WATFIV USER'S GUIDE

CONTENTS.....Page

1. INTRODUCTION

This section provides information required by the user of the WATFIV compiler, and could be duplicated, with appropriate acknowledgements, by an installation for distribution as a "WATFIV Programmer's Guide". Note, however, that the material provided is based on the 'standard' WATFIV compiler, and thus, may require some installation-dependent editing. Major sections, subsections or paragraphs which contain details that depend on the options described in section 2.7 of the WATFIV Implementation Guide are marked in the left margin with an '|'(1).

It is intended that these marks would be used as guides to areas that may require editting, because of options selected when the compiler was generated at the installation, by the person responsible for preparation of this manual for distribution to users.

It should be noted that this User's Guide is not a manual or text on FORTRAN programming. The Guide is intended for the 'experienced' FORTRAN programmer, i.e., one who already has some familiarity with FORTRAN in general and likely some familiarity with IBM's FORTRAN compilers. In particular, the authors of the Guide assume the reader has access to the following IBM publications:

IBM System/360 FORTRAN IV Language, Form GC28-6515
IBM System/360 FORTRAN IV (G and H) Programmer's Guide, Form GC28-6817

---

(1) This WATFIV User's Guide has been prepared using SCRIPT and the unformatted input file is available on request.

2. CONTROL CARDS

Two levels of control cards are required to run a program using WATFIV - control cards recognized by the compiler itself, and those required by the operating system job scheduler.

2.1 WATFIV CONTROL CARDS

Two control cards - $JOB and $ENTRY - are required to run a program under WATFIV. Their use is shown in the following diagram which defines a WATFIV job.

```
$JOB          identification,parameters
      .
      .
      . FORTRAN program consisting of a main
      . program and any number of subprograms
      .
      .

$ENTRY
      .
      . any data required by the program
      .
```

The control field $JOB is punched in columns 1 to 4 of the card, and $ENTRY in columns 1 to 6; column 5 and 7, respectively, must be blank. Columns 8 to 80 of the $ENTRY card are ignored. Accounting information and job parameters that may appear on the $JOB card are described in section 3 on page 6.

The $ENTRY card is required to initiate execution of the compiled program even if no data cards are present.

The FORTRAN program and data are punched according to the usual rules of FORTRAN. The main program and subprograms follow one another, as shown in the following example:

```
DIMENSION X(10)
 .
 .
 .
END
SUBROUTINE EXAMPLE
 .
 .
 .
END
FUNCTION FN (A)
 .
 .
 .
END
SUBROUTINE RTN (X,Y)
 .
 .
 .
END
```

The main program need not appear first.

2.2 OS/VS CONTROL CARDS

The OS/VS control cards are necessary to load WATFIV into main memory. Once there, it can process any number of WATFIV jobs in sequence, i.e., an OS/VS job consists of a 'batch' of one or more WATFIV jobs. Since the operations' personnel at your installation may collect WATFIV programs to run as a batch, or batching may be provided by other mechanisms, knowledge of the OS/VS control cards is not essential. The details are provided for those who must batch their own jobs for submission to the computer.

One form of an OS/VS job to run a batch of WATFIV jobs is shown in the next figure.

```
//jobname  JOB  accounting
//    EXEC    WATFIV
//GO.SYSIN  DD  *
$JOB            id,parms

      Program 1

$ENTRY

      Data 1

$JOB             id,parms

      Program 2

$ENTRY

      Data 2

$JOB             id,parms

      Program n

$ENTRY

      Data n

$STOP'1'
/*
```

---

(1) Optional as end of batch indicator.

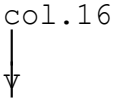
2.2.1 WATFIV CATALOGUED PROCEDURE

The following catalogued procedure is the standard WATFIV procedure. The WATLIB DD card references WATFIV's function library and any data sets containing subprograms that might be called by the user's program. The catalogued procedure WATFIV also contains DD cards for the card reader, printer, and punch (Fortran units 5, 6, and 7 respectively) and for temporary sequential data sets on units 1, 2, 3, and 4 (DD names, FT01F001, FT02F001, FT03F001, FT04F001, respectively). Certain files are given read-only status by WATFIV; see section 11.1 on page 86 for further information on this feature. Concatenation of compiler input is discussed and illustrated with examples in section 11.3 on page 87. Hereafter, the term 'job' will mean a WATFIV job. A listing of the WATFIV procedure for an OS/VS system follows:

```
//WATFIV  PROC  PROG=WATFIV,LIB='WATFIV.FUNLIB',V='',VOL=WATFIV
//        JB='WATFIV.JOBLIB'
//GO      EXEC  PGM=&PROG,REGION=150K
//STEPLIB DD  DSN=&JB,DISP=SHR,UNIT=SYSDA,VOL=SER=&V.&VOL
//WATLIB  DD  DSN=&LIB,DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),DISP=SHR,
//            VOLUME=SER=&V.&VOL,UNIT=SYSDA
//        DD  DSN=WATFIV.WATLIB,DISP=SHR,VOL=SER=&V.&VOL,UNIT=SYSDA
//FT01F001  DD  SPACE=(TRK,(20,10)),DCB=(RECFM=VS,BLKSIZE=256),UNIT=SYSDA
//FT02F001  DD  SPACE=(TRK,(20,10)),DCB=(RECFM=VS,BLKSIZE=256),UNIT=SYSDA
//FT03F001  DD  SPACE=(TRK,(20,10)),DCB=(RECFM=VS,BLKSIZE=256),UNIT=SYSDA
//FT04F001  DD  SPACE=(TRK,(20,10)),DCB=(RECFM=VS,BLKSIZE=256),UNIT=SYSDA
//FT05F001  DD  DDNAME=SYSIN
//FT06F001  DD  SYSOUT=A,DCB=(RECFM=FA,BLKSIZE=133)
//FT07F001  DD  SYSOUT=B
```

3. JOB CARD FORMAT

Starting in column 16 of the $JOB card you may punch your user account number or identification. This may be followed by a comma and a selection of job parameters from the list below. Otherwise, leave at least one blank column after your account number to separate it from any comments.




```
e.g.,  (1)  $JOB  D059JOEUSER  NO PARAMETERS
       (2)  $JOB  P7735YOURID,KP=29,TIME=1,NOLIST PARMS GIVEN
```

WATFIV control cards are identified by a C$ or $ starting in column 1. The main advantage of the C$ control characters is the ability to compile the source program under a FORTRAN compiler other than WATFIV, since these control cards will be treated as comments. WATFIV will thus treat the $ or C$ as valid control characters but all control cards will be prefixed with a C$ when they are printed out (except $JOB and $ENTRY cards).

3.1 WATFIV OPTIONS

The following options may be used on the WATFIV $JOB card or placed on the C$OPTIONS card. Where applicable, the standard default options are underlined, and short forms of the keywords are placed in parentheses. Where a choice is available, the options are separated by a /. The parameters may be punched in any order and may extend to column 79. The last parameter must be followed by at least one blank column. The C$OPTIONS or $JOB card scan will be terminated upon encountering the first blank in the options field. If an option is mispunched a warning message will be issued but the scan for remaining parameters (if any) is continued. If any parameter is specified more than once, the right-most value is used. The C$OPTIONS card was introduced to allow the programmer to change compile- or execution-time options within the program. Use of the C$OPTIONS card is shown in subsection 3.2.2 on page 10.

3.1.1 COMPILER CONTROL OPTIONS

(T)ime=s / (m) / (m,s) / <u>(3)</u>
: Specifies the time, in minutes and seconds, to be used as the upper limit for execution of your program.

(P)ages=n / 999
Specifies the maximum number of pages of output you wish to allow your program to produce at execution time.

(L)ines=n / 63
Specifies the number of lines that will be printed per page at compile and execution time. If n is set to 0, WATFIV will suppress all compiler-generated page ejects; the page count will be incremented after every 63 lines printed. If n is greater than 66 and the job is run with the DEBUG(1) parm, the lines per page will be reset to 66.

CHECK / NOCHECK / FREE
The CHECK option will cause the compiler to check, at execution time, for attempted use of variables which have not been assigned a value (undefined variables).

Using the NOCHECK option suppresses this execution-time checking of undefined variables (except those used as subscripts of arrays), thereby reducing the amount of object code produced and execution time consumed. (Note: NOCHECK does not suspend subscript error checking; for example, attempting to use an undefinied variable as an array subscript when NOCHECK is in effect would result in an error message indicating that the subscript is undefined.

The FREE option is the same as CHECK, but WATFIV will initiate execution of your program even if compile-time errors were encountered. If an executable statement which contained a source error is subsequently encountered, execution is terminated.

KP=29 / 26
KP=29 specifies that the program was punched on a model 029 (EBCDIC) keypunch. KP=26 indicates that it was punched on an 026 (BCD) keypunch. All WATFIV programs typed on any key-driven terminal are in 29 keypunch mode.

LIST / NOLIST
LIST causes the compiler to produce a source listing of the program. NOLIST suppresses this listing.

LIBLIST / NOLIBLIST
LIBLIST causes the compiler to produce a listing of the source subprograms retrieved from the subprogram libraries. NOLIBLIST suppresses this listing of the library routines. This option is not in effect if encountered in a subprogram library. Note that the LIST/NOLIST and LIBLIST/NOLIBLIST

---

(1) The DEBUG parm is explained in subsection 3.3 on page 12.

parameters are independent.

WARN / NOWARN
WARN causes the compiler to print all diagnostics of a severity less than a fatal error. NOWARN suppresses the printing of these messages.

EXT / NOEXT
EXT causes the compiler to print all extension messages, that is, indications of any WATFIV features that may not work under an IBM FORTRAN compiler. NOEXT suppresses all extension messages.

PGM=nnnnnn
If the WATFIV program to be executed is a mainline program that is stored in a library, it can be accessed by specifying its name as a "PGM=" operand.

SUB / NOSUB
Pseudo-Variable Dimensioning (PVD) was implemented in WATFIV to ease the programmer's task of implementing existing programs which use non-standard FORTRAN conventions for passing arrays to subprograms. WATFIV will allow a subprogram which receives an array through its argument list (this is called a subprogram dummy array) to have a rightmost dimension of 1. With PVD, this rightmost dimension of the "dummy" array will be adjusted by the compiler to occupy as much as possible of the storage allocated to the calling array. For such subprogram arrays, the option SUB causes checking to be done to ensure that each subscript used in this "dummy" array does not exceed its limit as defined in the subprogram. Specifying NOSUB will permit access to any member of the dummy array, as long as this array element is within the storage allocated to the calling array.

3.1.2 PROFILER CONTROL OPTIONS

The options for the WATFIV PROFILER (which is discussed in subsection 6.3.1 on page 37) are specified in the same manner as the above compiler options. They cause WATFIV to produce profiler output for the sections marked by C$PROFON and C$PROFOFF cards described below. If no C$PROFOFF is encountered, all source statements from the C$PROFON card on, including any from the library, are profiled. Three different types of profiler output are available, and can be obtained by using one or more of the following options (the last encountered form of each option will determine whether the corresponding output is produced):

PROFC / NOPROFC
PROFC turns on the profile count option; NOPROFC turns it off.

PROFP / NOPROFP
PROFP turns on the percentage histogram option; NOPROFP turns it off.

PROFA / NOPROFA
Turns on the absolute histogram option; NOPROFA turns it off.

PROF / NOPROF
PROF is equivalent to specifying PROFC, PROFP, and PROFA; NOPROF is equivalent to NOPROFC, NOPROFP, and NOPROFA.

C$PROFON / C$PROFOFF
The C$PROFON card will activate the accumulation of PROFILER statistics. The C$PROFOFF card will disable this collection. Any number of groups of statements can be surrounded by the C$PROFON and C$PROFOFF cards and will be displayed in the PROFILER output. This control card must be used in conjunction with the PROFILER control options on the $JOB or C$OPTIONS card.

3.2 WATFIV CONTROL CARDS

The following control cards may be placed anywhere in the source program. They must be punched with a 'C$' in columns 1 and 2. WATFIV currently supports a few C$... cards not described in this section. They are merely earlier implementations of features which are descibed in subsection 3.1.1 on page 6. Users should attempt to use the newly introduced C$OPTIONS card. No warning/extension messages are issued to flag the use of these earlier control cards.

3.2.1 CONTROL CARDS TO EDIT SOURCE LISTINGS

The following control cards are never printed and can be used for the final "production run" to make the output look more presentable. These control cards provide no page or line skipping while the NOLIST option is in effect.

C$EJECT
This control card causes the compiler to skip to the top of the next page to continue the listing of the source program.

C$SPACE n / 1
The insertion of this control card will cause the compiler to leave n blank lines in the source listing.

A source deck using these new cards might be as follows:

```
$JOB        D9999JOE.USER,NOLIST
                        cards not to be listed
C$OPTIONS LIST
                        cards to be listed
                        END
C$EJECT
                        subprograms
C$SPACE
                        END
$ENTRY
                        data cards
```

These control cards can be used to advantage when a large program is being tested. By suppressing print in areas where code has not been changed, the user can save on machine printout time and thus have the job run more economically.

3.2.2 OTHER WATFIV CONTROL CARDS

C$PROFON / C$PROFOFF
The C$PROFON card will activate the generation of PROFILER statistics. The C$PROFOFF card will disable this collection of PROFILER statistics. Any number of groups of statements can be surrounded by the C$PROFON and C$PROFOFF cards and will be displayed in the PROFILER output. This control card must be used with the PROF option on the $JOB or C$OPTIONS card.

C$ISNON / C$ISNOFF
The C$ISNON control card causes the execution-time tracing of any following statements until the C$ISNOFF card is encountered. At least one executable statement must precede

a C$ISNON card or a warning message is given. This is an execution-time control card in the sense that object code is generated for these control cards which must be executed to activate or deactivate the "ISN trace".

C$OPTIONS

This control card provides the programmer with the facility of changing options part way through the program.
With the recent proliferation of options available on the $JOB card for WATFIV, the C$OPTIONS card has been introduced to give the user more flexibility. This card may contain any options permitted on WATFIV's $JOB card. An example of possible applications of this new control card follows:

```
$JOB     WATFIV
         REAL ARRAY (10,10)
         .
         .
         CALL SUB26 (ARG1,ARG2)
         .
         .
         CALL PVD (ARRAY)
         .
         .
         END
C
C  User changes keypunch mode to BCD and
C  permits extensions to be printed
C
C$OPTIONS KP=26,EXT
C
         SUBROUTINE SUB26 (X,Y)
         .
         .
         .
         END
C
C  User reverts back to EBCDIC and suppresses
C  source listing; the NOSUB option will permit
C  the programmer to specify any subscript for
C  array MATRIX as long as the array element
C  falls within the bounds of ARRAY.
C
```

```
C$OPTIONS KP=29,NOLIST,NOSUB
C
         SUBROUTINE PVD (MATRIX)
         REAL MATRIX (10,1)
         .
         .
         .
         MATRIX (50,2)=5
         .
         .
         .
C  User sets up options of 40 lines/page
C
C$OPTIONS LINES=40
$ENTRY
         .
         DATA
         .
```

Using this feature, different options can be specified in various sections of the program.

3.3 STUDENT JOB STREAM ENVIRONMENT

Besides WATFIV's normal mode of operation, a student job stream environment can be activated by passing the parameter 'DEBUG'. In this environment, object decks are disabled when encountered in the input stream and all direct-access I/O statements are given execution-time error messages. The time, page, and line count also have a default and maximum as follows:

	Time (Sec)	Pages	Lines/Page
DEFAULT	2	5	63
MAXIMUM	3	10	66

4. USING WATFIV UNDER INTERACTIVE SYSTEMS

4.1 USING WATFIV UNDER TSO

A command procedure has been distributed with TSO WATFIV but it is likely that each installation will modify this procedure to meet its own requirements.

The standard procedure is called WATFIV and will default to doing all I/O to the terminal. The symbolic keyword operands INPUT and OUTPUT may be used to specify a file referenced by FT05F001 and FT06F001. For a more complete description of the WATFIV command type HELP WATFIV.

4.2 USING WATFIV UNDER CMS

An interface has been written for WATFIV to simplify its use under CMS and to make it more flexible by taking advantage of CMS facilities. The interface allows a user to type a program directly to WATFIV or to submit a program residing in a disk file to the compiler. Multiple filenames may be specified for input and using the options described below, it is possible to direct unit 6 output (listing output) to the printer, the terminal or a disk file.

