this chapter.

KEEP

The KEEP directive is used to specify whether or not the source code should be kept in memory with the compiled code. Using KEEP ON will have no effect on the speed of your compiled code. Using KEEP ON will, however, greatly increase the size of your program, since both the compiled and interpreted versions of the program are being kept together.

LONGCODE

The LONGCODE directive must be used when forward addressing exceeds 32K bytes of code. You should not use it unless necessary because this directive will make your code larger and slightly slower.

MC68020 and MC68881

6

The default value for the MC68020 directive is OFF on a computer without a MC68020/30 processor and ON on a computer with a MC68020/30 processor. The default value for the MC68881 directive is OFF on a computer without a MC68881/82 coprocessor and ON on a computer with a MC68881/82 coprocessor.

When these directives are OFF, the compiler generates code that checks for the presence of the MC68020/30 processor and MC68881/82 coprocessor and uses them if they are present. When these directives are ON, the compiler will emit code that is dependent upon the processor and/or coprocessor and uses them without checking to see if they are there. With these directives ON, your program will be smaller (since the compiler does not emit the "check for" code), and faster (since the check does not take place). If you use these directives, make sure that the processor and/or coprocessor is installed before you run your program. The difference will be significant in compute-bound code where the (co)processor is frequently used.

OPTIMIZE

OPTIMIZE ON will cause the compiler to produce the fastest code. OPTIMIZE OFF will cause the compiler to generate the smallest code. Space optimization will affect only the speed of I/O-bound code (by at most 10%). The speed of compute-bound code will not be affected by space optimization.

If you want the compiler to produce code that is the most efficient compromise between size and speed, you should use:

```
BEST, OP OFF
```

in the options list following the compiler invocation command. A compiler invocation command such as:

```
COMPILE SUBA TO END:  BEST, OP OFF
```

will do the trick.

OVERFLOWCHECK

With OVERFLOWCHECK ON, the compiler will emit code to check (at run time) for integer or string overflow after each integer or string operation. Since this causes extra code to be generated and these checks take time, OVERFLOWCHECK OFF will make your program smaller and faster. The difference will be most obvious in integer compute-bound code where overflow checking frequently occurs.

RANGECHECK

With RANGECHECK ON, the compiler will emit code to check (at run time) for out-of-range values (for example, when indexing arrays) before they are used. Since extra code is generated to make the check and the checking takes time, programs with range checking enabled will be slower and larger than those without.

SAVELINENUMBER

By default, the compiler will *not* emit code to save the number of each line (at run time) as it is executed. The SAVELINENUMBER ON directive can be used to tell the compiler to emit code to save the line numbers. Since this extra code is generated, and since the actual saving of the line number takes

time, programs with the SAVELINENUMBER ON directive will be larger and slower than programs that do not save the line numbers.

STACKCHECK

During program execution, space is needed for temporary variables. With STACKCHECK ON, the compiler will emit code to check for the availability of such space before it is allocated. A program compiled with STACKCHECK OFF will be faster and smaller than a program compiled with STACKCHECK ON since the code to perform the check takes up space and the actual checking takes time.

STATICARRAYS

Static arrays are local (non-parameter) arrays that are not declared using the ALLOCATE statement and never appear in a REDIM statement. If you do not use the REDIM statement in any of your local arrays, the compiler will be able to perform most of the array indexing operations at compile time. If you use STATICARRAYS ON in your program, the compiler will assume that all local arrays are static. Since array indexing operations are done only once, at compile time, setting this directive to ON will produce faster, smaller code. The speed of execution will be greatly enhanced when you are using multi-dimensional arrays.

Looping Control

Loop control statements such as REPEAT, WHILE, and LOOP will run much faster if you use local integer variables as controls. In a compiled program, the FOR ... NEXT statement has been highly optimized for the following cases:

- Integer loop counter, constant step, constant end
- Integer loop counter, constant step
- Integer loop counter, constant end
- Integer loop counter

Using Integers

All integer calculations are significantly faster than real number calculations. Hence, using integers whenever possible will increase the speed of your programs. Also, use local integer variables whenever you can.

The code generated for local integer variables is extremely fast. If you have to pass a value as a parameter, your code will be fastest if you assign it to a local variable and use the local variable from that point on. If you have to return the new value to the calling subprogram, reassign the value before exiting.

Arithmetic Expressions

You can increase the efficiency of your program if you take the time to "optimize" the structure of arithmetic expressions (those using +, −, *, /, DIV and MOD) in your program. "Optimization" of arithmetic expressions entails effective mixing of constants and variables within your expressions. Your compiled program will be faster if you let the compiler do most of the work. Any computations that can be done at compile time rather than at run time will increase the efficiency of your program.

A variable is a named data element whose value can change throughout the course of your program. A constant is a number whose value is always the same.

When the compiler comes across an arithmetic expression in your program, it will group together as many constants as possible, perform the desired operations, and emit code for the result. For example, if the compiler comes across the following line:

```
A = 1 + 2.5 + 3 + 16 + 22.7 + (198/3)
```

it will emit code for: `A = 111.2`

So, any time an expression is composed of constant operations, these will be operated on at compile time rather than at run time. Keep in mind that MAXREAL, MINREAL, PRT, PI, KBD, and CRT are treated as constants.

BEST Command

If the BEST command appears in the options list of the compiler invocation command, the compiler directives will be set to values that promote the production of the fastest and most compact code. The values will be set to:

```
CONFIGCHECK OFF        STATICARRAYS ON

EOL OFF                LONGCODE OFF

OVERFLOWCHECK OFF      SAVELINENUMBER OFF

OPTIMIZE ON            STACKCHECK OFF

RANGECHECK OFF
```

BEST will produce code that is the fastest possible and is fairly small. However, to produce the smallest possible code that is fairly fast use the following command:

```
COMPILE:  BEST, OP OFF
```

Be aware that BEST will set EOL OFF and STATICARRAYS ON. These two directives (in particular) may cause problems in your program depending upon the structure of your code. If your program does a lot of event processing, you should probably have EOL ON. Read the section, "Using the EOL Directive," for more information on this. If all local arrays in your program are static, the STATICARRAYS ON directive will not cause any problems. For more information on the STATICARRAYS directive, turn to the chapter "Compiler Directives."