Further options allow the user to specify execution mode and to have certain output typed at the terminal. The interface routine provides FILEDEFs for the standard input and output units, 5 and 6, as well as for WATLIB, the function library. Previously defined FILEDEFs are not overridden. Thus, the user may, for example, input a program from a tape by specifying a tape FILEDEF for unit 5 and issuing the WATFIV command (below) with a filename *, or the name of any FORTRAN file. If a library concatenation has been previously set up with a 'GLOBAL MACLIB ...' command, it will not be overridden by WATLIB's FILEDEF and the 'WATLIB MACLIB' library will not be available unless it appeared in the GLOBAL list. TXTLIB libraries will not be accessed by WATFIV, but compatible object code subroutines may be used from a MACLIB. Text and FORTRAN subroutines must be added to the MACLIB as files of type 'COPY' with the subroutine name as file name (see the MACLIB command in the IBM/370 Command Language User's Guide). The following command prototype may be used to invoke the WATFIV compiler.

```
WATFIV fn-i ... (DIsk  Concat   NODebug NOXtype Term   NOStats)
                 Print NOConcat Debug   XType   NOTerm Stats
                 TYpe                   NOWarn
                                        Ext
```

The fn-i (i=1,...) are the input filenames. They must be names of disk files (of type FORTRAN or WATFIV) or *, which

specifies that input is to be taken from the terminal. WATFIV will read all input files as one continuous job stream. If * is specified as the first filename, a $JOB card with the LIST option is provided and TYPE and STATS will be the default output option. Since WATFIV itself writes directly to the terminal there is no facility to eliminate the printing of job statistics. Since WATFIV is a batch processor it will attempt to read another job when the first is completed. If you do not wish to type in another job, hit carriage-return to signal end-of-file (end-of-batch). Refer to examples in subsection 4.2.2 on page 17 for further explanation.

4.2.1 OPTIONS

The default options for each group appear first. The shortest abbreviation acceptable for each option is shown in uppercase letters. If more than one of the options from the same group are specified, the last one typed is taken. The options may be typed in any order and must be separated by at least one blank (no commas).

Listing output options:

DIsk	The program listing and all unit 6 output is written to a disk file with filename fn-1 and filetype LISTING if CONCAT is in effect. A filename of WATFIV will be generated if fn-1 is *. The NOCONCAT option will cause a separate output disk file with filename fn-i for each input file.
Print (PRT)	The listing and unit 6 output are spooled to the virtual printer.
TYpe (*)	The listing and unit 6 output are typed at the terminal.

Note: If fn-1 is *, then TYPE will become the default and all options controlling execution-time output, diagnostic messages and job statistics will have no effect. The default CONCAT option will cause output to be placed in a WATFIV LISTING file. In the case of using the NOCONCAT option, if two program batches are input to WATFIV from the terminal, the listing file for the second will overwrite and destroy the listing file for the first.

Compiler-mode options:

Concat	This option causes WATFIV to treat all input files as one continuous job stream. All output is written to a disk file with a filename of the first file specified (fn-1) and filetype of LISTING. If * is specified for fn-1 the filename will default to WATFIV and a $JOB card will be generated. Hitting carriage return signifies end of file for terminal input.
NOConcat	This option causes each input file to be treated as a separate batch. Input for each filename is compiled separately. Each fn-i should contain at least one complete WATFIV job.

Execution-mode options:

NODebug	WATFIV will execute normally without the interactive execution-time debugging facilities.
Debug	WATFIV will execute in interactive debugging mode. See subsection 4.3.1 on page 18 for more information regarding debugging facilities.

The remaining options enable the user to elect to have unit 6 output typed at the terminal when DISK or PRINT has been specified for listing output. The output selected by these options for the terminal will also appear on the listing. These options will have NO effect if the listing output option is TYPE.

Execution-time output options:

NOXtype	This option may be used to negate an earlier XTYPE option specification. Otherwise, this option has no effect.
XType	Output on unit 6 will be typed at the terminal at execution-time as well as being written to the printer or a disk file as specified by the listing output option.

Diagnostic-message options:

Term	Compile-time error and warning messages, along with the statements in error, will be typed at the terminal. Execution time errors and traceback information will also be typed.
NOTerm	No error, warning or extension messages will be typed at the terminal if not in DEBUG mode. If DEBUG is specified, compile-time messages will not be typed but execution-time error messages will appear at the terminal.
NOWarn	Error messages will be typed at the terminal, but warning and extension messages will not.
Ext	Extension messages will be typed in addition to error and warning messages.

Note that when the statements in error are typed at the terminal by the interface routine, the maximum length for a statement is 20 cards (i.e., 19 continuation cards). Installations allowing longer statements will get, at most, the last 20 lines of an erroneous statement typed, unless they modify this interface to meet their needs.

Job statistics options:

NOStats	No end-of-job statistics will be typed at the terminal.
Stats	End-of-job statistics will be typed at the terminal.

4.2.2 USING THE CMS WATFIV COMMAND

An example of the use of the WATFIV command follows.

The file JOB WATFIV contains a $JOB card, with NOLIST specified as an option. The file ENTRY WATFIV contains a $ENTRY card. The file EX1 FORTRAN contains a small FORTRAN program that reads from unit 5 and writes on unit 6 until an end-of file condition is encountered on unit 5.

```
watfiv job ex1 entry *(type

$JOB   WATFIV  C,KP=29,NOLIST

$ENTRY
9            (The keyboard unlocked at this point for input)
   0.9000000E 01   0.8100000E 02
             (A blank line was entered to signify end-of-file.)

CORE USAGE ...
           ...
C$STOP                  (This card was generated by WATFIV)
R;
```

Another example follows. The file DATA WATFIV contains one record with a data card.

```
watfiv job ex1 entry data(xtype

  0.788999E 02   0.6225207E 04

CORE USAGE ...
           ...
C$STOP                  (This card was generated by WATFIV)
R;
```

4.3 USING THE INTERACTIVE DEBUGGING FACILITIES

4.3.1 INTRODUCTION

A new package has been developed to supplement the debugging facilities of the WATFIV compiler. Essentially, it introduces the capability of monitoring the execution of a WATFIV program interactively, from a terminal through CMS or TSO. No additions or changes to your programs are required in order to use this facility.

4.3.2 COMMAND SET

The command set is fairly small and simple, yet it provides some very useful debugging aids. It enables you to trace portions of your program, halt execution at various points, display or modify program variables, alter the logic flow of your program and correct certain execution-time errors interactively. A list of the commands and their functions follow:

TRACE isn'(1)'-range	turns on tracing in given range
OFF	turns off tracing.
STOP isn	specifies stop location. Execution may also be stopped with the attention key on the terminal
RUN	resume execution where program stopped.
GOTO stmt-no.	resume execution at line labelled by given statement number.
A ?	display contents of variable A, where A may be a simple variable, array name, or array element (in the current program segment'(2)').
A = value	display contents of variable A, where A may be a simple variable, array name, or array element (in the current program segment).

---

(1) isn: instruction sequence number (line number)
(2) A program segment is a mainline program or any subprogram.

EXIT	terminate debug session and return to CMS or TSO.
Execution-Time errors	if the error can be corrected by the modification of variables, execution may be restarted where interrupted. Otherwise, the error may be branched around with the GOTO command, or the EXIT command may be used.

Debugging commands are issued when the program is not executing and will be prompted for by the printing of 'CMD:' at the terminal. The first command prompt will be typed after successful compilation of the program. If compile-time errors are encountered in the program, WATFIV will return control to CMS or TSO without issuing a command prompt, unless the FREE option on the $JOB or C$OPTIONS card has been specified. If FREE is specified and source errors exist in the program, unpredictable results using debugging commands may result. More detailed information about specific commands follows.

TRACING

The TRACE command causes the line numbers within the specified range to be printed whenever the lines are executed (i.e., 'LINE n' is printed when line n is executed). The range must consist of two integers separated by a comma or blanks. The integers do not necessarily have to be valid line numbers from the program. For example

```
TRACE 1,999
```

will trace the entire program if its length is not more than 999 lines. Only one tracing range may be specified at one time. If more than one trace command is issued, the range specified by the most recent one will be in effect. Tracing may be turned off altogether with the OFF command.

STOPPING

Similarly, only one STOP location is in effect at one time and this will be the one most recently specified. When the program halts execution at the specified line, the stop location will no longer be in effect.

When using the attention key to stop program execution, it may be found that more statements have been executed than

was supposed. If the attention key is used to stop execution while tracing a program, often the line at which the program stops is not the same line for which the trace last printed a line number. It might also be noticed that, when the program is stopped in this way, the line at which it stopped is beyond a print statement in the sequence of execution and the print line has not yet appeared on the terminal. These situations arise because the program continues executing after the command to write a line to the terminal has been issued (by the program itself, or by WATFIV). That is, execution does not wait for the line to actually be printed, since this would slow it down considerably.

GOTO

The GOTO command may only be applied to statement numbers local to the current program segment. WATFIV issues warning messages for statement numbers which are not referenced and are on statements following a transfer. Statement numbers, for which such warnings have been issued, may not be used as operands of the GOTO command. Note that if execution is started with a GOTO in the middle of a DO-LOOP, unexpected results may occur if DO index variables have not been appropriately modified.

4.3.3 MODIFYING AND DISPLAYING VARIABLES

The contents of a variable may be displayed by typing the variable name followed by a question mark. The contents will be typed in a format appropriate to the variable type. If an entire array is being displayed, the elements will be printed in storage order, as in WATFIV free format output (see FORTRAN IV WITH WATFOR AND WATFIV: CRESS/DIRKSEN/GRAHAM, ch.8 pp.115-116). Similarly, when modifying a variable, the new value following the equals sign should conform to WATFIV free format rules for input.

A restriction on variable modification is that the new value must fit on one line. If an array being modified has more elements than will fit on one line, an error message will be given; however, if the variable is then displayed, it will be found that the values which fit on the line have been assigned to the appropriate elements of the array. Note that only variables which are accessible to the current program segment may be displayed or modified.

4.3.4 EFFICIENCY CONSIDERATIONS

The processing of debugging commands clearly takes time and thus will increase the cost of executing a program. The display and modification of variables can be costly if the program has a large number of variables, since these commands require a search of the data area for the variable name specified. Similarly, the GOTO command involves a search of the statement number list to find the address of the object code for the statement with the given statement number.

Tracing and stop locations require the monitoring of line numbers during execution. These commands then add some overhead to execution time. Tracing also causes extra code to be executed to print the line number each time a line is executed within the specified range. Thus, to minimize execution-time overhead, tracing should be limited to as small a range as possible and should only be used when necessary. When neither tracing nor a stop location is in effect, however, no monitoring overhead is added to execution time.

4.4 INTERACTIVE DEBUGGING OF WATFIV JOBS

As most FORTRAN programmers probably already know, WATFIV is an excellent debugging compiler. But even with WATFIV's superior diagnostic capabilities, debugging a program can be a tedious job. The programmer finds a mistake, re-edits the file containing the program (or re-types the card), and re-runs the program. Some of this tedium can be relieved by using WATFIV Interactive Debug under CMS or TSO. To use this feature, all that is needed is an up-to-date source listing of a program that has not been debugged, and, of course, the program itself. Then, the program can be run under Debug mode; when an error condition is encountered, control is returned to the programmer, who has the option of displaying the value of any variable (simple variables, arrays, or array elements), re-assigning the value of any variable, branching to any statement number in the program segment, or instructing the compiler to return control to him or her at any specified line. While doing this, the programmer can mark on the listing any changes which are necessary to make the program run; all that is then required to finish debugging is to edit the program once, making all necessary changes, and then re-run it once, to ensure that all required changes have been made.

An example of the use of Interactive Debug follows:

```
     $JOB
 1         REAL A(10),B(10)
 2         DO 2 I=1,9
 3         A(I+1)=I*2.0/0.19
 4   2     B(I)=I*SQRT(I*2.1)/4.3
 5         READ,N
 6   14    CALL STLINE(A,B,N,X,Y)
 7         PRINT,X,Y
 8         STOP
 9         END
10             SUBROUTINE  STLINE (X,Y,N,A,B)
11         REAL X(N), Y(N)
12   65    SX = 0.0
13         SY = 0.0
14         SXX = 0.0
15         SXY = 0.0
16   25    DO 9 I = 1, NN
17            XI = X(I)
18            SX = SX + XI
19            SXX = SXX + XI * XI
20            YI = Y(I)
21            SXY = SXY + XI * YI
22            SY = SY + YI
23   9     CONTINUE
24         XN = N
25         DEN = XN * SXX - SX * SX
26         A = (XN * SXY - SY * SX) / DEN
27         B = (SXX * SY - SX * SXY) / DEN
28   86    RETURN
29         END
     $ENTRY
```

This is the program that will be debugged. It is obvious that it would probably be unnecessary to employ something as powerful as Interactive Debug on a program this size; its real capabilities are best demonstrated on a large program. However, a small program will be used to make the example easier to follow. The subroutine STLINE accepts two real arrays containing the X-coordinate and Y-coordinate of a set of points, and an integer specifying the number of points. It fits a straight line through them, and returns the coefficients of the equation.

The program and data card is stored in the file "STLINE FORTRAN A". The data card follows the $ENTRY card and contains the value 10.4. The option DEBUG is specified, in order to have the program run under Debug mode, the option XTYPE is also specified to cause execution-time output to be typed at the terminal.

```
watfiv stline (xtype debug
 CMD:
n?
 UUUUUUUUUUUU
 CMD:
a(6)?
 UUUUUUUUUUUUUUU
 CMD:
```

As shown above, the lines in lower-case are the ones entered at the terminal and those in upper-case are the responses. The program is compiled, but before it begins execution, it returns control to the user, by typing CMD: (at this point, the user may enter any Debug command). The values of a few variables are displayed by typing the variable name, immediately followed by a ?, but since execution has not yet begun, everything is, of course, undefined. There is really nothing to do but let the program run, which is done by entering the command RUN.

```
run
***ERROR*** IMPROPER CHARACTER SEQUENCE OR INVALID CHARACTER IN INPUT DATA
FIRST 80 CHARACTERS OF INPUT RECORD ARE->'10.4                                '
  EXECUTION TIME ERROR. ENTER CORRECTION OR EXIT
 STOPPED AT LINE    5
 CMD:
```

The READ statement at line 5 is executed, and receives a value 10.4. This produces an error from WATFIV, because N is an INTEGER variable, and a REAL value was specifed.

```
n?
 UUUUUUUUUUUU
 CMD:
n=10
 CMD:
goto 14
***ERROR*** A DO-LOOP PARAMETER IS UNDEFINED OR OUT OF RANGE.
            NN     HAS THE VALUE -2139062144
 EXECUTION TIME ERROR. ENTER CORRECTION OR EXIT
 STOPPED AT LINE   16
 CMD:
```

The value of N is displayed and found to be still undefined, because of the error. It is assigned a value of 10, and then the READ statement is by-passed by using a GOTO 14 to transfer to the statement labelled 14, the next statement. Note that a GOTO can only transfer control to a statement

label, not an internal statement number (ISN). All other Debug commands operate on ISNs. An error at ISN 16 is then received, stating that NN is undefined or out of range.

```
nn?
 UUUUUUUUUUUU
 CMD:
nn=10
 CMD:
run
***ERROR*** VALUE OF X( 1) IS UNDEFINED
  EXECUTION TIME ERROR. ENTER CORRECTION OR EXIT
 STOPPED AT LINE   17
 CMD:
x(1)?
 UUUUUUUUUUUUUUU
 CMD:
```

The value of NN is displayed and found indeed to be undefined. By examining the program, it is determine that this was a typing error, and NN should be just N. To rectify this problem, assign NN the same value that N has, 10, and proceed.

At line 17, it is found that X(1) is undefined, and displaying its value verifies this fact. Looking at the algorithm in the mainline program for defining the array A (which gets passed to the array X in the subroutine) it can be seen that the programmer indeed botched it, and neglected to define the first element.

```
x(1)=3.98
 CMD:
x?
   0.3980000E 01  0.1052632E 02  0.2105263E 02  0.3157893E 02
   0.4210526E 02  0.5263158E 02  0.6315788E 02  0.7368420E 02
   0.8421053E 02  0.9473683E 02
 CMD:
run
***ERROR*** VALUE OF Y(10) IS UNDEFINED
  EXECUTION TIME ERROR. ENTER CORRECTION OR EXIT
 STOPPED AT LINE   20
 CMD:
```

X(1) is defined, and then the entire X array is displayed; since there are no more undefined values, execution is allowed to continue. However, a similar error has caused the tenth element of the array Y to be undefined.

```
y(0)=9.6
***ERROR*** SUBSCRIPT NUMBER 1 OF Y     HAS THE VALUE         0
  DEBUG ERROR. RE-ENTER
 CMD:
y(10)=9.6
 CMD:
stop 28
 CMD:
run
 STOPPED AT LINE   28
 CMD:
```

While trying to enter a value for Y(10), a 0 instead of a 10 was inadvertently typed. Debug informs the user of this error, and a correct value is entered. The user then decides that before leaving the subroutine, the values of A and B will be examined, so the compiler is instructed to stop when it reaches line 28, the RETURN statement, and give the user control.

```
a?
   0.1068893E 00
 CMD:
b?
  -0.4032579E 00
 CMD:
x(3)=7.89
 CMD:
y(7)=4.82
 CMD:
n=9
 CMD:
nn=9
 CMD:
stop 28
 CMD:
goto 65
 STOPPED AT LINE   28
 CMD:
a?
   0.9692568E-01
 CMD:
b?
   0.1816178E-01
 CMD:
```

The values of A and B are examined, some variables are reset, the compiler is asked to stop again at 28, and is sent back to the statement labelled 65.

```
n=7
```

```
 CMD:  nn=7
 CMD:  trace 12,999
 CMD:  goto 65
 LINE    12
 LINE    13
 LINE    14
 LINE    15
 LINE    16
 LINE    17
 LINE    18
 LINE    19
 LINE    20
 LINE    21
 LINE    22
 LINE    17
 LINE    18
 LINE    19
 LINE    20
 LINE    !
 CMD:  off
 CMD:  stop 28
 CMD:  run
 STOPPED AT LINE   28
 CMD:  a?
   0.9692568E-01
 CMD:  b?
   0.1816178E-01
 CMD:  exit

R(00004);
```

This is a simple program with no branches, but if it did have a number of GO TOs, it might be beneficial to have a TRACE of all ISNs that are executed. Some variables are reset, and WATFIV is instructed to trace all ISNs between 12 and 999, that is, the end of the program. Control is transferred to the statement labelled 65. After letting some of the trace messages print, the user decides that no further output is required, interrupts with the attention or break key, and stops tracing by saying OFF. The STOP is reset at 28, and execution continues; when the stop at 28 is encountered A and B are examined and found to be fine, so it is unnecessary to return to the main program and WATFIV Debug is left by specifying EXIT. All that remains now is to edit the original program, and make necessary changes.

5. JOB ACCOUNTING

The last three lines of output for each job are generated by the compiler and consist of certain accounting information. Specifically, the information provided is:

- the time, in seconds, taken to compile the program
- the time, in seconds, that the program executed(1)
- the amount, in bytes, of object code(2) generated for the program
- the amount, in bytes, of storage used by the program for arrays, common blocks, and equivalenced variables (the so-called 'array area')
- the total storage, in bytes, that was available for the run to contain object code and the array area
- the number of errors, warnings and extensions issued for the program
- the date and time the program finished execution
- the release date of version, and level of WATFIV in use

An example of accounting output follows:

```
CORE USAGE  OBJECT CODE = 320 BYTES, ARRAY AREA = 0 BYTES,
                          TOTAL AREA AVAILABLE = 39008 BYTES
DIAGNOSTICS NUMBER OF ERRORS = 2, NUMBER OF WARNINGS = 3,
                          NUMBER OF EXTENSIONS = 1
COMPILE TIME = 0.02 SEC, EXECUTION TIME = 1.23 SEC,
          01.26.02 SATURDAY 14 FEB 76 WATFIV - JAN 1976
```

---

(1) The time required to print out the PROFILER statistics is not included in this value.
(2) This includes constants, temporaries, non-equivalenced simple variables, save areas, any routines loaded from the object library, etc.

6. DIAGNOSTICS

6.1 ERROR DIAGNOSTICS

WATFIV issues compile-time diagnostics at three levels of severity - EXTENSION, WARNING and ERROR. A diagnostic is generated in-line in the source listing, immediately below the statement in which the condition was detected.

An EXTENSION message results if an extension of the FORTRAN language allowed by WATFIV was used. These are described in section 7.1 on page 46. The diagnostic is issued so that the problem can be eliminated, should the program be re-compiled with IBM's FORTRAN compilers.

A WARNING is issued for language violations for which the compiler can take some reasonable corrective action, e.g., truncating a name of more than 6 characters.

An ERROR is issued when a language violation severe enough to prevent execution is encountered. In this case, the compiler will normally inhibit execution of the program, unless the FREE option has been specified.

At execution time, all errors are fatal[1] in the sense that the compiler will terminate the current job and proceed to the next job in the batch. For execution-time errors, the compiler generates a diagnostic and a subprogram traceback in the printed output. This gives the line number of the statement in which the error occurred, the name of the subprogram in which the error occurred, the name of the subprogram which called it, etc., all the way back to the main program which is referred to as M/PROG. (The line number of each statement appears to the left of it in the source listing. This line number is compiler generated, and is distinct from, and should not be confused with, any FORTRAN statement number the programmer may have assigned to a statement).

---

(1) Exception: If a hardware I/O error occurs and the programmer has specified an ERR= return in the affected I/O statement, an error message is given and execution proceeds at the statement specified by the ERR=.

Example of a traceback:

```
***ERROR***     VALUE OF A IS UNDEFINED
PROGRAM WAS EXECUTING LINE 15 IN ROUTINE RTN2   WHEN TERMINATION OCCURRED
PROGRAM WAS EXECUTING LINE  9 IN ROUTINE RTN1   WHEN TERMINATION OCCURRED
PROGRAM WAS EXECUTING LINE  4 IN ROUTINE M/PROG WHEN TERMINATION OCCURRED
```

One of the design goals of WATFIV is to supply good diagnostics. We, the implementors, think the goal has been well met, but, sad to say, we have heard that a few users of the compiler at our installation have found some of the diagnostic messages to be vague, obfuscatory, or hubristic.

It is hoped that the following paragraphs will simplify, for the user, the interpretation of some of the error messages which may, at present, be too brief or may contain special words with meanings entirely clear only to the compiler implementors.

The user should be aware that an error in one statement may lead to apparent errors in subsequent statements. The case may be that, if the first error is corrected, the others will disappear as well on a subsequent compilation. This is particularly true if the first error occurred in a specification statement. The reason is that the compiler scans each source statement, column by column from left to right, and usually abandons compilation of a statement when a syntax error is encountered. Thus, correct information in a statement may be ignored if it follows a column which contained an error.

Consider the following program as an example.

```
DIMENSION A(10),B(104+C(10)
C(1)=2
  .
  .
  .
```

Both the first and second statements will be flagged with error messages - the first since there is no matching parenthesis for the dimension of B; the second since the compiler, lacking knowledge that C is an array because of the previous error, assumes that the second statement is a definition of a statement function C. (Statement function definitions must have variable names, not constants, as dummy arguments). The second error will disappear when the first error has been corrected.

The point is that the programmer, when confronted with an error message, must do some analysis to see if it is a real

error, or merely an apparent error arising from an error in a previous statement.

Certain of the error messages generated by the compiler imply a knowledge, on the programmer's part, of the left to right scan of statements. These messages usually relate to the syntax of statements, and contain the word 'expecting', for example, the statement

```
DIMENSION+A(10)
```

is flagged with the message

```
EXPECTING SYMBOL, BUT + BEFORE A WAS FOUND
```

This implies that the compiler, scanning the statement from left to right, expected to find a symbol after the keyword DIMENSION in order to consider the statement syntactically correct according to the rules of FORTRAN.

The following glossary is provided to define some terms which appear in the WATFIV diagnostics and which may not have a 'standard' or accepted meaning to FORTRAN programmers.

FORTRAN keyword - a word, such as STOP, READ, GOTO that identifies a FORTRAN statement.

Program Segment - a subroutine or function subprogram, or a main program.

Simple Variable - a variable which is not an array

ASSIGNED GOTO Index - a variable used in an ASSIGN statement or ASSIGNED GOTO statement, e.g., I is an ASSIGNED GOTO Index in the following statement.

```
ASSIGN 5 to I
```

Statement Number Constant - &5 is a statement number constant in the following statement.

```
CALL SUBR(X,&5)
```

Operator - usually an arithmetic operator such as '+', '-', etc., but generally any delimiter, e.g., '(', '&', ',', etc.

End-of-Statement - the implied end-of-statement operator that the compiler expects to find at the end of a correct statement.

Symbol - a symbolic name, i.e., the name of a variable, array, subprogram, etc.

Temporary - a value which is the result of evaluating an expression. For example, 3.*A+2. is a 'temporary' in the following statement.

```
CALL RTN (3.*A+2.)
```

Argument - a value passed to a subprogram. For example, A, 3.5, SIN(X) are arguments in the following statement.

```
CALL SP1(A,3.5,SIN(X))
```

Parameter - a symbolic value used in a subprogram and which is replaced by a real argument when the subprogram is referenced at execution time; sometimes called 'dummy arguments' by other authors. For example, A and B are parameters in the following statement.

```
SUBROUTINE EGGMOR(A,B)
```

DO-loop Parameter - a simple integer variable or integer constant used to control the number of times a DO-loop is performed. For example: I,3,J,2 are DO-loop parameters in the following statement.

```
D0 17 I=3,J,2
```

Object of a DO - the last statement of a DO-loop. The statement numbered 15 is the object of the DO-loop defined by the statement numbered 7 in the following example.

```
 7  DO  15   I=3,J,2
          .
          .
          .
15  X(I) = A(I)*B(I)
```

Dimension - a value used to declare the maximum value that a subscript of an array may assume at execution time. For example, 10, 15, and 5 are dimensions of A in the following statement.

```
DIMENSION A(10,15,5)
```

Subscript - a value used to refer to a member of an array. For example, I, 7, and 3*K+12 are subscripts of A in the following statement.

```
Y=A(I,7,3*K+12)
```

Type - this usually refers to one of the types LOGICAL, INTEGER, REAL, COMPLEX, (and with WATFIV), CHARACTER. However, it may refer to a particular sub-type. For example, the following statements define X to have type REAL*4, A to have type REAL*8, and Z to have type LOGICAL.

```
REAL X*4,A*8
LOGICAL Z
```

Mode - this generally refers to the usage of a symbolic name within a subprogram, or a program as a whole. By usage, we mean variable name, common block name, sub-program name, etc. The name AB has mode 'common block' in the statement

```
COMMON /AB/X,Y,Z
```

Sometimes it may include type as well, e.g., the symbolic name FN has mode 'REAL*8 function subprogram' in the following example.

```
REAL FUNCTION FN*8 (A,B)
```

Defined - at compile time, we say the mode and/or type of a symbolic name is defined when there is no longer any doubt what its mode and/or type might be. The mode and/or type can be established explicitly from information in specification statements that refer to the symbolic name, or implicitly from the first use of the name in a program segment. Once the mode and/or type of a name have been

defined, they may not be redefined. Consider the following sequence of statements:

```
REAL I, J(10),K,L*8/1.D0/
DIMENSION I(5)
EXTERNAL K
M=L + FN(I)
```

The first statement defines the type of all four names, I, J, K, L. Furthermore, it also defines the modes of names J and L. J is explicitly identified as an array, and L is assumed to be a simple variable since it is initialized and initialization constitutes a use of a name. The second and third statements explicitly define the modes of names I and K as array and subprogram, respectively. The fourth statement implicitly defines the mode and type of names M and FN since they are used in that statement. Since this is their first use or appearance in the program, their types are determined from the usual FORTRAN first letter rule, and their modes are established from their usage - M is a simple integer variable, FN is a REAL*4 function.

At execution time, a variable or array element or function name is defined if it has been assigned a value.

Undefined - at execution time, a variable or array element is said to be undefined if it has not had a value assigned to it. For example, if the statement

```
X=Y
```

were the first statement of a main program, then, at execution time, Y would be undefined since there would be no way it could have had a value assigned to it.

WATFIV will check your program at execution time for attempts to use undefined variables unless you specify NOCHECK on the $JOB card or C$OPTIONS card.

NOTES:

1. The authors of the compiler do not advocate the use of the FREE option; it is provided for those programmers who feel it is desirable to obtain some execution-time output, even from a program which may contain serious compile-time errors. Note that some errors are of such a serious nature that execution will be inhibited even if FREE is specified, e.g., if memory space cannot be allocated to contain arrays declared in the program.

2. Under CHECK or FREE, the compiler will terminate your job if you use an undefined variable in an expression, i.e., if you attempt some evaluation that involves a variable that has not been assigned a value. However, the compiler will allow you to print undefined values without terminating your program. Such values appear on the page as a string of U's.

For example, if the statements

```
I=1
K=2
PRINT,I,J,K
```

were the first to be executed in a program, the line of output produced by the PRINT statement would appear as

```
1 UUUUUUUUUUUU     2
```

Note that U's are still printed for undefined variables even under NOCHECK. NOCHECK suppresses only the check for attempted use of undefined variables in the evaluation of expressions.

3. EXTENSION and WARNING messages may be suppressed from the source listing by specifying NOEXT and NOWARN as $JOB card parameters. It is a good practice not to suppress these diagnostics in the initial stages of debugging a program.

4. Section 15.1 on page 108 of this manual contains a complete list of all diagnostics that the WATFIV compiler can produce.

5. The following compiler-generated names appear in some diagnostics.

```
M/PROG - name of the main program
//     - name of the blank common block
```

6.2 CONTROL OPTIONS FOR CERTAIN DIAGNOSTICS

Six control options are available to control the printing and generation of certain diagnostics. The WARN and NOWARN options control the printing of compiler-generated warning and extension messages; the EXT and NOEXT options control the printing of compiler-generated extension messages; the CHECK and NOCHECK options control the compiler's checking of undefined variables.

If the compiler encounters the NOWARN option in the source deck, all warning messages will be suppressed from that point on. A C$OPTIONS WARN card will allow the warning messages to be restarted if the NOWARN option was punched on the $JOB or C$OPTIONS card. The generating of extension messages may be controlled in a similar way by the EXT and NOEXT options.

When the NOCHECK option is encountered by the compiler, it bypasses the generation of object code that checks for undefined variables at execution time. A C$OPTIONS CHECK card causes the compiler to generate the checking code if NOCHECK was specified (or defaulted) as a $JOB or C$OPTIONS card parameter.

The source deck using these cards might be as follows:

```
$JOB   D9999JOE.USER,NOWARN

      compile with "CHECK"

      no warning messages

C$OPTIONS NOCHECK,EXT

      compile with "NOCHECK" and print extension messages

         END

C$OPTIONS CHECK,WARN

      subprograms

      compile with "CHECK" and print warning messages

         END

$ENTRY

      data cards
```

These options allow local control of their function. This can be useful if a program is being debugged in stages, with routines being added or changed over a sequence of runs. If the NOCHECK option can be used because a segment of a program is known to be free of undefined variables, several advantages can result:

- less object code is generated; thus, a somewhat larger program can be compiled for a given amount of available memory.

- the program will run somewhat faster since the checking code is not executed.

6.3 WATFIV DEBUGGING AIDS

6.3.1 EXECUTION-TIME PROFILER

WATFIV is a powerful tool for the writing and debugging of FORTRAN programs. The diagnostic messages and debugging aids provided have helped get programs into the "execution phase" with a great saving in programmer time. However, once the program has been successfully debugged and tested and is ready for use in the "production" stage, little or no feedback is available on what it is doing. The PROFILE option of WATFIV is an attempt to address this critical area of programming and should be used to find bottlenecks and deficiencies once the program is working. This new WATFIV feature will provide information concerning what segments of the program are being executed most often. The output from the WATFIV PROFILER provides a frequency count of the number of times each statement was executed, and a histogram scaled in percent or relative count.

The new options available under WATFIV are as follows:

PROFC	Turns on the profile count option
PROFP	Turns on the histogram percentage option
PROFA	Turns on the histogram absolute count option
PROF	Equivalent to specifying PROFC PROFP, and PROFA

The above options may appear on the $JOB card or on the C$OPTIONS card. At least one of these options must be specified for formatting the output of the WATFIV PROFILER. The actual enabling of the PROFILER is controlled by two new control cards which are inserted around any group of executable statements. Only statistics on these statements will be displayed in the PROFILER output. These control cards are:

C$PROFON	Turns on execution-time count facility
C$PROFOFF	Turns off execution-time count facility

If the C$PROFOFF card is omitted, then performance monitoring will be done until the end of the program.

It is often the case that over 50% of the execution time of a program is spent in less than 10% of the source statements. Up to now this critical area of a program has been extremely hard to pinpoint; the area of software monitoring has little room for intuition. With the introduction of the WATFIV PROFILER, programmers will be able to receive some measurement feedback.

By examining the PROFILER histogram in conjunction with all

control statements, a restructuring of the program can be done to minimize the number of frequently taken branches. Programs operating in a paging environment will benefit by adhering to this "locality of reference" concept.

The output from the WATFIV PROFILER only indicates frequency count (execution time would be too compiler dependent). Those statements with a high frequency count should be examined to determine if a different algorithm or data structure can be used to decrease some execution-time overhead.

The following programming techniques have been adapted from the STANFORD University Fortune User's Guide and can be applied when evaluating the output from the PROFILER:

1) Test for most probable cases first so the execution of certain IF statements can be eliminated.

2) Commonly referenced expressions should be calculated and stored in variables so that execution time is not consumed in recalculating these values.

3) Multi-dimensional arrays should be equivalenced to vectors to avoid complicated subscripting algorithms (internal to the compiler).

4) Certain subroutines should be made in-line to eliminate the calling sequence overhead.

Another advantage of the PROFILER is the ability to verify the existence of good test data. The list of statements not executed can be examined and test data reconstructed to ensure that this code is executed. The presence of a block of statements which is never executed indicates the possibility of a bug in a working program.