Overall Program Efficiency

To optimize the efficiency of your program, you may need to compromise between speed and code size. As evidenced by the discussion on compute-bound and I/O-bound code, the effect of the compiler is much greater for computational tasks than for I/O tasks. When you are writing your program, you would be wise to put all of your computations (or as many as possible) in one or more subprograms or functions and keep them separate from those subprograms and functions that contain a great amount of I/O and graphics interactions.

In some instances, where memory is a scarce resource, it may be wise to compile your subprograms and functions that are compute bound, and leave the I/O-bound subprograms and functions in interpreted BASIC. This would keep your code file relatively small, and you would not be losing too much speed. The only thing that you really lose in such a situation is the security provided by the compiler. Compiled programs are much harder to gain access to than interpreted ones.

Using the EOL Directive

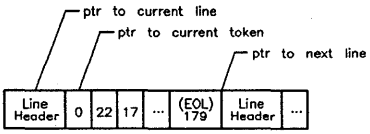
The EOL directive can be used to control when and where end-of-line activity code is to be emitted. EOL can be toggled ON and OFF throughout your program, but using EOL improperly can cause some very serious problems with the execution of your program. A compiled program using EOL haphazardly may run differently from its interpreted counterpart. If you have a program that toggles the value of EOL and you encounter problems, turn EOL ON throughout the entire program. This may very well help.

EOL is described thoroughly in this section to help you avoid any mishaps with its use. We urge you to read this section very carefully.

Before you can understand EOL checking in a compiled environment, you must understand how event processing occurs in an interpreted environment.

Interpretive Event Processing

In interpreted BASIC, each line contains a line header, the contents of the statement, and an end-of-line token (EOL token). The line header contains information about the line number, how far the line is indented (for the Editor), where the previous line is, and where the next line is. At the end of each line, the EOL code will check to see if any events occurred while the line was executing. If no events occurred, then the EOL code will establish pointers to the next line, and execution will continue. The diagram here shows an interpreted BASIC line:



The "ptr to current token" is a pointer that moves through the line as it is executing, pointing to the token (I-code) that is currently being executed. The "ptr to current line" and "ptr to next line" are pointers that keep track of where the current and next lines begin.

ON ERROR Events

When an error occurs during the execution of a line, it will be acknowledged and processed immediately, without waiting for the line to finish executing. If a statement of the form:

```
ON ERROR GOSUB location
```

was established in the same context where the error occurred, or:

```
ON ERROR CALL subprogram name
```

was established anywhere in the program, the location of the line where the error occurred will be saved. After the specified **subroutine** (in the case of GOSUB) or subprogram (in the case of CALL) has been executed control will return to the line where the error occurred (in the case of a normal subprogram exit using RETURN or SUBEND) or to the line following the line that caused the error (when the subprogram was exited with ERROR RETURN or ERROR SUBEXIT). Program execution will continue from there.

With statements of the form:

```
ON ERROR GOTO location
```

or:

```
ON ERROR RECOVER location
```

the location of the line where the error occurred does not get saved. Program control transfers to the location specified by the ON ERROR statement and program execution will continue from there.

ON TIMEOUT and ON END Events

ON TIMEOUT and ON END events are processed as soon as they occur because they normally generate an error when they occur. If a statement of the form:

```
ON TIMEOUT device selector GOSUB location
ON END io_path GOSUB location
```

was established in the same context where the error occurred, or:

```
ON TIMEOUT device selector CALL subprogram name
ON END io_path CALL subprogram name
```

was established anywhere in the program, the location of the line where the timeout or end occurred will be saved. After the specified subroutine (in the case of GOSUB) or subprogram (in the case of CALL) has been executed, control will return to the line immediately following the line where the timeout or end occurred. Program execution will continue from there.

With statements of the form:

```
ON TIMEOUT device selector GOTO location
ON END io_path GOTO location
```

or:

6

```
ON TIMEOUT device selector RECOVER location
ON END io_path RECOVER location
```

the location of the line where the timeout or end occurred does not get saved. Program control transfers to the location specified by the ON *event* statement, and program execution will continue from there.

All Other Events

For all other events, their occurrence will be acknowledged by the operating system; they will not be immediately processed. When the interpreter reaches the EOL token, it will check to see if an event occurred during execution of that line. If so, the event is processed. For statements of the form:

```
ON event GOSUB subprogram name
ON event CALL subprogram name
```

the event will be processed, and control will return to the line following the line where the event occurred.

For statements of the form:

```
ON event RECOVER location
ON event GOTO location
```

control will pass to the specified location when the event occurs, and execution will continue there.

Compiled Events Processing

When a SUB is compiled, the compiler creates a dummy line header for each CSUB. This dummy line header contains two pointers:

- one to the first executable line in the CSUB.
- one to the SUBEND statement.

The pointer to the first line of the CSUB is established as the "current line" pointer. The pointer to the SUBEND statement is the "next line" pointer.

EOL ON

If the SUB was compiled with EOL ON, pointers are established (at run time) that point to the actual "current line" and the actual "next line," before the line is executed. For each line of the program, the compiler generates:

- special "beginning of line" code
- the machine code for the line itself
- special end-of-line code

After each line has executed, any event that occurred will be processed, and program execution will resume at the next line. In other words, the CSUB will behave exactly like the corresponding interpreted version.

EOL OFF

If the SUB was compiled with EOL OFF, pointers are *not* established to the actual "current" and "next" lines. The pointers remain pointing to the first line of the CSUB (as the "current line" pointer) and the SUBEND (as the "next line" pointer). No EOL activity is performed until after the SUBEND statement.

ON ERROR Events (with EOL OFF)

The ON ERROR statement should be the first statement of your subprogram or subroutine.