The usefulness of this new facility is demonstrated by the following programs. A brief guide to interpreting the PROFILER output follows the listings.

```
    $JOB          *************
    C
    C
    C     FORTRAN FACTORIAL FREQUENCY FOLLY
    C
    C
    C$PROFON
    C$OPTIONS PROFP,PROFA,L=0
1         INTEGER FACTOR/2/
2         INTEGER KNT/0/
3         DO 100 I=3,1000
4             FACTOR=FACTOR* (I*(I-1))
5             IF( FACTOR .LT. 0)KNT=KNT+1
6   100   CONTINUE
7         STOP
8         END

    $ENTRY

              WATFIV PROGRAM PROFILE

    1997  STATEMENT(S) EXECUTED
       6  SECONDARY STATEMENT(S) EXECUTED
       0  STATEMENT(S) NOT EXECUTED

                                    HISTOGRAM OF PERCENTAGE FREQUENCY COUNT
    STMT     COUNT 0%       10%       20%       30%       40%       50%       60%       70%       80%       90%       100%
       3         1 *         .         .         .         .         .         .         .         .         .         .
       4       998 **************************************************         .         .         .         .         .
       5       998 **************************************************         .         .         .         .         .
  OBJECT         6 *         .         .         .         .         .         .         .         .         .         .
            2003 TOTAL STATEMENT(S) EXECUTED


                                    HISTOGRAM OF ABSOLUTE FREQUENCY COUNT
    STMT     COUNT
                   1       101       201       301       401       501       601       701       801       901      1001
       3         1 *         .         .         .         .         .         .         .         .         .         .
       4       998 ****************************************************************************************************
       5       998 ****************************************************************************************************
  OBJECT         6 **        .         .         .         .         .         .         .         .         .         .
            2003 TOTAL STATEMENT(S) EXECUTED

CORE USAGE       OBJECT CODE=     456 BYTES,ARRAY AREA=       0 BYTES,TOTAL AREA AVAILABLE=   51200  BYTES

DIAGNOSTICS        NUMBER OF ERRORS=       0, NUMBER OF WARNINGS=       0, NUMBER OF EXTENSIONS=       0

COMPILE TIME=     0.05 SEC,EXECUTION TIME=     0.09 SEC,     14.22.11    WEDNESDAY     7 APR 76    WATFIV - JAN 1976 V1L5
```

```
    $JOB           *************
    C$OPTIONS  PROFC,PROFA,L=0
    C$PROFON
 1        CHARACTER*1 CARD(200,80),NCHAR1(5)/'@','<','%','&','#'/
 2        CHARACTER NCHAR2(5)/'''',')','(','+','='/
 3        CHARACTER KARD(200,80)
 4        INTEGER TCHARS
 5        TCHARS=0
 6        DO 20 I1=1,200
 7  10        READ(5,920)(CARD(I1,K),K=1,80)
 8            AT END DO
 9                LL=I1-1
10                WRITE(6,940)TCHARS
11                WRITE(8,930)((CARD(L,KK),KK=1,80),L=1,LL)
12  940           FORMAT(' THE TOTAL NUMBER OF NON-BLANK CHARACTERS IS',I5)
13                STOP
14  930           FORMAT(1X,80A1)
15  920           FORMAT(80A1)
16            END AT END
17            IF(CARD(I1,1) .NE. 'C')GOTO 25
18            LL=I1
19            DO 30 J=1,80
20                DO 30 K=1,LL
21  30        KARD(K,J)=CARD(K,J)
22            GO TO 10
23  25        DO 20 I2=1,80
24                IF(CARD(I1,I2).EQ.' ')THEN DO
25                    GO TO 20
26                ELSE DO
27                   TCHARS=TCHARS+1
28                END IF
29                I3=1
30                WHILE (I3.LE.5) DO
31                    IF(CARD(I1,I2).EQ.NCHAR1(I3))THEN DO
32                        CARD(I1,I2)=NCHAR2(I3)
33                        I3=99
34                    ELSE DO
35                        I3=I3+1
36                    END IF
37                END WHILE
38  20    CONTINUE
39        STOP
40        END

        $ENTRY
THE TOTAL NUMBER OF NON-BLANK CHARACTERS IS 6116
```

WATFIV PROGRAM PROFILE

```
   105471  STATEMENT(S) EXECUTED
      164  SECONDARY STATEMENT(S) EXECUTED
        5  STATEMENT(S) NOT EXECUTED
```

TABLE OF FREQUENCY COUNT

FROM	TO	COUNT	FROM	TO	COUNT	FROM	TO	COUNT	FROM	TO	COUNT
5	6	1	7	7	165	9	11	1	17	17	164
23	23	164	24	24	13120	25	25	7004	27	27	6116
28	ENDIF	6116	29	29	6116	30	30	26559	30	ENDWHILE	6116
31	31	20443	32	33	5172	34	ELSEDO	5172	35	35	15271
36	ENDIF	20443	38	LEVEL 2	164						

THE FOLLOWING STATEMENTS WERE NOT EXECUTED

FROM	TO	FROM	TO	FROM	TO	FROM	TO	FROM	TO
18	21	21	LEVEL 2	21	LEVEL 1	22	22	26	ELSEDO
38	LEVEL 1								

HISTOGRAM OF ABSOLUTE FREQUENCY COUNT

```
STMT    COUNT
              0      2656      5312      7968     10624     13280     15936     18592     21248     23904     26560
    5         1 *         .         .         .         .         .         .         .         .         .         .
    6         1 *         .         .         .         .         .         .         .         .         .         .
    7       165 **        .         .         .         .         .         .         .         .         .         .
    9         1 *         .         .         .         .         .         .         .         .         .         .
   10         1 *         .         .         .         .         .         .         .         .         .         .
   11         1 *         .         .         .         .         .         .         .         .         .         .
   17       164 **        .         .         .         .         .         .         .         .         .         .
OBJECT      164 **        .         .         .         .         .         .         .         .         .         .
   23       164 **        .         .         .         .         .         .         .         .         .         .
   24     13120 ************************************************.         .         .         .         .         .
   25      7004 **************************                      .         .         .         .         .         .
   27      6116 ************************        .         .         .         .         .         .         .
   29      6116 ************************        .         .         .         .         .         .         .
   30     26559 ********************************************************************************************************
   31     20443 ***************************************************************************  .         .         .
   32      5172 ********************.         .         .         .         .         .         .         .
   33      5172 ********************.         .         .         .         .         .         .         .
   35     15271 ************************************************************  .         .         .         .
         105635 TOTAL STATEMENT(S) EXECUTED
```

CORE USAGE OBJECT CODE= 6696 BYTES,ARRAY AREA= 32010 BYTES,TOTAL AREA AVAILABLE= 153600 BYTES

DIAGNOSTICS NUMBER OF ERRORS= 0, NUMBER OF WARNINGS= 0, NUMBER OF EXTENSIONS= 7

COMPILE TIME= 0.20 SEC,EXECUTION TIME= 9.17 SEC, 20.38.04 WEDNESDAY 7 APR 76 WATFIV - JAN 1976 V1L5

```
    $JOB           *************
    C     PALINDROMIC PROFILER PROGRAM
    C$PROFON
    C$OPTIONS PROF,L=0
 1        DO 10 I=1,100
 2          DO 10 J=1,100
 3            DO 10 K=1,100
 4              LL=I+J*K
 5  10    CONTINUE
 6        STOP
 7        END

    $ENTRY

               WATFIV PROGRAM PROFILE

  1010101  STATEMENT(S) EXECUTED
        0  SECONDARY STATEMENT(S) EXECUTED
        0  STATEMENT(S) NOT EXECUTED


                                    TABLE OF FREQUENCY COUNT

FROM       TO    COUNT          FROM       TO    COUNT          FROM       TO    COUNT          FROM       TO    COUNT
   1        1        1             2        2      100             3        3    10000             4        4  1000000
   5  LEVEL 3    10000             5  LEVEL 2      100             5  LEVEL 1        1


                                  HISTOGRAM OF PERCENTAGE FREQUENCY COUNT

    STMT    COUNT 0%       10%       20%       30%       40%       50%       60%       70%       80%       90%       100%
       1        1 *         .         .         .         .         .         .         .         .         .         .
       2      100 *         .         .         .         .         .         .         .         .         .         .
       3    10000 **        .         .         .         .         .         .         .         .         .         .
       4  1000000 **********.*********.*********.*********.*********.*********.*********.*********.*********.*********.
          1010101 TOTAL STATEMENT(S) EXECUTED


                                  HISTOGRAM OF ABSOLUTE FREQUENCY COUNT

    STMT    COUNT
                   1    100001    200001    300001    400001    500001    600001    700001    800001    900001   1000001
       1        1 *         .         .         .         .         .         .         .         .         .         .
       2      100 *         .         .         .         .         .         .         .         .         .         .
       3    10000 **        .         .         .         .         .         .         .         .         .         .
       4  1000000 **********.*********.*********.*********.*********.*********.*********.*********.*********.*********
          1010101 TOTAL STATEMENT(S) EXECUTED

CORE USAGE       OBJECT CODE=     416 BYTES,ARRAY AREA=       0 BYTES,TOTAL AREA AVAILABLE=   51200  BYTES

DIAGNOSTICS        NUMBER OF ERRORS=       0, NUMBER OF WARNINGS=       0, NUMBER OF EXTENSIONS=       0

COMPILE TIME=     0.05 SEC,EXECUTION TIME=    63.76 SEC,     14.28.40    WEDNESDAY     7 APR 76    WATFIV - JAN 1976 V1L5
```

The number of statements executed is the actual count of executable FORTRAN statements -- control statements, arithmetic and logical IF statements, I/O statements, and assignment statements. Certain statements are deemed to be non-executable by the WATFIV PROFILER. Some of these are CONTINUE, STOP, END, DATA, FORMAT, DEFINE FILE, and specification statements. Thus, they will not show up in the PROFILER output (an END statement generated by WATFIV or STOP as the object of a logical IF statement will show up in the PROFILER count).

The number of secondary statements executed includes a count for the number of times the OBJECT of a logical IF or WHILE EXECUTE statement was executed. The final line of output includes the total number of unexecuted statements encountered while the C$PROFON card was in effect.

The output of the PROFILER can consist of a variety of formatted reports. The PROFC option will produce a statement flow with a separate count for each contiguous block of source statements. This will consist of the initial WATFIV ISN (Internal Statement Number) under the FROM heading and the final ISN under the TO heading. Where the object of an IF or WHILE EXECUTE statement is not executed the same number of times as the initial IF or WHILE EXECUTE clause, a count break condition is set and a separate FROM, TO, and COUNT field is displayed. This will show up as the ISN under the FROM field and the OBJECT or WHILE EXECUTE under the TO field. This is because the FORTRAN language considers a "logical IF" to be two separate statements. The PROFILER will then pinpoint the area where it is advantageous to reprogram the condition being checked by the logical expression. Similarly, all statements terminating a DO loop structure will cause a count break and the FROM and TO fields will contain the ISN terminating the DO loop and the name of the DO construct. For a standard DO loop the name will show up as LEVELn, where n is the level of nesting starting with the outermost loop as level 1. For structured WATFIV language constructs, the name will be of the form ENDWHILE, ELSEDO, or ENDIF. The COUNT field contains the number of times the program fell through the loop normally. The ENDIF count will include the number of times the ELSE DO clause made a normal exit.

In closing, here is a practical approach to using the PROFILER.

1) Write and debug the program.

2) Once all "known" bugs are eliminated, use the PROFILER facilities to find which routines contain most of the statements being executed.

3) Concentrate on these few statements and see if they can be written more efficiently or reprogrammed in a completely different manner.

6.3.2 STATEMENT TRACE FACILITY

An execution-time statement trace or "ISN trace" feature may be invoked. The trace is turned on using a C$ISNON card and is turned off using a C$ISNOFF card. At least one executable statement must precede a C$ISNON. A sample program follows:

```
$JOB    id,parameters
        A=1
        J=3
C$ISNON
        (statements to be traced)
C$ISNOFF
        STOP
        END
$ENTRY
```

6.3.3 ON ERROR GOTO STATEMENT

The ON ERROR GOTO statement allows a program which has an error to recover and take some alternate and possibly corrective action, such as giving a diagnosis. This feature can only be executed once in a program (to prevent infinite loops) however, any number of ON ERROR GOTO statements may appear in the source program. The last ON ERROR GOTO statement encountered before an error occurs is the one which is executed.

A program using this feature might be as follows:

```
    $JOB  id,parameters
          ON ERROR  GOTO 50
          I=0
    5     READ(5,*,END=40)A
          I=I+1
          PRINT,A
          GO TO 5
    50    PRINT,'CARD NUMBER', I, 'IS INVALID'
    40    STOP
          END

    $ENTRY
```

The ON ERROR GOTO statement is not an executable statement;

it may be placed anywhere in the program. However, it is not advisable to place the object of an ON ERROR GOTO statement within the range of a DO-loop as no checking is performed to determine if the transfer at execution time will be valid (i.e., infinite looping may result).

7. LANGUAGE ACCEPTED BY WATFIV

WATFIV attempts to support the language described in the IBM publication "IBM System/360 FORTRAN IV Language", form GC28-6515,[1] subject to the restrictions given below in subsection 7.1.5. In addition, WATFIV supports a number of extensions to the language, which are described in section 7.1 below.

7.1 EXTENSIONS

Uses of the following language extensions, except for 1, 2, 12, 13, 14, 15 and 17 are flagged with *EXTENSION* messages. These mean that the program is acceptable to WATFIV but will not likely compile on other compilers. The messages can be suppressed by the use of the NOEXT parameter on the $JOB or C$OPTIONS card. (See section 3.1.1 on page 6.)

7.1.1 FORMAT-FREE INPUT OUTPUT

This allows the programmer to do I/O without reference to a FORMAT statement. For example, the statement

```
PRINT, A,B
```

will cause the values of A and B to be printed with a standard format. Section 7.2 on page 51 describes format-free I/O in more detail.

7.1.2 CHARACTER VARIABLES

This variable allows the manipulation of data in the form of character strings. As a byproduct, in-core formatting of data may be performed. See Chapter 8 on page 57 for complete details.

A simple example of the use of a CHARACTER variable follows:

---

(1) The current level of WATFIV (V1L5 Jan/76) corresponds to the -10 version of GC28-6515.

```
CHARACTER  A*7
   .
   .
   .
A='FINALLY'
   .
   .
   .
```

7.1.3 MULTIPLE ASSIGNMENT STATEMENTS

Statements of the form

v1 = v2 = ...= vn= expression

are allowed, where v1, v2, etc., represent variable names or array elements. The effect is that of the sequence of statements

```
vn= expression
vk= vn

  .
  .
  .

v1= v2
```

e.g., A = B(5) = C = 1.5

7.1.4 EXPRESSIONS IN OUTPUT LISTS

Expressions may be placed in output statements, e.g.,

```
WRITE(6,2) SIN(X)**2,A*X+(B-C)/2
```

The expression may not, however, start with a left parenthesis because the compiler uses this as a signal that an implied DO follows in the list. For example:

```
PRINT, (A+B)/2
```

would result in an error message. However, the equivalent

PRINT, +(A+B)/2 is acceptable.

Note that CHARACTER constants are forms of expressions acceptable in output statements, e.g.,

```
PRINT,'VALUE OF X=',X
```

7.1.5 INITIALIZING OF BLANK COMMON

Variables in blank common may be initialized in DATA or type statements, e.g.,

```
COMMON X
INTEGER X/3/
```

7.1.6 INITIALIZING COMMON BLOCKS

Common blocks may be initialized in other than BLOCK DATA subprograms.

7.1.7 IMPLIED DO IN DATA STATEMENTS

Implied DOs are allowed in DATA statements, i.e., a statement of the form

```
DATA  (C(I), I=1,5,2)/3*.25/
```

is valid.

In fact,

```
DATA (A(I), I=L,M,N)/ constant list/
```

is acceptable if L,M,N have been previously initialized and at least MOD(M-L,N)+1 constants are present in the constant list.

7.1.8 SUBSCRIPTS IN FUNCTION DEFINITIONS

Subscripts may be used on the right-hand side of the statement function definitions, e.g.,

```
F(X) = A(I)+X+B(I)
```

7.1.9 SUBSCRIPT USAGE

The real part of a complex value is converted to an integer, and this value is used for indexing into the array. For example, if Z is complex, and A is an array, then A(Z) is equivalent to A(INT(REAL(Z))).

For rules and examples of logical and character values as subscripts, see subsection 8.4.1 on page 69.

7.1.10 OBJECT OF DO STATEMENT

A logical IF statement used as the last statement (object) of a DO loop may contain a GOTO of any form, a PAUSE, STOP, RETURN, or arithmetic IF statement.

```
      e.g.,          DO 25 I=1,N
                     .
                     .
                     .
                25   IF (X.EQ.A(I)) RETURN
```

7.1.11 EXCEEDING CONTINUATION CARD LIMIT

A statement may be continued over more cards than is allowed by the, so-called, continuation card limit '[1]'.

7.1.12 MULTIPLE STATEMENTS PER CARD

WATFIV allows the programmer to punch more than one statement on a single card. This is particularly suitable for programs that are to be stored on libraries since less direct-access storage space is required, and fewer input operations are necessary to retrieve a subprogram.

The rules for this feature are:

(a) Only columns 7-72 may be used for statements.

(b) A semi-colon is used to indicate the end of a statement.

(c) The normal continuation card rules are used for a statement which is to be continued beyond column 72.

(d) Statement numbers appear in column 1-5, as usual, or following a semicolon and followed by

---

(1) The continuation card limit is installation dependent. The value currently in use is 5.

a colon. They may not be split onto a continuation card.

(e) Comment cards and FORMAT statements must be punched in the conventional manner.

```
            Column 6
                   |
                   V

e.g.,       25       A=B;C=D;39:PRINT A,B,
                    * C,D;X=A+B*C+D
                      PRINT, X; 99: STOP;END
```

This could be punched in the conventional manner as:

```
25   A=B
     C=D
39   PRINT,A,B,C,D
     X=A+B*C+D
     PRINT,X
99   STOP
     END
```

7.1.13 COMMENTS ON FORTRAN STATEMENTS

The compiler terminates the left-to-right scan of a particular card when a □ (pronounced 'zigamorph', and punched as a 12-11-0-7-8-9 multi-punch) is encountered. Effectively, this means comments may follow a FORTRAN statement on the same card if a □ is used to terminate the FORTRAN statement.

Note that a □ is unprintable, as well as being almost unpunchable.

```
e.g.,    X=A+SIN(Y) □ EVALUATE X
```

7.1.14 DUMPLIST STATEMENT

The DUMPLIST statement is designed especially as a program debugging aid; it is used as follows:

(i) The DUMPLIST statement is essentially a NAMELIST statement, except that the word DUMPLIST replaces the word NAMELIST. The usual rules for NAMELIST statements apply. Sample statements are:

```
DUMPLIST /XXX/A,XYZ,APE/LOK/XX,NEXT
DUMPLIST /THIS/N,TWO,SIX,OLD
```

(ii) A DUMPLIST list name need never appear in a READ or WRITE statement.

(iii) A DUMPLIST statement has no effect unless the program in which it appears is terminated because of an error condition; then, WATFIV will automatically generate NAMELIST-like output of all DUMPLIST lists appearing in program segments which have been entered. The values printed are those which the variables had when the program was terminated.

To avoid producing too much output, only a few key variables should be placed in DUMPLIST statements.

7.1.15 ON ERROR GOTO STATEMENT

The ON ERROR GOTO statement was introduced to allow a program to recover from a software error and possibly take some corrective action. This statement is described in more detail in subsection 6.3.3 on page 44.

7.1.16 PSEUDO-VARIABLE DIMENSIONING

WATFIV recognizes the dimensions of an array from the main line program when that array is used as a subprogram argument and the final dimension specified in the subprogram is 1. See section 12.6 on page 95 for a complete description.

7.1.17 STRUCTURED PROGRAMMING STATEMENTS

A number of new control statements have been added to WATFIV to enable the FORTRAN programmer to design and write programs in a structured manner. See subsection 9 on page 71 for a complete description of these statements.

7.2 FORMAT-FREE INPUT OUTPUT

Format-free I/O is a programming convenience for at least the two following reasons:

- learning and inexperienced programmers can defer the use of FORMAT statements until some experience and confidence have been gained in FORTRAN, yet programs involving I/O can be attempted early on.

- experienced programmers will find format-free output

statements convenient for producing debugging output without having to bother with coding associated FORMAT statements.

7.2.1 SOURCE STATEMENT FORMS

Format-free I/O has been implemented in WATFIV to function with statements of the forms:

```
READ, list
PRINT, list
PUNCH, list
READ(unit,*,END=m1,ERR=m2) list
WRITE(unit,*) list
```

The I/O for the first three forms is done on the standard reader, printer, and punch units, i.e., 5,6,7, respectively. The asterisks in the last two forms imply format-free I/O, and 'unit' may be a constant, integer variable or a character variable. The END and ERR returns are optional, as with the convenitonal READ statement.

Note that the two statements

```
READ,list
READ(5,*)list
```

are equivalent, as are

```
PRINT,list
WRITE(6,*)list
```

Some examples follow:

```
      READ,A,B,  (X(I),I=1,N)
      PRINT,(J ,Z(J),J=N,K,L),I,P
   99 WRITE(6,*)'DEBUG OUTPUT',99,X,Y,Z+3.5
      READ(I,*,END=27) (X(J),J=1,N)
      PUNCH,'X=' ,X
```

7.2.2 INPUT DATA FORMS

Data items may be punched one per card, or many per card; in the latter case, the data items must be separated by a comma and/or one or more blanks. The first data item on a card need not start in column 1. A data item may not be continued across two cards, i.e., the end of a card acts as a delimiter.

Successive cards are read until enough items have been found to satisfy the requirements of the 'list' part of the statement. Any items remaining on the last card read for a particular READ statement will be ignored since the next READ statement executed will cause a new card to be read.

It is perfectly valid to use format-free READ statements and conventional READ statements in the same program.

The forms of data items which may be used for the various types of FORTRAN variables are:

Integer	- signed or unsigned integer constant
Real	- signed or unsigned real constant in F, E, or D forms
Complex	- 2 real numbers enclosed in parentheses and separated by a comma, e.g., (1.2,-3.8)
Logical	- a string of characters containing at least one T or F. The first T or F encountered determines the logical value.
Character	- a string of characters enclosed by quotes. If a quote is required as input,two successive quotes should be punched. Section 14.3 on page 104 describes the use of the EBCDIC and BCD quotes.

The type of data item must match the type of the variable it is being read into.

A duplication factor may be given to avoid punching the same constant many times. For example, if we have

```
DIMENSION A(25)
READ,A
```

the data for the READ statement could be punched as

```
15*0.,10*-3.8
```

Examples:

(i)	source statement	READ,X,I,Y,J
	typical data	2.5 3,-7.9,-41
(ii)	source statements	COMPLEX Z(5)

```
                                  READ, (Z(I),I=1,3)
            typical data          (5.2,-16.0) 2*(0.,.5E-3)

      (iii) source statements     LOGICAL L1,L2,L3
                                  READ,L1,L2,L3
            typical data          T    .FALSE. , CAT

      (iv)  source statements     CHARACTER  A*1, B*3
                                  READ,A,B
            typical data          'A','DOG'
```

7.2.3 OUTPUT FORMS

The compiler supplies formatting for list items output by format-free statements. Line overflow is automatically accounted for, i.e., several records may result from one output statement.

The formats used are:

Integer - I12

Real*4 - E16.7

Real*8 - D28.16

Complex*8 - '(' E16.7 ',' E16.7 ')'

Complex*16 - '(' D28.16 ',' D28.16 ')'

Logical - L8

Character*n - An

7.3 RESTRICTIONS

The user of WATFIV should take note of the following restrictions in language and facilities provided by the compiler.

1. The name of a common block must be unique, i.e., it may not also be used as the name of a variable, array, or statement function. This is in violation of GC28-6515.

2. The service subprograms DUMP and PDUMP defined in Appendix C of GC28-6515 are not supported.

3. The Debug Facility described in Appendix E of GC28-6515 is not supported.

4. There are no facilities in WATFIV which correspond to the IBM FORTRAN options MAP, EDIT, XREF, OPT=, DECK, LOAD, NAME=, LIST.

5. The Extended-Error Handling facility (available with IBM FORTRAN is not supported.

6. No overlay facility is available; no 'module map' is produced.

7. The number of continuation cards and the use of operator messages with STOP and PAUSE statements are installation options.

8. No more than 255 DO statements are allowed in a program segment.

9. FORMAT(is a reserved character sequence when used as the first 7 characters of a statement. It is the only reserved character sequence. For example,

```
FORMAT(I) = 3.5
```

   will result in FORMAT error messages, whereas

```
X=FORMAT (I)
```

   is legal, assuming FORMAT to be an array or function name.

10. WATFIV is a 'one-pass' compiler, and requires several restrictions on statement ordering. These are:

    (a) Specification statements referring to variables used in NAMELIST or DEFINE FILE statements must precede the NAMELIST or DEFINE FILE statements.

    (b) COMMON or EQUIVALENCE statements referring to variables used in DATA or initializing type statements must precede the DATA or initializing statements.

e.g., REAL I/5.2/
COMMON I

will produce error messages, whereas,

```
COMMON I
REAL I/5.2/
```

is acceptable.

(c) A variable may appear in an EQUIVALENCE statement and then in subsequent explicit type statement only if the type statement does not declare the length of the variable to be different than could be assumed for it, based on the first letter of the variable name, at the time of its appearance in the EQUIVALENCE statement.

For example,

```
EQUIVALENCE (A,B)
REAL*8 B
```

will produce an error message, whereas,

```
REAL*8 B
EQUIVALENCE (A,B)
```

will not. Note that

```
EQUIVALENCE (A,B)
INTEGER B
```

is acceptable since the length of B is not changed by the type statement.

11. Not all floating-point constants are converted to the correct internal hexadecimal format; in addition there exists differences in the handling of floating-point constants at compile and execution time.

8. CHARACTER VARIABLES

At a meeting held during the SHARE XXVIII Conference in San Francisco in February, 1967, the SHARE FORTRAN Project proposed that IBM adopt a new type of variable as an extension to the FORTRAN language supported by IBM's compilers. The following material is copied from Appendix B of the minutes of that meeting of the FORTRAN Project since it defines, for the most part, WATFIV's implementation of CHARACTER variables. Additional material is given below in subsection 8.4 on page 68.

Character data is recognized as a legitimate data form which may be manipulated to a limited extent. The general effect to the language is:

1. CHARACTER is a variable type.

2. Core-to-core READ and WRITE statements allow in-core formatting.

3. Implicit record-size for CHARACTER arrays for FORMAT statement control is defined in the Type statement (not in the READ and WRITE statements).

4. A WRITE statement may be used to define a variable.

8.1 DECLARATION OF CHARACTER VARIABLES

8.1.1 VARIABLE TYPE: CHARACTER

A variable of type CHARACTER represents a character string (literal data). The standard and optional length specifications which determine the number of characters that are reserved for each character variable are:

Variable Type	Standard	Optional[1]
Character	1	$1 \leq n \leq m$

where m should be the size of the maximum print line or greater.

A programmer may declare a variable to be of type CHARACTER by use of the:

1. IMPLICIT specification statement.
2. Form of the explicit specification statement: CHARACTER.

8.1.2 IMPLICIT STATEMENT

The type CHARACTER is permitted in the IMPLICIT statement with a specified length. If length is omitted, the standard length of 1 is assumed.

Example:

```
IMPLICIT CHARACTER*80 (A-D), CHARACTER ($,Z)
```

Explanation:

All variables beginning with the characters A through D are declared as CHARACTER type, each variable or array element 80 characters in size. All variables beginning with the character $ and Z are declared as CHARACTER. Since no length specifications was explicitly given, 1 character (the standard length for CHARACTER) is allocated for each

---

(1) WATFIV uses 255 for m.

variable.

8.1.3 CHARACTER TYPE STATEMENT

The general form of the CHARACTER type statement is as follows:

CHARACTER*s a*s1(k1)/x1/,b*s2(k2)/x2/,...,z*sn(kn)/xn/

Where: *s,*s1,*s2,...,*sn are optional. Each s represents one of the permissible length specifications.

a,b,...,z represent variable or array names.

(k1),(k2),...,(kn) are optional. Each k is composed of 1 through 7 unsigned integer constants separated by commas, representing the maximum value of each subscript in the array. Each k may be an unsigned integer variable only when the CHARACTER statement in which it appears is in a subprogram.

/x1/,/x2/,...,/xn/ are optional and represent initial data values.

The information necessary to allocate storage for arrays (dimension information) may be included within the statement. However, if this information does not appear in a CHARACTER statement, it must appear in a DIMENSION or COMMON statement (see, "DIMENSION Statement" or "COMMON Statement").

Initial data values may be assigned to variables or arrays by use of /xn/ where xn is a constant or list of constants separated by commas.

This set of constants may be in the form "r* constant", where r is an unsigned integer, called the repeat constant. The initial data values may only be literal constants and must be the same length as, or shorter than, the corresponding variable or array element. Initial data values will be truncated from the right and diagnosed if too long, and they will be padded with blanks on the right if too short (see "Example 2" below).

An initially defined variable or element of an array may not be in blank common. In a labelled common block they may be initially defined only in a BLOCK DATA subprogram.

The CHARACTER statement overrides the IMPLICIT statement. If the length specification is omitted (i.e., *s), the standard length of 1 is assumed. If an array is used in a subprogram and is not in COMMON, the size of this array may be specified implicitly by an integer variable of length 4 which can appear explicitly in the SUBROUTINE statement or implicitly in COMMON (adjustable dimensions).

Example 1:

```
      CHARACTER*80 CARDS (10), LINES*132(56,2),TCARD
```

Explanation:

This statement declares that the variable TCARD and the arrays named CARDS and LINES are of type character. In addition, it declares the size of the array CARDS to be 10 and array LINES to be 112 (2 groups of 56 each). Each element of the array LINES is assigned 132 characters for a total of 14,784 (112 times 132) for the array.

Each element of the array CARDS and the variable TCARD is assigned 80 characters (the length associated with the type). The array CARDS is assigned a total of 800 characters.

Example 2:

```
      CHARACTER X*3(4)/'ABC','DEFG','HI','JKL'/
```

Explanation:

This statement declares that the array of four elements of three characters each named X has initial values:

X(1)	ABC
X(2)	DEF
X(3)	HI
X(4)	JKL

The statement is incorrectly written, and the value specified for X(2) has been altered by truncating.

8.2 USING CHARACTER VARIABLES IN FORTRAN STATEMENTS

8.2.1 DIMENSION STATEMENT

Character type array names may appear in DIMENSION statements.

8.2.2 COMMON STATEMENT

Character type variables or array names may appear in COMMON statements.

8.2.3 NAMELIST STATEMENT

Character type variables or array names may appear in NAMELIST statements.

8.2.4 DATA STATEMENT

Character type variables, array element names or array names may appear in DATA statements. The data values may only be literal constants and must be the same length as, or shorter than, the corresponding variable or array element. Initial data values will be truncated from the right and diagnosed if too long, and they will be padded with blanks on the right if too short (see "Example 2" under "CHARACTER Statement" above).

8.2.5 EQUIVALENCE STATEMENT

Character type variables, arrays or array elements may appear in EQUIVALENCE statements. Character type data may be equivalenced to other than Character type data but implies storage sharing only.

Example:

```
          .
          .
          .
     CHARACTER A*5,B*2,C*1
     CHARACTER D*1(5)
     EQUIVALENCE (D(1),A),(D(2),B),(D(5),C)
          .
          .
          .
```

Explanation:

These statements cause the following alignment of characters:

```
     A-----

     B --

     C    -
```

The use of the array D enables equivalencing to characters in the middle of the variable A.

8.2.6 CALL STATEMENT

Character variable names, array element names, array names, and literal constants may appear as parameters in a CALL statement.

8.2.7 FUNCTION REFERENECE

Character variable names, array element names, array names, and literal constants may appear as parameters in a function reference.

Example:

```
      CHARACTER CARD*80
      .
      .
      .
      READ (5, 1) CARD
    1 FORMAT (A80)
      IF (COMPAR (CARD, 'END',4)) 2,3,2
    3 STOP
    2 CONTINUE
      .
      .
      .
```

Explanation:

An 80 character image is read into the element CARD. A function, COMPAR, is used to compare the first four characters of CARD with END and used to return a positive, negative, or zero numeric value which is used conditionally to terminate the program.

8.2.8 STATEMENT FUNCTION STATEMENTS

Non-subscripted character variable names may appear as parameters in a statement function statement.

8.2.9 SUBROUTINE STATEMENT

Character variable names and array names may appear as parameters in a SUBROUTINE statement.

8.2.10 FUNCTION STATEMENT

Character variable names and array names may appear as parameters in a FUNCTION statement.

8.2.11 REPLACEMENT STATEMENT: A=B

A replacement statement in which all variables, constants or array elements are of type CHARACTER is permissible. In such a statement the item on the left-hand side may only be a character variable name or a character array element; the item on the right hand-side may be a character variable

name, a character array element, or a character (literal) constant.

The element on the right-hand side must be the same length as, or shorter in length than, the element on the left-hand side. The value of the right-hand element will be truncated from the right during replacement and diagnosed if too long, and will be padded with blanks on the right if too short.

NOTES:

1. The term "literal constant" should be replaced in the language definition by "character constant".
2. Multiple assignment statements for CHARACTER variables are not supported by WATFIV.

8.3 CORE-TO-CORE I/O STATEMENTS

An additional type of I/O statement provides for core-to-core transmission of data under FORMAT control. There are two core-to-core I/O statements: READ and WRITE. In a core-to-core operation no actual input/output takes place; data conversion and transmission take place between an internal buffer and the elements specified by a list.

8.3.1 WRITE STATEMENT

The general form of the core-to-core WRITE statement is as follows:

WRITE (a, b) list

Where: a is a character array, array element or variable name which specifies the starting location of the internal buffer to which data is to be transmitted.

b is a statement number of a FORMAT statement or an array name or array element indicating the beginning location of a FORMAT statement which describes the data to be transmitted.

list is a series of variable or array names, separated by commas, which may be indexed and incremented. They specify the number of items to be written and the locations in storage from which the data is taken.

This form of the WRITE statement causes the data items specified by the list to be converted to character strings, according to the FORMAT specified by b, and placed in storage beginning at first character element specified by a.

Characters are placed into the buffer, starting with the first character position of the first element specified by a, in consecutive character positions. When a new record is begun, it starts at the first character position of the next element.