Remember, errors are processed as soon as they occur, rather than waiting for the end of the line. If a statement of the form:

```
ON ERROR GOSUB location
```

was established in the same context where the error occurred, or:

```
ON ERROR CALL subprogram name
```

was established *anywhere* in the program, the "current line" pointer will be saved. After the specified subroutine (in the case of a GOSUB) or subprogram (in the case of CALL) has been executed, control will return to the line pointed to by the "current line" pointer. In the case of a CSUB compiled with EOL OFF, the "current line" pointer is still pointing to the first executable line of the CSUB. Therefore, after the specified subroutine or subprogram is executed, control will return to the beginning of the CSUB in which the error occurred, rather than to the actual line where the error occurred. The entire CSUB will be executed again, no matter where the error occurred.

With statements of the form:

6

```
ON ERROR GOTO location
```

or:

```
ON ERROR RECOVER location
```

the location of the line where the error occurred does not get saved. Program control transfers to the location specified by the ON ERROR statement, and program execution continues from there. In this case, a CSUB compiled with EOL OFF will behave exactly like its interpreted counterpart.

ON TIMEOUT and ON END Events (with EOL OFF)

ON TIMEOUT and ON END events are also processed as soon as they occur. If a statement of the form:

```
ON TIMEOUT device selector GOSUB location
ON END io_path GOSUB location
```

was established in the context where the error occurred, or:

```
ON TIMEOUT device selector CALL subprogram name
ON END io_path CALL subprogram name
```

was established in the program, the "next line" pointer will be saved.

After the specified subroutine (in the case of GOSUB) or subprogram (in the case of CALL) has been executed, control will return to the line pointed to by the "next line" pointer, and program execution will continue from there. In the case of a CSUB compiled with EOL OFF, the "next line" pointer is pointing to the SUBEND statement of the CSUB where the event occurred. Therefore, after the specified subroutine or subprogram is executed, control will return to the end of the CSUB in which the event occurred, rather than to the line following the actual line where the error occurred.

With statements of the form:

```
ON TIMEOUT device selector GOTO location
ON END io_path GOTO location
```

or:

```
ON TIMEOUT device selector RECOVER location
ON END io_path RECOVER location
```

the location of the line where the event occurred does not get saved. Program control transfers to the location specified by the ON *event* statement, and program execution continues from there. In this case, a CSUB compiled with EOL OFF will behave exactly like its interpreted counterpart.

All Other Events (with EOL OFF)

All other events will not be processed immediately. They will be processed when end-of-line code is encountered. In the case of a CSUB compiled with EOL OFF, end-of-line code is not generated until after the SUBEND

statement. So, events (other than TIMEOUT, END or ERROR) will not be processed until after the entire CSUB has finished executing.

In the case of statements of the form:

```
ON event GOSUB subprogram name
ON event CALL subprogram name
```

the event will be processed after the SUBEND is reached. After the specified subroutine or subprogram is executed, control will return to the line pointed to by the "next line" pointer, which will be the SUBEND. Exceptions to this rule are outlined in the next section, "Special Considerations."

For statements of the form:

```
ON event RECOVER location
ON event GOTO location
```

control will pass to the specified location, and execution will continue from there, but *not until* the entire CSUB has been executed.

Special Considerations

There are some statements which *must* have EOL checking enabled, such as CALL statements. For these statements, EOL checking is enabled whether or not EOL ON is in effect. If EOL OFF is in effect, end-of-line checking will be enabled at the occurrence of the statement and disabled immediately afterward. What this does, is move the "current line" and "next line" pointers. Suppose you have the statement:

```
CALL Subtwo
```

in your CSUB. When the statement is encountered, the "current line" pointer will be set to that line, and the "next line" pointer will be set to point to the following line. To see what this means, consider the following subprogram:

```
10   SUB Thissub
20   !$EOL OFF
30   ON ERROR CALL Errproc
       .
       .
100  CALL Subtwo
110  PRINT A
       .
       .
500  SUBEND
```


EOL checking is turned OFF in this CSUB, so when the compiled version of the above SUB is executed, the "current line" pointer will point to the first executable line of the SUB, and the "next line" pointer will point to the SUBEND statement.

When line 100 is reached, end-of-line checking will be turned on around the statement, and the pointers will be reset. The "current line" pointer will be set to point to line 100 (CALL Subtwo), and the "next line" pointer will be set to point to line 110 (PRINT A). EOL checking will be disabled as soon as line 100 is compiled, so the pointers will remain there.

Now, suppose an error occurs in line 490. Because the ON ERROR CALL exists, the subprogram Errproc will be executed. After this subprogram is executed, control will pass to the line pointed to by the "current line" pointer. The "current line" pointer is pointing to line 100, since the pointers were reset when the CALL was encountered. So, after the error is processed, Subtwo will be executed. If the above example had used ON TIMEOUT or ON END, control would have passed to line 110.

The following is a list of statements that will have end-of-line checking enabled around them, even when EOL OFF is in effect:

CALL *sub name*	LINPUT *anything*
DEALLOCATE *anything*	ON *expression* GOSUB *location*
ENABLE	OUTPUT *anything*
ENABLE INTR *anything*	PAUSE
ENTER *anything*	READ *read list*
GOSUB *location*	SEND *anything*
INPUT *anything*	STATUS *anything*
LABEL *anything*	TRIGGER *anything*

End-of-line checking will also be enabled around statements like:

```
10 GOTO 10 ! GOTO to the same line
```

or statements of the form:

```
10 IF expression THEN statement
```

where *statement* is any of the statements listed above, including `GOTO 10`.