The number of characters for a record caused to be generated by the FORMAT statement and list should not be greater than the size of the elements specified by a. If fewer characters are generated than are necessary to fill the element, it is filled out with trailing blanks.

Example 1:

```
      CHARACTER M*12
         .
         .
         .
      I=15
      J= 7
         .
         .
         .
      WRITE (M,2) I,J
    2 FORMAT (2H(F,I2,1H.,I1,1H))
         .
         .
         .
```

Explanation:

These statements might be used to create, for later use, a FORMAT stored in variable M. The FORMAT so created would appear as:

```
      (F15.7)bbbbb
```

where b represents the character blank.

Example 2:

```
      CHARACTER M*12, N*132
         .
         .
      K=FUNC (A, B, C, D)
         .
         .
         .
    2 WRITE (M,4) K
    4 FORMAT(1H(,I3,6HX,1H*))
    6 WRITE (N,M)
         .
         .
         .
         .
```

Explanation:

These statements prepare a character string 132 long for use in printer plotting. The print position K is determined by the function FUNC. Statement 2 creates a FORMAT stored in variable M, which, for a value of K of 96, would appear as:

```
      (b96X,1H*)bb
```

Statement 6 then used the above FORMAT (in variable M) to prepare a character string 132 long in variable N which consists of all blanks except for an asterisk in the ninety-seventh character.

8.3.2 READ STATEMENT

The general form of the core-to-core READ statement is as follows:

READ (a, b) list

Where: a is a character array, array element, or variable name which specifies the starting location of the internal buffer from which data is to be transmitted.

b is either the statement number of a FORMAT statement or a character array element indicating the beginning location of a FORMAT statement which describes the data to be transmitted.

list is a series of variable or array names, separated by commas, which may be indexed and incremented. They specify the number of items to be read and the locations in storage into which the data is placed.

This form of the READ statement causes the character string beginning at the first character element specified by a to be converted to data items, according to the FORMAT specified by b, and stored in the elements specified by the list.

Characters are obtained from the buffer starting with the first character position of the first element specified by a, from consecutive character positions. When a new record is begun, it starts at the first character position of the next element.

The FORMAT statement and list should not require more characters from an element than the length of that element. A new record is begun when specifically requested by the FORMAT.

Example:

```
      CHARACTER*80 R(10)
           .
           .
           .
      DO 20 I=1,10
3     READ (R(I),5) J
5     FORMAT (I1)
      GO TO (11, 12, 13, 14, 15, 16, 17, 18, 19), J
11    READ (R(I),21) (A(K), K=1,10)
21    FORMAT (1X, 10F8.3)
      GO TO 31
12    READ (R(I),22) K1, K2, K3, K4
22    FORMAT (1X,4I5)
      GO TO 32
13    READ (R(I), 23) X, Y, Z
23    FORMAT (1X,3E20.9)
      etc.
           .
           .
           .
```

Explanation:

The statements illustrate a method of processing randomly ordered input cards of varying format and data content. The card type is identified by a digit from one to nine in the first column. Statement 3 converts the digit from character form to integer form. The GO TO then transfers to the READ/FORMAT combination prepared to process the specified format.

8.3.3 INPUT/OUTPUT LIST

CHARACTER variable names, array element names, and array names may appear in input/output lists.

8.4 ADDITIONAL CHARACTER FEATURES SUPPORT

The features of CHARACTER variables given in this subsection were not described in the SHARE proposal of the last subsection, and hence, are considered as extensions to that proposal.

It should also be mentioned that WATFIV supplies no particular alignment for CHARACTER variables, unless, of

course, they are forced to some half-word, full-word, or double-word boundary by COMMON and/or EQUIVALENCE statements.

8.4.1 USE AS SUBSCRIPTS

Subscripts may be of LOGICAL or CHARACTER value. The first character (left-most byte) in the quantity is used as the low-order byte of a four-byte integer to form the actual subscript. For example, A('123') is the same as A(241) since the internal representation of the character '1', taken as an integer value, is equivalent to 241.

Example of use: The following loop will translate each character of a card according to the translate table TRANSL.

```
      CHARACTER*1  TRANSL(255),CARD(80)
           .
           .
           .
      DO 1 I=1,80
    1 CARD(I)=TRANSL(CARD(I))
           .
           .
           .
```

8.4.2 USE WITH RELATIONAL OPERATORS

CHARACTER variables may be used as operands of relational operators provided both operands are of type CHARACTER. All values are treated as if they were in IBM 360/370 EBCDIC representation.

e.g.,

```
      CHARACTER A*1, B*5, C*5(10)
           .
           .
           .
      IF (A .EQ. C(I)) GO TO 10
           .
           .
           .
      IF (B .LE. 'AAAAA') GO TO 30
           .
           .
           .
```

For the purposes of the comparison when operands of unequal length are involved, the shorter operand is considered as if padded on the right with blanks to the length of the longer operand. A warning message is issued at compile time when operands of differing lengths are used.

Note that this feature is highly dependent on the IBM 360/370 machine representation of EBCDIC characters.

9. STRUCTURED PROGRAMMING STATEMENTS

A number of new control statements have been added to WATFIV to facilitate the control of program flow without the use of GOTO statements. These statements have been introduced to enable the FORTRAN programmer to design and write programs in a structured manner. These statements are clearly extensions to FORTRAN-IV and are incompatible with other FORTRAN implementations.

The statements introduced are the following:

1. IF-THEN-ELSE
2. WHILE-DO
3. DO CASE
4. EXECUTE and REMOTE BLOCK
5. WHILE-EXECUTE
6. AT END DO

It is hoped that the use of these new control statements will encourage better programming and design practices among beginners, and will aid the more experienced programmer in writing bug-free programs.

Since these new language constructs are not available with other compilers, a translator has been written to convert structured control statements to standard FORTRAN. The translator was written using structured constructs and does not contain a single GOTO. Programs which do not use any other WATFIV extensions to FORTRAN, and compile correctly under WATFIV, may be translated by this program to a form acceptable to IBM FORTRAN. Using the combination of WATFIV and the translator, programmers can write well-structured FORTRAN programs, debug them using WATFIV, then produce a production version with the translator to be optimized with IBM FORTRAN.

The format of these new statements and their blocks is illustrated below. Following this the use and meaning of each statement is described and illustrated with examples.

In each of these illustrations, the blocks are denoted by 'statement(s)' and are delimited by the control statements and special END statements.

9.1 IF - THEN - ELSE

The ELSE portion of this construct is optional, thus there are two possible formats.

a) IF (logical-expression) THEN DO

```
        statement(s)
    END IF
```

b)
```
    IF (logical-expression) THEN DO
        statement(s)
    ELSE DO
        statement(s)
    END IF
```

This construct is an extension of the FORTRAN logical IF statement. If the value of the parenthesized logical expression is true in case a, the block of statements following the THEN DO is executed, after which control passes to the statement following the END IF; otherwise, control will pass directly to the statement following the END IF. When ELSE DO is used and the logical expression is false, the block following the ELSE DO is executed and then control passes to the statement following the END IF.

Examples follow which illustrate the use of the two formats:

```
IF  (I.EQ.0) THEN DO
    PRINT,'I IS ZERO'
    I=1
END IF
```

If I is zero when the IF statement is executed, the string 'I IS ZERO' will be printed, I will be assigned the value 1, and the statement following the END IF will be executed. If I is not zero when the IF statement is executed, control will pass to the statement following the END IF.

```
IF  (A .GT. B) THEN DO
    PRINT, 'A GREATER THAN B'
    A = A - B
ELSE DO
    PRINT, 'A NOT GREATER THAN B'
END IF
```

If the value of variable A is greater than the value of B when this IF statement is executed, the string 'A GREATER THAN B' will be printed and A will be assigned the value of the expression A-B. Control will then pass to the statement following the END IF.

If the value of A is not greater than the value of B when the IF statement is executed, the string 'A NOT GREATER THAN B' will be printed and control will pass to the statement following the END IF.

9.2 WHILE - DO

```
WHILE (logical-expression) DO
    statement(s)
END WHILE
```

This control statement causes its block of code to be executed repeatedly while the parenthesized logical expression is true. The logical expression is evaluated before entry to the block. If the value is false, control passes to the statement following the END WHILE statement. If the logical expression is true, the statements of the block are executed. When the END WHILE statement is reached, the WHILE logical expression is re-evaluated and the above program control decisions are repeated.

Note that the word DO must be part of the WHILE statement. An example follows:

```
WHILE (J.GT.0) DO
    A(J) = B(I+J)
    J = J-1
END WHILE
```

If J is zero or negative when the WHILE statement is executed, the WHILE block of code will be by-passed and the statement following the END WHILE will be executed.

If J is greater than zero when the WHILE statement is executed, the WHILE block will b executed repeatedly until J becomes equal to zero. The effect of this loop will be to assign values to elements of array A from array B, starting with the element of A corresponding to the initial value of J and working backwards down the array to element 1.

9.3 DO CASE

```
DO CASE index
CASE
     statement(s)
CASE
     statement(s)
 .
 .
 .
CASE
     statement(s)
IF NONE DO
     statement(s)
END CASE
```

In the above definition 'index' is a simple integer variable.

The DO CASE construct is similar in concept to the FORTRAN computed GOTO. It allows one of a number of blocks of code (case blocks) to be selected for execution by means of an integer CASE index variable.

The first block may be started with a CASE statement; however, this first CASE statement is optional. The IF NONE DO block is also optional. The last block is ended by the END CASE statement. Intermediate case blocks are separated by CASE statements. The number of cases is optional, from one to many; however, it is recommended that the DO CASE construct not be used for fewer than 3 cases. The conditional execution of one or two blocks of code is handled more efficiently by the IF-THEN-ELSE construct.

When the DO CASE statement is executed with index i, the i'th case block is executed and control passes to the statement following the END CASE. If the IF NONE DO block is omitted and the index is out of range when the DO CASE is executed (that is, index variable is zero, negative, or exceeds the number of case blocks), control is passed to the statement following the END CASE and none of the case blocks is executed.

```
DO  CASE  I
    Y = Y+X
    X = X*3.2
CASE
    Z = Y**2+X
    PRINT,X,Y,Z
CASE
    Y = Y*13.+X
    X = X - 0.213
CASE
    Z = X**2+Y**2 - 3.0
    Y = Y+1.5
    X = X*32.0
    PRINT,'CASE 4',X,Y,Z
END CASE
```

This example will execute in the manner described below for each of the possible values of variable I.

i) I is zero or negative:
control will pass to the statement after the END CASE

ii) I = 1:
the value of X will be added to Y
X will be multiplied by 3.2
control will pass to the statement after the END CASE

iii) I = 2:
Z will be assigned the value of the expression Y**2 + X
the values of X,Y and Z will be printed
control will pass to the statement after the END CASE

iv) I = 3:
Y will be assigned the value of the expression Y* 13. + X
0.213 will be subtracted from X
control will pass to the statement after the END CASE

v) I = 4:
Z,Y and X will be assigned new values
the string 'CASE 4', followed by the values of X,Y and Z will be printed
control will pass to the statement after the END CASE

vi) I = 5,6,. . .:
control will pass to the statement after the END CASE

IF NONE DO allows a block of code to be specified for execution when the CASE index is out of range. It must follow all CASE blocks and thus is ended by the END CASE statement. The IF NONE DO statement terminates the previous and last CASE block. Note that only one IF NONE DO block may be specified in a DO CASE construct.

If an IF NONE DO block were included in the above example, it would be executed in cases of the description. After an IF NONE DO block is executed, control then passes to the statement after the END CASE.

Empty or null case blocks are permitted (that is, two delimiter statements with no statements between). The net result of executing a null case block is to effectively bypass the DO CASE construct. These null case blocks, however, affect the numbering of other case blocks for indexing.

9.4 EXECUTE AND REMOTE BLOCK

```
EXECUTE   name
   .
   .
   .
REMOTE BLOCK name
     statement(s)
END BLOCK
```

where name is a valid FORTRAN symbolic name.

The EXECUTE statement allows a named block of code to be executed. The named block of code may be defined anywhere in the same program segment and is delimited by the REMOTE BLOCK and END BLOCK statements. Executing a REMOTE BLOCK is similar in concept to calling a subroutine, with the advantage that shared variables do not need to be placed in a COMMON block or passed in an argument list. In addition there is less overhead involved in executing a REMOTE BLOCK than in calling a subroutine (in both amount of object code and execution time). When execution of the REMOTE BLOCK is complete, control returns to the statement following the EXECUTE which invoked it.

This feature is helpful in avoiding duplication of code for a function (such as I/O) required in a number of places throughout a program. It can also be an aid to writing a well-structured program.

Each REMOTE block must have a different name; however, they need not be different than subprogram or variable names. Note that a REMOTE block is local to the program segment in which it is defined and may not be referenced (executed) from another program segment. Due to symbol table restrictions, a maximum of 255 REMOTE BLOCKs may be defined in a program segment.

REMOTE BLOCKs may be defined anywhere in the program segment; however, they must be preceded by an instruction causing transfer of control. Also, executable statements following a REMOTE BLOCK will never be executed unless they are numbered and referenced by a standard FORTRAN control statement. The END BLOCK is implicitly a transfer statement, since it returns program control from the block; thus REMOTE BLOCKs may follow each other and may precede the END statement for the program segment.

Note that the nested definition of REMOTE BLOCKs is not permitted.

```
            .
            .
            .
      EXECUTE A
      PRINT, 'FIRST'
            .
            .
            .
      EXECUTE A
      PRINT, 'SECOND'
            .
            .
            .
      REMOTE BLOCK A
            I=I+1
            PRINT, 'I=',I
      END BLOCK
            .
            .
            .
```

Both EXECUTE statements will cause REMOTE BLOCK A to be executed. That is, variable I will be incremented and its value will be printed. When the block has been executed by the first EXECUTE, control returns to the PRINT statement following it and the word FIRST is printed. Similarly, when the block is executed by the second EXECUTE, control returns to the PRINT statement following it and the word SECOND is printed.

REMOTE BLOCKs may be executed from other REMOTE BLOCKS. For

example, REMOTE BLOCK A might contain the statement EXECUTE B, where B is a REMOTE BLOCK defined elsewhere in the program segment. The execution of REMOTE BLOCKs from other REMOTE BLOCKs may take place to any level; however, the recursive execution of REMOTE BLOCKs is not permitted, either directly or through a chain of EXECUTEs. Attempts to execute REMOTE BLOCKS recursively are detected as errors at execution time.

9.5 WHILE - EXECUTE

```
WHILE (logical-expression) EXECUTE name
```

This control statement is a combination of the WHILE-DO construct and the EXECUTE statement.

```
WHILE (I.GT.0) EXECUTE A
```

When this statement is executed, if the logical expression is not true, control passes to the next statement. If the expression is true, REMOTE BLOCK A (assumed to be defined elsewhere in the program segment) is executed, and the logical expression is re-evaluated. This is repeated until the logical expression, when evaluated, is false; control then passes to the next statement.

9.6 AT END DO

```
(READ statement)
AT END DO
    statement(s)
END AT END
```

The AT END DO control statement is an extension of the 'END=' option of the FORTRAN READ statement for sequential files. It allows a block of code following the READ statement to be executed when an end-of-file condition is encountered during the READ and to be by-passed otherwise. The AT END DO statement must immediately follow a READ statement. It is not valid to use this control statement with direct-access or core-to-core READs. Clearly, it is not valid to use this statement when 'END=' is specified in the READ statement.

```
READ,I,X
AT  END  DO
    PRINT,'END-OF-FILE ENCOUNTERED'
    EOFSW=.TRUE.
END  AT  END
```

If the READ statement is executed without encountering end-of-file, control passes to the statement following the END AT END. If an end-of-file condition occurs during the read, the string, 'END-OF-FILE ENCOUNTERED', is printed, logical variable EOFSW is assigned the value .TRUE., and control passes to the statement following the END AT END.

9.7 PROGRAMMING CONSIDERATIONS

In addition to the definitions and examples of these six constructs, the following points should be noted:

1. Any of the new control statements with their blocks may be used within the block of any other statement. For example, a WHILE-DO block may contain another WHILE-DO or an IF-THEN-ELSE. Blocks may be nested in this manner to any level within storage limitations. An important exception to this rule is the REMOTE BLOCK. A REMOTE BLOCK may contain other types of blocks (nested to any level); however, another REMOTE BLOCK may not be defined within it.

2. When nesting blocks, the inner blocks must always be completed with an appropriate 'END' state before the outer blocks are terminated. Similarly, when nesting blocks with DO-LOOPS, a DO-LOOP started within a block must be completed before the block is completed. A block started within a DO-LOOP must be terminated before the DO-LOOP is completed. Indenting the statements of each new block, as shown in the examples, is helpful in avoiding invalid nesting and helps to make the structure of the program visually obvious.

3. The normal flow of control of the new programming constructs described earlier may be altered by standard FORTRAN control statements. For example, the program may exit from a block using a GOTO, STOP, RETURN or arithmetic IF statement. Similarly, a block may be entered in the middle with some of the above statements. When a block is entered in this manner, the remainder of the block (from the point of entry on) will be executed and control will pass to the statement following the special END statement which terminates the entire control structure. For example, if a CASE block was entered with a GOTO, the remainder of the block would be executed and control would pass to the statement following the END CASE. However, these new constructs allow the programmer to eliminate most of the transfer statements that

would ordinarily appear in a program.

The WHILE-DO block is an exception to the above rule. When entered this way, the remainder of the WHILE-DO block will be executed and control will pass to the WHILE statement, where its logical expression is evaluated. If the value of the expression is false, control passes to the next statement after the END WHILE. If the expression is true, the WHILE-DO block is executed normally and is repeated until the value of the WHILE logical expression becomes false.

Another exception to this rule is the REMOTE BLOCK. Transfer of control into or out of a REMOTE BLOCK by means of standard FORTRAN control statements is not permitted. Attempts to do this are flagged as errors during compilation of the program.

4. Special END statements, CASE, REMOTE BLOCK, IF NONE DO, ELSE DO, and AT END DO statements are branched to directly by means of a GO TO statement or other FORTRAN control statements.

5. None of the new statements can form the object of a LOGICAL IF, or be the last statement of a DO-LOOP, with the one exception of the EXECUTE statement.

6. Comments may follow the CASE and END BLOCK statements. This enables the user to number case blocks or denote the block's function. Any valid characters following the words CASE or END BLOCK are ignored, with the exception of the assignment operator (=) which may result in the statement being decoded as an assignment statement.

7. The format and keywords of these new control statements are still under discussion and may be subject to change. Comments and suggestions will be welcome.

9.8 CONTROL STATEMENT TRANSLATOR

TRANSL is a subroutine that translates a WATFIV program containing structured statements to standard FORTRAN. Programs which do not use any other WATFIV extensions to FORTRAN, and compile correctly under WATFIV, may be translated by this program to a form acceptable to IBM FORTRAN. A copy of this subprogram is in WATFIV's standard source subprogram library --- WATFIV.WATLIB.

How To Use:

```
          CALL  TRANSL (DECK,PUNCH)

where     DECK  =  unit number for input data

          PUNCH =  unit number for output data
```

To read cards from the reader and punch a new deck, the following job may be run:

```
          $JOB  id,parameters
                CALL TRANSL(5,7)
                STOP
                END
          $ENTRY
                input deck
```

A number of extension messages will be printed since TRANSL uses structured control statements. Printing of these messages may be suppressed by specifying the NOEXT option on the $JOB card or on the C$OPTIONS card.

Restrictions:

1. Statement numbers 90000-99999 are reserved for the translator. Variable names beginning with '$' are reserved for the translator.

2. Structured control statement keywords (e.g., WHILE, ELSEDO) should not be used as variable names on the left-hand side of an assignment statement.

3. Structured control statements must be complete on one card, with the exception of IF-THEN and WHILE-DO header statements. For these exceptions 'IF(' and 'WHILE(' must be complete on the first card of the statement.

4. A maximum of 19 continuation cards will be allowed for READ statements.

5. Comment cards between continuation cards of a statement are not allowed.

6. Error checking is generally not performed by the translator, but certain errors are detected by the translation algorithm. Translation is terminated when such errors occur.

7. Remote block names must be unique in their first 5 characters and remote block definitions must follow all their references.

8. 'EXECUTE' as the object of a DO LOOP will not translate correctly. This problem may be circumvented by using a 'CONTINUE' statement following the EXECUTE, as DO-object.

9. The generated statement 'IMPLICIT INTEGER($)' may have to be re-positioned in the output deck if subprograms are translated separately or multiple mainline programs are translated together. This is not necessary for decks with mainline first, followed by subprograms.

Systems Notes:

TRANSL generally does not perform error checking. Programs being translated should compile correctly under WATFIV and conform to the listed restrictions. In some circumstances, errors are detected by the translation algorithm and a message is printed. If an error is not detected, WATFIV run-time error messages may result, or the translator may just produce incorrect code.

TRANSL has two sets of arrays which are defined with one of the dimensions set at 50 in each:

1. LABEL1, LABEL2, TYPE, CASIN1, CASIN2

2. BNAME, STRTNO, RETRNS

Set 2 is used for remote blocks and set 1 is used for all other blocks. If subscripting for any of these arrays goes out of bounds, the dimension set at 50 may be increased. If one array is increased in size, the rest of the arrays in its set should be increased also.

This should not be necessary except for very large programs segments. The translator will translate itself within the present array bounds.

The following notes may help in diagnosing non-obvious problems:

1. 'GOTO 0' is generated as part of the 'EXECUTE' code when the remote block referenced has been previously defined.

2. 'GOTO 0' may also be generated for an 'EXECUTE' when two remote blocks have names that are not unique in their first five characters.

10. INTERRUPTS

This section provides information on the treatment of interrupts that may occur during the execution of a FORTRAN program.

Normally, WATFIV terminates execution of the program at the first occurrence of an exponent overflow, exponent underflow, fixed divide, or floating divide interrupt. However, a library subroutine, TRAPS, is provided to allow the programmer to accept more interrupts of the types just mentioned. Thus, with appropriate uses of subroutine DVCHK and OVERFL, a programmer may handle, to some extent, the treatment of interrupts.

A call to TRAPS may have up to five integer-valued arguments, and these correspond to the number of fixed overflows, exponent overflows, exponent underflows, fixed divide, and floating divide interrupts the programmer wishes to allow. The arguments of TRAPS set up internal counters used by the compiler's interrupt routine. The latter routine decrements the appropriate counter by 1 when an interrupt occurs; when any counter reaches zero, the program is terminated.

TRAPS may be called (and subsequently recalled) at any point in the main program or a subprogram to set (or reset) the interrupt counters. Arguments of TRAPS are screened so that the absolute value of any negative argument is used as a positive count, and a zero value is taken to mean that the current value of the corresponding interrupt counter should be left unchanged. If the value of an argument is undefined, the program is terminated (unless NOCHECK has been specified).

EXAMPLES:

1. CALL TRAPS (0,5,7,-3,1)

   sets the interrupt counters so that the program will be terminated on the occurence of the first of the:

   - 5th exponent overflow, or
   - 7th exponent underflow, or
   - 3rd fixed divide, or
   - 1st floating divide exception following the execution of this call to TRAPS.

The statement CALL TRAPS (0,5,7,3) has the same effect.

2. LUNFLO = 100
   LOVFLO = LUNFLO
   CALL TRAPS (0, LUNFLO, LOVFLO)

   sets the counts to terminate the program on the occurrence of the first of the:

   - 100th exponent overflow, or
   - 100th exponent underflow, or
   - 1st fixed divide, or
   - 1st floating divide exception following the execution of this call.

3. CALL TRAPS (14)

   sets the fixed overflow counter to 14.

   Termination would occur at the 1st exponent overflow, underflow, or divide exception, or the 14th fixed overflow if the installation has activated this interrupt. Note that the distributed version of WATFIV operates with this interrupt masked off, and furthermore, that this is the normal mode of operation of IBM FORTRAN.

OVERFL, DVCHK

These routines function as follows:

```
CALL DVCHK (j)
```

where j is an integer variable that is set to 1 if the (pseudo-) divide-check indicator was on, or to 2 if off. After testing, the indicator is turned off.

The indicator is set on when a fixed or floating divide exception occurs.

```
CALL OVERFL (j)
```

where j is an integer variable that is set to reflect the most recent setting of a pseudo-indicator. The variable j is set to 1 if an exponent overflow was last to occur, to 2 if no exponent overflow or underflow condition exists, or to 3 if an exponent underflow was last to occur. After

testing, the indicator is set for no condition, i.e., to 2.

NOTES:

1. The compiler interrupt routine loads the affected machine floating-point register with zero or the properly signed, largest floating-point number for exponent underflow or overflow, respectively.

2. The five interrupt counters are initialized by the compiler to 1 at the start of each program. The divide-check and overflow indicator are not initialized; it is the programmer's responsibility to do this, e.g., by dummy calls.

3. The terminating message is the only indication given by the compiler that interrupts have occurred. It is the programmer's responsibility to monitor these using OVERFL and DVCHK.

4. WATFIV operates with the fixed overflow and significance interrupts masked off entirely.

5. WATFIV automatically corrects for boundary alignment errors at execution time, but this is done not without some overhead. Thus, programmers are advised to ensure that operands are aligned properly, where possible, by steps taken at the source level.

11. INPUT OUTPUT CONSIDERATIONS

For execution-time I/O on units other than 5 and 6, WATFIV uses routines taken directly from IBM's FORTRAN library. Consequently, the rules and considerations for performing execution-time I/O are generally the same as are described for load module execution in the IBM FORTRAN-IV Programmer's Guide (IBM form GC28-6817), to which the reader is directed. Differences only are given in the following notes.

11.1 GENERAL NOTES

1. Since the WATFIV compiler is essentially a one-step job, any DD cards for execution-time data sets must be included in the JCL used to invoke the compiler. An example using the catalogued procedure WATFIV, follows:

```
//jobname   JOB   accounting
//  EXEC    WATFIV
//GO.FT01F001  DD DSN=etc.
//GO.FT02F001  DD DSN=etc.
//GO.SYSIN     DD *

       WATFIV JOBS
/*
```

2. The compiler reads the compile-time input (source programs) and execution-time card-image data for unit 5 from the data set defined by a DD card with DD name FT05F001. (The WATFIV procedure contains the DD card //FT05F001 DD DDNAME=SYSIN to redefine the compiler input to SYSIN.) Similarly, compile- and execution-time output is on one data set defined by the FT06F001 DD card.

3. The WATFIV procedure (see section 2.2.1 on page 5) defines temporary data sets for DD names FT01F001, FT02F001, FT03F001, and FT04F001.

4. The upper limit, generated into the compiler, for data set reference numbers is 16.

5. Files referenced by DD names FT15F001 and FT16F001 are given read-only status. An execution-time error message will be issued when a program attempts to write on data sets in this particular range of unit numbers.

6. Buffer space and other dynamically obtained storage

for DCBs, access method routines, etc., is not included in the core usage figures given in the accounting output for a job.

7. WATFIV error messages relating to execution-time I/O give, where appropriate, the corresponding IBM error code (for which, see GC28-6817).

11.2 COMPILER DATA SET ASSUMPTIONS

WATFIV uses the Queued Sequential Access Method (QSAM) to process the data sets defined by the FT05F001 and FT06F001 DD cards (i.e., compile-time input and output, and execution-time input on unit 5 and output on unit 6).

The following DCB assumptions are made:

	RECFM	LRECL	BLKSIZE	BUFNO
FT05F001	FB	80	80	2
FT06F001	FBA	133	133	2

NOTE:

1. The BLKSIZE and BUFNO values may be supplied from the DD card or data set label; the values given in the table above are defaults.

2. The BLKSIZE, if not that shown above, must be a multiple of the LRECL value.

11.3 CONCATENTATING COMPILER INPUT

WATFIV's input stream may consist of a concatenation of distinct data sets. The following examples illustrate potential uses of this feature:

1) Source program and execution card-image data to be read by 'card reader' unit 5 can be in disjoint data

sets.

```
//osjob   JOB   accounting
//   EXEC   WATFIV
//GO.FT05F001  DD  DDNAME=PROG
//             DD  DDNAME=DATA
//GO.PROG      DD  *
$JOB   id,parms


        source program


$ENTRY
//GO.DATA  DD  DSN=WATFIV.SUBSUB(DATA),DISP=SHR
```

2) Segments of the source program to be compiled can come from different sources.

```
//osjob   JOB   accounting
//   EXEC   WATFIV
//GO.FT05F001  DD  DDNAME=JOB,DCB=BLKSIZE=800
//             DD  DSN=WATFIV.MAINPROG,DISP=SHR
//             DD  DSN=WATFIV.SUBSUB(SUB1),DISP=SHR
//             DD  DSN=WATFIV.SUB2,DISP=SHR
//             DD  DDNAME=ENTRY
//GO.JOB       DD  *
$JOB           id,parms
//GO.ENTRY     DD  *
$ENTRY


            any data


/*
```

(This example assumes the operating system allows multiple DD * data sets in the input stream.)

NOTES:

1. All data sets appearing in the concatenation are subject to the assumptions of section 11.2 above.

2. When compile-time input (i.e., source program components) is being processed, the total memory required for input buffers (BLKSIZE*BUFNO) must not increase as the compiler proceeds from one data set to the next in the concatenation. (Input buffer space can increase when proceeding from compile-time to

execution-time input.)

Example (b) above, shows a simple way to meet the requirements of Note 2. This is to put the largest BLKSIZE on the first DD card of the concatenation[1]. (For the purposes of the example, the largest BLKSIZE of the 5 data sets in the concatenation is assumed to be 800.)

---

(1) Strictly speaking, this restriction applies only if the compiler has been generated with the 'dynamic memory allocation' feature; this is the most likely way of generating the compiler.

12. SUBPROGRAM FACILITIES

This section provides some information on the subprogram facilities available with the WATFIV compiler. Rules for passing values between subprograms are also discussed.

12.1 SOURCES OF SUBPROGRAMS

Any subprogram referenced in a FORTRAN program run under WATFIV must come from one of three possible sources:

- card decks in the compiler's input stream (SYSIN), i.e., the usual program input.
- core-resident library routines internal to the compiler itself. For example, the routines EXP, DEXP, ALOG, ALOG10, DLOG, DLOG10, EXIT, SQRT, etc., may be in core as an installation choice.
- routines from libraries stored on direct-access devices and defined by appropriate DD cards in the control cards used to invoke the compiler.

The search for subprograms is made in the order just mentioned, i.e., a user may supply a subprogram EXIT, for example, but an in-core version (assuming there is one) will be used in preference to one which may be on a direct-access library.

Normally, a user need not be concerned with which routines are in core; problems may arise only if an attempt is made to supply, from a direct-access library, routines with names the same as any FORTRAN-supplied subprograms which happen to be core-resident in the version of WATFIV being used..

12.2 FORTRAN SUPPLIED ROUTINES

The user of WATFIV has available all function and subroutine subprograms (except DUMP and PDUMP) mentioned in Appendix C of the IBM publication "IBM System/360 and System/370 FORTRAN IV Language", form GC28-6515. The coding used for the double precision versions of the mathematical functions is essentially that used with IBM's FORTRAN library (without the Extended Error Handling Facility). Consequently, the algorithms used and error estimates for these routines may be found in the IBM publication "FORTRAN IV Library - Mathematical and Service Subprograms", form GC28-6818.

The following additional points should be noted. Single-precision versions of many of the mathematical functions used in WATFIV produce the truncated value of the

corresponding double-precision version. (Exceptions are the functions such as ABS, MOD, FLOAT, etc., which don't require complicated approximation formulae.) For example, the evaluation of SQRT by WATFIV is essentially equivalent to

```
SQRT(X)=SNGL(DSQRT(DBLE(X)))
```

12.3 AUTOMATIC FUNCTION TYPING

Since the initial release of WATFIV, the method of handling FORTRAN built-in functions has been incompatible with all of IBM's FORTRAN compilers. The major problem encountered is WATFIV's requirement that the type of these functions must be explicitly declared if it is different than can be assumed from the implicit rules. This restriction has now been removed and the method of handling FORTRAN built-in functions conforms to the current FORTRAN standards.

The following example shows that the DSQRT function need not be explicitly declared REAL*8.

```
$JOB     WATFIV
         REAL*8 VALUE,X(100),X(100)
         .
         .
         .
         VALUE(I)=X(I)*DSQRT(Y(I))
         .
         .
         .
         END
$ENTRY
```

To determine if a built-in (intrinsic) function is being invoked, the following requirements must be met:

1) The name of the function must not appear in an EXTERNAL statement. It may not be the name of an array, a character variable, a subprogram, or a statement function.

2) This name may not appear in a specification statement of type different from that of the function specified in the list of FORTRAN Built-in Functions (see Appendix A of FORTRAN IV with WATFOR and WATFIV - Cress/Dirksen/Graham).

3) The appearance of the symbol name (except in a type statement as described in 2) must be followed immediately by an actual argument list enclosed in parentheses.

Essentially, if you wish to use a function with the same name as the built-in function and it is not supplied in your source deck, then you must specify the name in an EXTERNAL statement to direct the compiler to use the function you supplied.

12.4 SUBPROGRAM ARGUMENTS

The rules for passing values between subprograms are generally the same as those described in the IBM publication "IBM System /360 FORTRAN IV Language", form GC28-6515. The relevant sections in that manual are "Arguments in a Function or Subroutine Subprogram", "Multiple Entry into a Subprogram", "Object-time Dimensions". The following remarks augment the rules stated in GC28-6515.

If a dummy argument of a called subprogram is an array, then GC28-6515 specifies that the corresponding actual argument provided by a calling routine must be (1) an array name, or (2) an array element. Furthermore, in case (1) the size of the dummy array as declared in the called subprogram must not exceed the size of the actual array provided by the calling subprogram. (here 'size' means amount, in bytes, of memory allocated.) In case (2), the size of the dummy array must not exceed the size of that portion of the actual array that follows and includes the specified element.

WATFIV allows a third possibility, namely, that the actual argument may be a simple variable (or expression). The rule is similar to that of case (1); the size of the dummy array must not exceed the number of bytes occupied by the simple variable.