In addition, end-of-line checking will be enabled around an empty LOOP ... END loop construct:

```
10 LOOP
11 END LOOP
```

Toggling EOL

Because of all the conditions outlined above, you should be very careful about toggling EOL ON and OFF. Every time you toggle EOL ON, the "current line" and "next line" pointers will be reset. This capability to toggle EOL can be useful, however. For example:

```
        .
110   ON INTR 7 CALL Intproc
120   ENABLE INTR 7
        .
        .
200   CALL Numcrunch
        .
        .
300   END
310   SUB Numcrunch
320   A=1
330   WHILE A
340     PRINT
        .
        .
900   END WHILE
910   SUBEND
```

Suppose that you are using the above program. The main program makes a call to the SUB Numcrunch. Line 110 of the main program:

```
110   ON INTR CALL Intproc
```

was set up so that the number crunching routine (Numcrunch) could be interrupted.

SUB Numcrunch is a highly computational routine working on a large amount of data. If you compiled this routine with EOL ON, your program would be unnecessarily large and much slower than you would like. On the other hand, if you compiled it with EOL OFF, an interrupt would never be acknowledged until the entire Numcrunch subprogram had finished executing. The statement in line 110 above would be ineffective. To get around this, you could do the following:

```
310   SUB Numcrunch
315   ! $EOL OFF
320   A=1
325   ! $EOL ON
330   WHILE A
335   ! $EOL OFF
        .
        .
900   END WHILE
910   SUBEND
```

Compile the subprogram with EOL OFF, and turn EOL ON around a single line in the WHILE loop. End-of-line checking would only be enabled during a very small percentage of the program, so you would not lose a significant amount of speed. Also, the compiled program would be much smaller than if EOL ON was used throughout. Events such as interrupts would be acknowledged and processed once for each iteration of the loop when line 330 is executed.

You should remember that your "current line" and "next line" pointers will be set to lines 330 and 340, respectively. Any:

```
ON event GOSUB
```

or:

```
ON event CALL
```

construct would return from the event processing to one of these lines:

- ON ERROR would return to line *330*
- ON TIMEOUT, ON END, and all other *events* would return to line *340*

Thus, if the program receives an ON TIMEOUT, ON END, or any other *event*, the EOL OFF directive is ignored.

Special Consideration

For the sake of space and speed, optimizing GOTO statements that go to lines other than themselves will not have EOL activities performed around them even if $EOL ON was used around the GOTO statement. In other words, events will never be processed or acknowledge after GOTO statements unless they go to themselves. So avoid depending on using the GOTO statement as the place to toggle EOL ON for event acknowledgement in a loop construct. Use another statement in the loop.

Using the DS Command

The DS compiler command is a tool provided with the compiler that can help you evaluate the performance versus code size of your program. When the DS compiler command is used, the compiler will generate statistical information on each CSUB or CDEF in your compiled program.

As soon as the compiler finishes compiling a SUB or DEF, a table like the following will be included in the compiler listing:

```
CSUB Header          :     60 bytes ==>   5%
Symbol Tables        :     78 bytes ==>   6%
CSUB Entry Code      :    432 bytes ==>  33%
CSUB Body Code       :    542 bytes ==>  41%
Event Lines          :     26 bytes ==>   2%
Constant Pool        :     26 bytes ==>   2%
Added Libraries      :     38 bytes ==>   3%
Relocation Tables    :    104 bytes ==>   8%
Data Statements Pool :     12 bytes ==>   1%
Local DIM Table      :     10 bytes ==>   1%
TOTAL CODE           :   1328 bytes ==> 100%
```

This tells you how large each section of your subprogram or function is and what percentage of the total code each section occupies. The sections include the following:

CSUB Header — The CSUB header will always occupy 60 bytes: the header code contains information that is needed to run the context.

Symbol Tables	This section includes all of the tables (symbol table, formal parameter table, token table, etc.) that are needed to run the context.
CSUB Entry Code	This section contains code that initializes the CSUB before it is run. The size of this code will be the same for any subprogram or function compiled. The size of this code may change from one release of the compiler to the next.
CSUB Body Code	The CSUB body code is the section containing your compiled program.
Event Lines	This section is comprised of special code that is emitted by the compiler to allow your compiled program to process events (such as ON ERROR, ON TIME, etc.). Each event destination line will have special code representing it.
Constant Pool	All string constants and IMAGE statements are stored in the constant pool. Duplicates are eliminated.
Added Libraries	This section contains common segments of code including math support code that are needed to execute an operation.
Relocation Tables	The relocation tables contain data needed to initialize the CSUB. These tables are used by the CSUB entry code to make sure that your program can run where it is loaded.
Data Statements Pool	This section contains the data in DATA statements.
Local DIM Table	The Local DIM table is a table needed to describe arrays or string variables declared locally in the CSUB.
TOTAL CODE	The TOTAL CODE entry in this table is the sum of all of the sections listed above.

If a program contains more than one program unit, a summary table will be printed at the end of compilation, generating the above information about all the code that was generated during compilation. You may notice that the TOTAL CODE entry in the summary table will not exactly match the total size of your code that is listed at the end of the compiler listing. The TOTAL CODE entry of the summary table and the total size of your code that is listed at the end of the compiler listing will usually differ by about 94 bytes or so. The total size number includes code that is needed by the operating system to manage your CSUB. This code is not included in the TOTAL CODE entry of the summary table.

A

Error and Warning Messages

Errors numbered from 0 to 999 are generated by the BASIC Language System. For information on these errors, please refer to the *HP BASIC Language Reference*. The compiler may emit some of these errors during compilation.

Errors numbered from 1002 to 1999 are errors generated by the BASIC compiler, and will have an associated error message. These are documented in the section of this appendix titled "Compiler Error Messages."

Error number 1002 indicates an internal compiler error. When such an error is generated, the compiler will print a message of the form:

error message / codes: *n1*, *n2*

where *n1* and *n2* are internal error codes. If the compiler generates such an error, you should contact your local HP sales office. Upon contacting your local HP sales office, you need to report the above error codes. If possible, send a copy of your program to the local HP sales office. At least include a listing of the section of code and the line where the error occurred. This will help to recreate the error condition(s) and find a solution to your problem much quicker.

Warning messages 1 through 22 do not prevent the compiler from generating code. However, the code generated may not perform in the manner that you expect.

Compiler Error Messages

1001 COMPLEX type not allowed here.

1002 Internal compiler error. Note the error codes reported and contact your local HP sales office.

1003 Line number exceeds limit. Re-compile with KEEP OFF. Since the compiler adds more lines and re-numbers your program while compiling with KEEP ON, you may end up with the tail end of your program having line numbers exceeding the limit that your editor can cope with.

1004 To many SUBs/DEFs. Recompile without a compile list. Whenever the COMPILE command contains a list of subprograms then the compiler is limited to 2048 subprograms in RAM.

1009 Unsupported Binary Group. The compiler does not support non-HP binaries. If you get this error message with an HP binary, please contact your local HP sales office.

1010 NON-COMPILABLE Command. Non-compilable commands are listed in the section "Compiler Limitations" found in the chapter titled "The BASIC Compiler." Remove these statements, or use conditional compilation.

1016 UNKNOWN Command. May be using a statement that is not recognized by the compiler. This statement may be supported by some foreign system binary. Isolate the statement and make sure it is part of the HP supported group. If all else fails, contact your local HP sales office.

1018 Context code too long. Recompile with $LONGCODE ON.

1019 Unrecognized CONTEXT. The compiler will only recognize contexts that begin with a SUB, DEF, CSUB or CDEF statement.

1020 Nested $IF Compiler Directives not allowed. Make sure that no $IF or $IFNOT directive appears between another $IF or $IFNOT and its corresponding $END directive.

1021 $END directive does not match a preceding $IF or $IFNOT. Make sure that every $END directive has a corresponding (preceding) $IF or $IFNOT directive.

1022 There is no closing $END directive for a prior $IF or $IFNOT.

1023 String overflow. Constant string is too big (too long) for locally dimensioned string. This type of construct:

```
DIM A$[1]
```

will generate this error:

```
A$ = "AB"
```

1025 Too many SELECT/CASE statements nested. You may have up to 100 nested SELECT/CASE statements.

1026 CASE structure mismatch. CASE, CASE ELSE or END SELECT encountered before SELECT. Check to see if SELECT is within a conditional compilation block.

1027 SELECT Expression and CASE Expression mismatch. This error may occur if the CASE expression does not match the SELECT expression (i.e., mixing string with numeric).

1028 END/SUBEND/FNEND not found. $IF directive may have inhibited it.

1029 Statement too complex. Possible IO List too long. Break up your statement into two statements, and recompile.

1030 Expression(s) too complex. Too many factors in statements. Break long expressions into two pieces across two statements, and recompile.

Compiler Warning Messages

1 Unrecognized compiler directive; directive ignored. Check spelling and syntax.

3 Unrecognized interpreted BASIC control variable name; control variable ignored. Make sure the first character of the control variable is alphabetic (A-Z, a-z).

7 Improper compiler control variable name; control variable ignored. Make sure the first character of the control variable is alphabetic (A-Z, a-z).

8 Improper compiler directive syntax; directive ignored.

12 Improper control variable value or syntax; control variable ignored. First character of control variable must be alphabetic (A-Z, a-z); value must be an integer.

13 Improper ON/OFF toggle value. Check syntax.

14 Improper character in compiler directive or command string; remainder is ignored. Check spelling and syntax.

15 Directive does not appear at the beginning of the context; directive ignored. This directive's set location is at the beginning of a context. It must appear after a SUB or DEF statement and before the first executable statement in that context.

16 Possible DEAD Code after the SELECT statement.

17 A GOSUB to the same line is not allowed.

18 Referenced IMAGE/DATA was never found in CONTEXT. You have an I/O or RESTORE statement that references a non-existent IMAGE/DATA statement. This is all right in interpreted BASIC as long as the referencing statement is not executed. Compiled BASIC requires that all IMAGE or referenced DATA statements exist. You might also check to see if the statements are contained within an $IF ... END or $IFNOT ... END compiler directive pair.

19 Label(s) listed above this warning are not defined in this context. This is permitted in interpreted BASIC, as long as the labeled statement is never executed. Compiled BASIC requires that all labels exist. You might also check to see if the labeled statement is contained within an $IF ... END or $IFNOT ... END directive pair.

20 REDIM used with $SA ON. Multi-dimensional arrays will be affected. This warning will be generated if your program has $STATICARRAYS ON specified and a REDIM is encountered. This warning is ignored if the REDIM is working on:

- an array in COM
- a single-dimensional array
- a multi-dimensional array whose *last* dimension is being changed

Pay attention to this warning if you are REDIMming a local, multi-dimensional array and you are REDIMming some dimension other than the last.

21 Improper ID on Declaration Directive. Make sure you are using the directives COMPLEX and REAL properly. A function name specified with these directives *must* begin with FN and *cannot* end with the character $. This warning will also be generated when a function name is specified in one of these directives and the function is never called within the context that declared it.

22 Compiler created Mainsub and a SUB named Mainsub was also found. It is not illegal to have two subprograms with the same name but it can be confusing to the system. The first instance of the SUB will be the one that will always be executed.

B

Troubleshooting

This appendix is a "what to do if ... " section. It lists some common problems that you may encounter when using the compiler, suggests a possible cause and solution, and tells you where in the manual to go for more detailed information.

How Problems Are Presented

Each problem is presented in the following format:

Problem

Probable Cause

This section suggests a likely cause.

Solution

This section offers a solution if the problem was caused by conditions listed in the above section.

Reference

This section tells you where to look for details. For example:

Chapter/Section	Page
Writing Efficient Programs	6-3

Compiler Problems

To assist you in finding the task which you would like to perform, here is an index for the "Troubleshooting" chapter.

Problem and Solution Reference

There are 16 problems and their solutions listed in this section. Each problem has listed with it a probable cause, a solution for the problem, and a reference to more detail.

Compiled Program Runs Too Slowly

Probable Cause

The speed of your program will depend upon a number of criteria. Using integers instead of real numbers and constants instead of variables whenever possible will be of help. Also, certain compiler directives or compiler commands can be used to enhance program speed.

Solution

In very general terms, to get the fastest code (while possibly increasing the size of your compiled code), you should use the BEST command option in your compiler invocation command:

```
COMPILE: BEST
```

Be aware that BEST will set a number of compiler directive defaults, such as EOL OFF and STATICARRAYS ON. These two directives in particular may adversely affect the way in which your program works. If you compile with BEST and encounter problems, try one of these invocation commands:

```
COMPILE: BEST, EOL ON
COMPILE: BEST, SA OFF
COMPILE: BEST, EOL ON, SA OFF
```

A program that consists largely of trigonometric and exponential functions will not achieve a great speed increase no matter what you do. Likewise, programs that consist of mainly I/O and graphics will show minimal speed increases when compiled. Programs that gain the most speed when compiled are those that are compute bound. Programs consisting of integer computations benefit the most by compilation.

Reference

Chapter/Section	Page
STATICARRAYS (SA) Directive	5-40
Writing Efficient Programs	6-3
Optimizing Your Program	6-7
BEST Command	6-12

Compiled Program is Too Big

Probable Cause

Writing efficient programs entails establishing a balance between code size and program speed. This balance can be achieved through the judicious use of compiler directives.

Solution

In general, you will get the best compromise of space and speed by using the compiler invocation command:

```
COMPILE: BEST, OP OFF
```

Be aware that the BEST command will set a number of compiler directive defaults, such as EOL OFF and STATICARRAYS ON. These two directives in particular may adversely affect the way in which your program works. If you compile with BEST and encounter problems, try one of these invocation commands:

```
COMPILE: BEST, OP OFF, EOL ON
COMPILE: BEST, OP OFF, SA OFF
COMPILE: BEST, OP OFF, EOL ON, SA OFF
```

Reference

Chapter/Section	Page
STATICARRAYS (SA) Directive	5-40
Writing Efficient Programs	6-3
Optimizing Your Program	6-7
BEST Command	6-12

Program Runs in Interpreted Mode but not When Compiled

Probable Cause

There could be a number of reasons for this, but three of the most common are:

- improper use of the EOL directive
- improper use of conditional compilation
- improper use of compiler directives

Solution

If you are toggling EOL ON and OFF throughout your program, turn EOL ON throughout and see if this helps.

If you are using conditional compilation, make sure that all of your $IF and $IFNOT directives have $ENDs, and that you are not conditionally *not* compiling an essential section of your code.

Using any compiler directive improperly may cause your compiled program to run incorrectly. The best way to see if compiler directives are the problem is to use:

```
COMPILE
```

without including any other compiler directives in the code. After the program runs this way, you can start setting various compiler directives in the code. You will have to recompile and run the program each time you do this. Repeat this process until you find the compiler directive which caused the problem.

Reference

Chapter/Section	Page
Compiler Directives	4-2
Commands in the Options List	4-15
IF and IFNOT Directives	5-16
Using the EOL Directive	6-13

System Hangs During Compilation

Probable Cause

There are two probable causes for this problem:

- Since the compiler runs quietly (does not produce any messages as it compiles), it may appear that the system has hung when it has not. This is especially true if you are trying to compile a very large program.
- You may be compiling to a printer that is not on-line.

Solution

For the first case, you should use the compiler invocation command:

```
COMPILE: SHOW
```

which tells the compiler to generate messages during compilation.

In the second case, check to make sure your printer is on-line.

Reference

Chapter/Section	Page
Commands in the Options List	4-15
Compiler Output	4-20

Memory Overflow Error During Compilation

Probable Cause

You may get a memory overflow error if you do not have enough RAM to compile the entire program. Keeping the source code while you compile will greatly increase the memory requirements for a given program.

Solution

STORE a copy of the source code on disk, and compile the program with the invocation command:

```
COMPILE: KEEP OFF
```

or use the KEEP OFF compiler directive in each of your SUBS. This will cause the compiler to discard the source code as it is compiled. If your program is still too large, you can compile one SUB at a time. Do this by preceding each SUB (or group of SUBs) to be compiled with a main program that consists solely of an END statement. After all subprograms have been compiled you can remove the BASIC Compiler and load all of the subprograms into one program. This process will give you a program that is small enough to fit within your memory constraints. If not, you need more memory.

Another possible solution would be to recompile your program with the following compiler command:

```
COMPILE: BEST, OP OFF, KEEP OFF
```

Reference

Chapter/Section	Page
Switches in the Options List	4-10
KEEP Directive	5-20
OPTIMIZE Directive	5-27
BEST Command	6-12

Compiler Did Not Create Mainsub for Program

Probable Cause

There are two common reasons for this:

- Mainsub will not be created for main programs that consist of only an END statement.
- When the dummy main program is created, the END statement is marked to indicate that this main program was created by the compiler. If you delete the SUB Mainsub and the `CSUB Mainsub`, and you create a new main program, this new main program will not be compiled if you use the END statement that was at the end of the dummy main program. This is because the compiler will recognize that the END statement is marked, and it will think that it is the dummy main program.

Solution

If your main program consists of only an END statement, then there is no reason for the compiler to create Mainsub. If you re-use the END statement of the dummy main program to create a new main program, you must move the cursor to the END statement, and press [ENTER] ([Return] key on an ITF Keyboard). This will remove the mark that identifies the dummy main program.

Reference

See the section titled "A Dummy MAIN Program and Mainsub" in chapter 2, page 2-6.

Cannot TRACE or Single-step Compiled Program

Probable Cause

TRACE and single-stepping are not supported by the compiler.

Solution

Do not use TRACE or single-stepping.

Reference

See the section titled "Compiler Limitations" in chapter 2, page 2-7.

Error 1010—Non-compilable Command

Probable Cause

You are using a command that is not appropriate in a compiled environment, and therefore not supported by the compiler.

Solution

Remove any of the following commands:

```
GET     RE-SAVE     SAVE
```

from your program. Note that GET followed by a file name is allowed. GET followed by a file name and line number is *not* permitted.

Reference

See the section titled "Compiler Limitations" in chapter 2, page 2-7.

Error 1018—Context Code Too Long

Probable Cause

Code that uses forward addressing that exceeds 32K bytes of code must use extended addressing.

Solution

Put the directive LONGCODE ON in the offending subprogram(s). We do not recommend turning LONGCODE ON throughout the program as this directive greatly increases the size of your compiled program.

Reference

See the section titled "LONGCODE (LC) Directive" in chapter 5, page 5-22.

Program Runs on One Computer but not Another

Probable Cause

You are compiling on a computer with a MC68020/30 processor and MC68881/82 coprocessor and you ran the program on a computer without them.

Solution

Either run the program on a computer that has an MC68020/30 processor and MC68881/82 coprocessor, or compile the program with the compiler directives MC68020 OFF and MC68881 OFF.

Reference

Chapter/Section	Page
MC68020 (MTP) Directive	5-24
MC68881 (MCP) Directive	5-25

Configuration Error at Run Time

Probable Cause

You are trying to access a binary or a subprogram that is not present.

Solution

Make sure that all needed binary or user subprograms are available at run time.

Reference

Chapter/Section	Page
Compiler Limitations	2-7
Commands in the Options List	4-15

Internal Compiler Error (1002) at Compile Time

Probable Cause

Internal compiler errors are very rare. They are caused by problems with the compiler itself rather than problems with your program.

Solution

When internal compiler errors are generated, the compiler will print a message of the form:

error message / codes: *n1,n2*

where *n1* and *n2* are internal error codes. If your compiler generates such an error, you should contact your local HP sales office and report the numbers, as well as the source of the line which caused the error. It would also help to have a copy of the program available to help recreate or talk through the error condition.

Wrong Line Number Reported with Run-Time Errors

Probable Cause

There are two probable causes for this problem:

1. You are compiling a subprogram using SL and then loading it into another program using LOADSUB ... FROM. This gives you the wrong line number when a run-time error occurs.
2. Compiling a program with SAVELINENUMBER OFF (the default condition) will affect the line number reported with any run-time error messages.

Solution

A solution for the first probable cause is to remember that separately compiled subprograms with SL ON will report the wrong line number if a run-time error occurs in the compiled subprogram. For example, you may compile a subprogram using COMPILE: SL with line numbers ranging from 1 to 10 and store it in a file. First make sure that this subprogram will cause a run-time error such as division by zero. Perform a LOADSUB of the compiled CSUB into another program which contains lines with line numbers greater than 10. The loaded CSUB will end up with a line number greater than 10. Running the program which calls the compiled CSUB will generate a run-time error which points to a line number less than 10. This line number should be in the CSUB, but when you EDIT the program you will be looking at the wrong line. This is due to the fact that when the CSUB was called it told the BASIC Language System that its line numbers are in the range of 1 to 10 since it was compiled that way. The CSUB had no knowledge of where it was loaded.

A solution for the second probable cause is either re-compile the program with SAVELINENUMBER ON (which we do not recommend, as it will make your code unnecessarily large), or read the reference provided below.

Reference

See the section titled "SAVELINENUMBER (SL) Directive" in chapter 5, page 5-37.

Too Many Error/Warning Messages

Probable Cause

If the compiler is generating large amounts of warning and error messages, you have some problems with your program. You will need to correct these errors. In the meantime, tell the compiler not to prompt you when errors occur, but to continue compiling the rest of the program.

Solution

To suppress the prompt (do you want to stop now, or continue compiling?) when errors occur, use the compiler invocation command:

```
COMPILE: ERROR OFF
```

or include the compiler directive:

```
$ERROR OFF
```

in each subroutine where you wish to have this prompt suppressed. The compiler will still report the appropriate error messages to the device specified for output, but it will not stop and ask if you wish to correct the error immediately.

Reference

Chapter/Section	Page
Switches in the Options List	4-10
Correcting Compile-Time Errors	4-18
Correcting Run-Time Errors	4-19
ERROR (ERR) Directive	5-15

Computer Hangs in Middle of CSUB

Probable Cause

A DELSUB, LOADSUB, or INITIALIZE of a memory volume was performed from within the program or from the keyboard.

Solution

Do DELSUB, LOADSUB, and INITIALIZE of a memory volume from the interpreted dummy MAIN program.

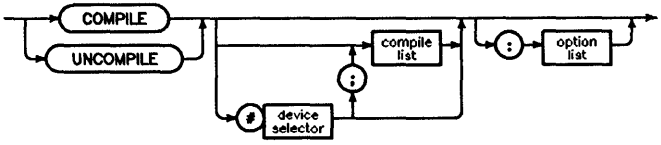
Reference

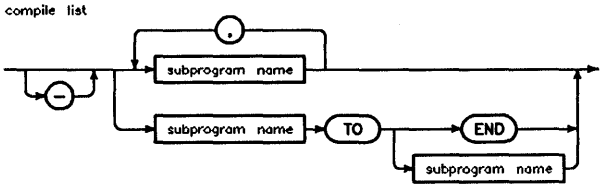
See the section titled "Compiler Limitations" in chapter 2, page 2-7.

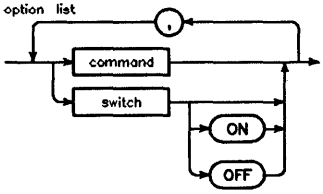
C

Quick Reference

The quidk-reference table on the following two pages summarizes the BASIC Compiler directives, switches, and commands.







Directive	Switches and Commands	Default	Compiler Feature Affected	Page
COMPLEX	none	none	Sets functions to COMPLEX type	5-6
CONFIGCHECK	CC[1]	ON	Checks subprograms and binaries	4-10, 5-10
CONTROLVAR	C	none	Control variable values	5-11
EOL	EOL[1]	ON	End of line activity code	4-10, 5-13
ERROR	ERR[1]	ON	Prompt when errors occur	4-11, 5-15
IF	none	none	Conditional compilation	5-16
IFNOT	none	none	Conditional compilation	5-16
KEEP	KEEP[1]	ON	Keep the source code	4-11, 5-20

C

[1] Switch

Directive	Switches and Commands	Default	Compiler Feature Affected	Page
LONGCODE	LC[1]	OFF	Extended addressing code	4-11, 5-22
MC68020	MTP[1]	[3]	Code specific for MC68020/30 processor	4-11, 5-24
MC68881	MCP[1]	[4]	Code specific for MC68881/82 math coprocessor	4-12, 5-25
OPTIMIZE	OP[1]	OFF	Optimize for space (OFF)/speed (ON)	4-12, 5-27
OVERFLOWCHECK	OC[1]	ON	Overflow checking	4-12, 5-29
RANGECHECK	RC[1]	ON	Out-of-range value check	4-12, 5-31
REAL	none	none	Sets functions to REAL type	5-33
SAVELINENUMBER	SL[1]	OFF	Save program line numbers	4-13, 5-37
STACKCHECK	STC[1]	ON	System stack bound check	4-14, 5-39
STATICARRAYS	SA[1]	OFF	Use of static local arrays (REDIM not used)	4-12, 5-40
SYMBOLS	ST[1]	OFF	Symbol table listing	4-14, 5-42
Not a Directive	BEST[2]	none	Produces the fastest code	4-15
Not a Directive	DS[2]	none	Prints statistical information	4-16
Not a Directive	HIDE[2]	none	Provides security to CSUB and CDEF parameters	4-16
Not a Directive	SHOW[2]	none	Outputs status of compilation	4-17

[1] Switch
[2] Command
[3] If the computer used has a MC68020/30 processor, then the default is ON.
[4] If the computer used has a MC68881/82 coprocessor, then the default is ON.

D

Glossary

The following is a list of terms used in this manual:

CDEF	is a section of compiled code that results from compiling an interpreted BASIC function.
CSUB	is a section of compiled code that results from compiling an interpreted BASIC subprogram or main program. CSUBs can also be generated on the Pascal Workstation System using the BASIC CSUB utility. Note that BASIC CSUBs created using the BASIC CSUB Utility are not the same as CSUBs created using the BASIC compiler.
Compile List	is an optional part of the compiler invocation command. A compile list is a list of subprograms that are to be included in the compilation (or, to be uncompiled). If the compile list is preceded by a minus sign (−), the list specifies the subprograms that are *not* to be compiled or uncompiled.
Compiler Command	is an optional element of the compiler invocation command. Compiler commands are used to control the compilation environment.
Compiler Directive	is a command to the compiler which is included within an interpreted BASIC program before compilation begins.
Compiler Invocation Command	is any command that is used to invoke the BASIC compiler. The invocation command may include a compile list and/or an options list.
Compiler Switch	can be included in the compiler invocation command options list to change the default values of certain compiler directives.
Compute-Bound Code	is code that consists of mainly computational statements (including assignment statements, arithmetic statements, etc.).

Context	is a section of code in an interpreted program. The main context is all the text appearing up to the END statement. All the text from a SUB (or DEF) statement to the next SUB (or DEF) statement makes up a context. Contexts that are not main programs are subprograms.
Context-Scoped	refers to the scope of a compiler directive. A context-scoped directive is one which is reset to a default value at the start of each context. Such directives assume their default values at the beginning of each context and retain these values unless they are explicitly changed.
Control Variable	is used for conditional compilation. It consists of one or more characters and digits. The first must be alphabetic, and it must be unique. The compiler distinguishes between upper and lower case, so there are 52 possible starting letters (A-Z, a-z). The compiler can distinguish between control variables and regular program variables so you can have a control variable in a program with the same name as a program variable. Control variables are only used with IF/IFNOT directives.
Dummy Main Program	is created at compile time. At compile time, the compiler converts your main program into a SUB, and then compiles it into a CSUB called Mainsub. Since this means that your main program no longer exists, the compiler creates a dummy main program for you. The dummy main program created by the compiler contains all COM declarations from your original main program, a call to the SUB Mainsub, and an END statement. It also may contain an OPTION BASE statement.
Function	refers to a procedural call that returns a value. The call can be to a user-defined-function subprogram (such as FNInvert) or a machine-resident function (such as COS or EXP). The value returned by the function is used in place of the function call when evaluating the expression containing the function call.
Global-Scoped	refers to the scope of a compiler directive. A globally scoped directive is one that retains its value across context

boundaries. In other words, its value is not reset to the default at the beginning of each context.

IF ... END — is the segment of text from (and including) the IF compiler directive, to (and including) its corresponding END directive. The same is true for the IFNOT compiler directive.

I/O-Bound Code — is code that consists mainly of I/O operations (which includes graphics). This would include PRINT, READ, etc.

Main Program — (or main context) is a section of code at the beginning of a BASIC program which is terminated by an END statement. When compiled, the main program becomes the first CSUB of your compiled program called Mainsub.

Options List — is an optional part of the compiler invocation command. The options list contains compiler commands and/or compiler switches and can be used to change the default values of a number of compiler directives or to specify special actions to be taken by the compiler during compilation.

Scope — is a term used in relation to compiler directives. The scope of a directive is the amount of code that it affects. Compiler directives with context scope are reset to their default values at the beginning of each new context. Compiler directives with global scope will retain their values across context boundaries.

Subprogram — is a section of code making up a context. This code can perform every function that a main program can with the exception of "execution." A subprogram must be called by a main program or another subprogram in order to be executed. The code in a subprogram begins with a SUB or DEF statement and ends with a SUBEND or FNEND statement.

Subroutine — is a section of code occurring within a main program or a subprogram. This code is designed to perform a specific task and in most cases is used more than once within a program. Subroutines are executed when a statement such as `ON` *event* `GOSUB` or `GOSUB` is encountered. The subroutine must reside in the same context as the GOSUB statement.

Index

D

E

F

G

H

I

K