All three rules can be stated more briefly, if somewhat less precisely, by the single rule that the dummy array must fit into the space provided by the actual argument, i.e., the dummy array may be smaller, but may not be larger. These rules are in the language presumably so that programmers will not index beyond the confines of an array, thus possibly clobbering other data or program areas. WATFIV takes the trouble to make sure the rules are not violated at execution time by making checks on arguments that are passed to dummy arrays. If a rule is violated, the program is presumed to be at fault, and is terminated with an error message and a subprogram traceback.

An example of case (2) follows in which the dummy array is smaller than the actual array. Note that, according to the

rules, B could be dimensioned at, but not greater than, 76.

```
      DIMENSION A(100)
        .
        .
        .
      CALL RTN (A(25))
        .
        .
        .
      END
      SUBROUTINE RTN(B)
      DIMENSION B(50)
        .
        .
        .
      END
```

Object-time dimensions can be very useful for creating subprograms for which it is not known beforehand what dimensions should be used for dummy arrays. See the following example.

```
      C**  ADVERTISEMENT FOR OBJECT-TIME DIMENSIONS
           DIMENSION A(100)
             .
             .
             .
           CALL RTN (A(25),76)
             .
             .
             .
           CALL RTN (A(I),101-I)
             .
             .
             .
           END
           SUBROUTINE RTN(B,N)
           DIMENSION B(N)
             .
             .
             .
           END
```

The following remarks pertain to the use of Hollerith constants as subprogram arguments. Since CHARACTER variables are implemented in WATFIV, a Hollerith (or CHARACTER) constant should be passed to a dummy argument which is a CHARACTER variable of appropriate length. This is merely an application of the general rule that an actual argument should agree in type and length with its corresponding dummy argument. An example follows.

```
      .
      .
      .
CALL RTN('LENGTHIS9')
      .
      .
      .
END
SUBROUTINE RTN(X)
CHARACTER*9 X
      .
      .
      .
```

However, to allow some compatibility with existing programs, Hollerith constants used as subprogram arguments are also treated in the following way. The compiler pads the constant on the right, with blanks, to make its length a multiple of four, if necessary. It is then treated as a vector, with a dimension equal to the number of words the constant occupies. Thus, the corresponding dummy argument must be a vector of appropriate dimension. The following example illustrates this.

```
      .
      .
      .
CALL RTN('LENGTHIS9',3)
      .
      .
      .
END
SUBROUTINE RTN(I,N)
DIMENSION I(N)
      .
      .
      .
```

Hollerith constants are always aligned on a word boundary.

12.5 USER LIBRARIES

As mentioned above, WATFIV will retrieve subprograms from a direct-access library. In fact, the FORTRAN-supplied subprograms not kept in core are handled this way. The mechanism for retrieving subprograms is sufficiently general that it will retrieve subprograms from communal or installation-supplied libraries.

For assistance on how to set up and specify libraries,

contact the system programmer responsible for WATFIV at your installation. Complete details are given in section 4.1 of the WATFIV Implementation Guide.

12.6 PSEUDO-VARIABLE DIMENSIONING

Certain distributed subroutine packages such as IMSL (International Mathematical and Statistical Library) produced execution-time error messages under WATFIV when some dummy parameters in the subroutines were dimensioned at 1. To eliminate this problem, the concept of pseudo-variable dimensioning (or PVD) was implemented.

WATFIV has been modified to internally generate information about all dummy array arguments whose last dimension is 1. Upon invocation of a subroutine, the total array storage of the calling argument and the dummy parameters are calculated. If the dummy array does not fit into the space provided by the actual argument (that is, the dummy array is smaller) and its last subscript is 1, then the last subscript declared for the dummy array is changed internally so that the storage required for both arrays is the same.

Consider the following programmes:

Example 1:

```
$JOB   WATFIV  P1234J.USER
       REAL A(10),A1(25)
       CALL SUB1(A)
       CALL SUB1(A1)
       STOP
       END

       SUBROUTINE SUB1(B)
       REAL B(1)
       DO 20 I=1,30
20     B(I)=FLOAT(I)
       RETURN
       END
$ENTRY
```

When running Example 1 under V1L4 (the previous version of WATFIV) the error message "SUBSCRIPT NUMBER 1 OF B HAS THE VALUE 2" would be issued when attempting to access B(2). Under the new version, the dimension of B is set to 10 when passing argument A, and is set to 25 when A1 is passed. This pseudo-variable dimensioning only takes effect when the parameter B has a last dimension of 1. If argument A were passed to SUB1, and a reference to B(11) was made in SUB1, the error message "SUBSCRIPTS EXCEED BOUNDS OF ACTUAL ARRAY"

would be generated. A similar message is issued for argument A1 if the program attempts to modify the 26th element of B.

Example 2:

```
$JOB   WATFIV  P1234J.USER
       REAL A(100)
       CALL SUB1(A)
       CALL SUB2(A)
       STOP
       END

       SUBROUTINE SUB1(B)
       REAL B(5,1)
       DO 20 I=1,20
20     B(5,I)=0.0
       RETURN
       END

       SUBROUTINE SUB2(C)
       REAL C(2,2)
       DO 10 I=1,10
       DO 10 J=1,10
10     C(I,J)=0.0
       RETURN
       END
$ENTRY
```

In Example 2, the size of array B in subroutine SUB1 would be modified to 5 rows and 20 columns while the dimension of C in subroutine SUB2 would remain the same and an error message would result when C(1,3) was referenced. This demonstrates that PVD takes effect only when the last dimension of a dummy parameter is 1.

Example 3:

```
$JOB   WATFIV P1234J.USER
       REAL A(3)
       CALL SUB1(A)
       STOP
       END

       SUBROUTINE SUB1(B)
       COMPLEX B(1)
       B(1)=(1.0,2.0)
       B(2)=3.0,4.0)
       RETURN
       END
$ENTRY
```

In Example 3, the size of A in the calling program is 12 bytes. In the subroutine, the dummy argument B is COMPLEX, and will therefore occupy 8 bytes if dimensioned at 1, or 16 bytes, if dimensioned at 2. It will be dimensioned at 1 to fit into the space provided by the calling array, and hence an error message will result when B (2) is referenced.

Although this feature is transparent to existing programmes, it allows the programmer to use subroutine packages such as IMSL without modification. The concept of PVD (although incompatible with standard FORTRAN) eliminates the need for passing down variable dimensions as arguments or the need to restrict the size of arrays that subroutines can process.

WATFIV now permits the specification of two new options on either the $JOB or C$OPTIONS cards. These options, "NOSUB" and "SUB", will only be used while PVD is in effect. The NOSUB job card parameter permits the user to access any member of a dummy array as long as this array element is within the storage reserved for the calling array. The SUB option forces the user to make sure that the subscripts specified for an array element do not exceed the corresponding subscripts specified in the declaration of the dummy array.

The following example illustrates the different error messages received for this option.

```
$JOB     WATFIV
         REAL A(10,10)/100*0./
         CALL SUB1(A)
         STOP
         END

         SUBROUTINE SUB1(B)
         REAL B(10,1)
         B(50,2)=1.
         RETURN
         END
$ENTRY
```

Since SUB is the default, this job will receive the error message "SUBSCRIPT NUMBER 1 OF B HAS THE VALUE 50" when attempting to access B(50,2). Specifying the NOSUB option will permit the programmer to specify any subscript for array B as long as the array element falls within the area defined by A. Attempting to access B(51,2) will cause the error message "ARRAY BOUNDS EXCEEDED FOR ARRAY B".

When the SUB option is activated (that is, array subscript checking is in effect) the method of checking subscripts is done from right to left due to the design of the WATFIV

compiler.

Consider the following program:

```
$JOB     WATFIV
         REAL A(1,2,1,3,1,2,3)
         CALL SUB1(A)
         STOP
         END

         SUBROUTINE SUB1(B)
         REAL B(1,2,1,2,1,2,1)
         B(91,92,93,94,95,96,3)=1.1
         RETURN
         END
```

Since subscript checking is in effect, the first subscript of B, which has the value 91 should be diagnosed as invalid. However the array size calculations are done from right to left for variable dimensioning and thus the message "SUBSCRIPT NUMBER 6 OF B HAS THE VALUE 96" will be issued.

Finally, an extension message of the form "PSEUDO VARIABLE DIMENSIONING ASSUMED FOR ARRAY B" is now issued to inform users when PVD is in effect.

12.7 SUBPROGRAMS IN OBJECT DECK FORM

WATFIV will accept subprograms in object deck form from either the input stream (SYSIN) or libraries. In fact, all routines in the library, WATFIV.FUNLIB, of FORTRAN-supplied subprograms are in object deck form.

A subprogram in object deck form may appear in any place that a subprogram in source form may appear, but object decks are never listed. The example below shows a job composed of a main program and two subprograms, R1 and R2, in object deck and source form, respectively.

```
$JOB          id,parameters
               .
               .
               .
              CALL R2(A)           Main program
               .
               .
               .
              END


                     Object deck for R1


              SUBROUTINE R2(X)
               .
               .
               .
              Y=R1(X)              R2 in source form
               .
               .
               .
              END
              $ENTRY


                          Any data
```

The question naturally arises: "May object decks acquired from the IBM FORTRAN compilers be used?" The answer is: "Under certain circumstances." However, the circumstances are so restrictive that, effectively, the answer is: "No". The intention is that the object-deck loading facility of WATFIV will be used with special-purpose routines, e.g., plotter routines, hand-coded in Assembler language.

Since the calling-sequence conventions are not unlike those used with the IBM FORTRAN compilers, anyone who has coded assembler subroutines before should have little difficulty adapting the subprograms for use with WATFIV. Complete details can be found in section 4.3 of the WATFIV Implementation Guide.

12.8 ADDITIONAL SUBPROGRAMS SUPPORTED BY WATFIV

12.8.1 SPECIAL FUNCTIONS

WATFIV supports the four function subprograms described in the following table.

The term "word length" refers to any type of variable which occupies four bytes, e.g., INTEGER*4, REAL*4, LOGICAL*4, CHARACTER*4, etc. All 32 bits of each argument are used in composing the result of the function evaluation.

Function Name	Purpose	Number of Arguments	Type of Arguments	Type of Result
AND	Logical 'and' of arguments	2 or more	Word length	REAL*4
OR	Logical 'or' of arguments	2 or more	Word length	REAL*4
EOR	Exclusive 'or' of arguments	2 or more	Word length	REAL*4
COMPL	Logical 1's complement of argument	1	Word length	REAL*4

12.8.2 STATEMENT COMPRESS/UNCOMPRESS ROUTINES

FIVPAK is a subroutine that compresses 'one statement per card' FORTRAN source decks into 'multi-statements per card' decks usable in WATFIV. (UNPACK reverses the process.)

The compressed form of source input is efficient if programs are to be stored in source form in data sets on disks since the results are:

(a) faster compile time

(b) less disk space required

Method:

Blanks are removed from all FORTRAN statements, except where they are embedded between apostrophies. Comment cards are

reproduced as read.

```
      DATA A,B/2H *,' *'/
      X=5.0
36    GO TO (3,8),I
```

is compressed into,

```
      DATAA,B/2H *,' *'/;X=5.0;36:GOTO(3,8),I
```

The cards produced are sequence numbered in increments of 10.

How to Use

CALL FIVPAK(NREAD,NPUNCH), or CALL UNPACK(NREAD,NPUNCH)

where

NREAD = unit number for input data

NPUNCH = unit number for output data

Both programs must be called from a program run under WATFIV[1], since they use CHARACTER variables. To read cards from the reader and punch a new deck, the following job may be run:

```
$JOB  id,KP=26,NOWARN
      CALL FIVPAK(5,7)
      STOP
      END
$ENTRY
    one-statement-per-card deck to be compressed
```

NOTE: More than one program deck may be compressed using FIVPAK by placing a card with an asterisk (*) in column 1

---

(1) FIVPAK and UNPACK reside in WATFIV's source library, WATFIV.WATLIB.

between each complete deck. UNPACK does not require such a "separator" card. The output from FIVPAK or UNPACK will be produced on the unit specified by NPUNCH as well as the printer.

13. RETURN CODES

A return code is provided by the compiler after a batch of WATFIV jobs has been executed. The highest return code generated by any job in the batch is returned.

RETURN CODE	EXPLANATION
0	End of batch; no non-zero return codes generated (no diagnostics of any type were generated for all the jobs in the batch)
1	Extension at compile time
2	Warning at compile time or execution time
3	Error at compile time
4	Error at execution time
5	Compiler error - remainder of jobs in batch abandoned; compiler termination successful, i.e., files closed, dynamic areas freed.
8	Compiler error - remainder of jobs in batch abandoned; compiler termination may be unsuccessful.

These return codes have been chosen to give the programmer control over executing the next step when running under OS/VS. The return code should be used in conjunction with the COND parameter on the EXEC card to specify conditions under which the step is not to be executed.

14. MISCELLANEOUS

14.1 CARRIAGE-CONTROL CHARACTERS

WATFIV will replace, without warning, invalid carriage control characters by blanks. Valid carriage control characters, with corresponding meanings, are:

blank	Advance one line before printing
0	Advance two lines before printing
-	Advance three lines before printing
1	Advance to first line of next page
+	No advance

Note that both EBCDIC and BCD '+' are supported.

14.2 TREATMENT OF LOGICAL VALUES

If a logical variable has been assigned a value of .TRUE. or .FALSE., a T or F, respectively, will be printed for the variable under L format. WATFIV also considers two other cases: if the variable has not been assigned a value, i.e., is 'undefined', a U is printed. If a value has been assigned but it is not the internal representation of .TRUE. or .FALSE., a J (for Junk) is printed. The latter case could arise through improper use of EQUIVALENCE.

Note that WATFIV uses only the high-order byte of a four-byte logical variable in computations. For example, if A and B are four-byte logical variables, then the statement

```
A=B
```

involves the movement of only one byte in memory.

14.3 CHARACTER-SET CONVENTIONS

WATFIV allows a program to be punched on either the Model 029 or 026 keypunches, i.e., EBCDIC or BCD modes. Intermixing of EBCDIC and BCD within a program is allowed, subject to the following restrictions.

(1) The user specifies by the KP= parameter on the $JOB or C$OPTIONS card the keypunch mode to be considered as the principal mode for the program.

(2) The left parenthesis, right parenthesis, equal sign of either character set may be used. A warning message is issued so that the card could be repunched for a subsequent recompilation under the G or H compilers.

(3) Quote marks may not be intermixed. If KP=29 is specified or assumed by default, then the EBCDIC quote or apostrophe (') must be used when delimiting Hollerith constants or as the unit number/record number separator in direct-access I/O statements, e.g.,

```
FORMAT('HOLLERITH INFORMATION')
FIND(3'I)
```

If KP=26 is specified or assumed, the BCD quote @ must be used, e.g.,

```
FORMAT(@HOLLERITH INFORMATION@)
FIND(3@I)
```

(4) If KP=29 is specified or assumed, then the EBCDIC & (12 punch) and the BCD + (12 punch) are taken as the statement number argument indicator, e.g., CALL RTN(&2). The only 'plus' sign is the EBCDIC + (12-8-6 punch).

If KP=26 is specified or assumed, then the EBCDIC +, BCD +, and EBCDIC & are taken as 'plus'. To indicate a statement number argument, use a $, i.e., the IBM compiler convention, e.g., CALL RTN($2).

14.4 INCOMPATIBILITIES WITH IBM FORTRAN

Note that the differences listed below do not include the language extensions and restrictions given in Chapter 7 on page 46. Nor do they include differences which arise either because object programs compiled under IBM FORTRAN are freely allowed to violate the language rules defined in GC28-6515 (e.g., passing an argument of type INTEGER to the SQRT subroutine), or because the IBM compilers accept syntax

not defined in GC28-6515, e.g.,

```
WRITE(6,2) (A(I), A(2))
```

The major causes of differences between WATFIV and IBM FORTRAN are likely to be the treatment of FORTRAN-supplied functions and number conversions.

1. WATFIV provides execution-time page skipping, controlled by the LINES= job-parameter.

2. WATFIV allows any number of contiguous comments cards; comments cards may precede a continuation card.

3. WATFIV uses only the high-order byte of a logical quantity in logical operations. For example, if A and B are of type LOGICAL *4, execution of the statement

   ```
   A=B
   ```

   causes only one byte to be moved.

4. DO-loops may be nested to any depth in WATFIV.

5. WATFIV supports both EBCDIC and BCD '+' as carriage control characters.

6. WATFIV considers the program to be in error if it executes a RETURNi statement in which the value of 'i' is zero, negative, undefined, or greater than the number of statement number arguments which appeared in the argument list of the CALL statement which invoked the subprogram from which the return is being made.

7. WATFIV prints no message equivalent to the IHC210I ("old PSW is ...") message when an interrupt occurs.

8. With WATFIV, a use of T format that does a 'backward' tab in an output buffer does not cause existing characters in the buffer to be blanked out. For example, consider the statements:

   ```
            K= 9
            J= 1
            WRITE (6,7)K,J
      7     FORMAT (' $$$.00',T3,I2,T6,I2)
   ```

   With WATFIV, the line appears as:

   ```
   $$9.01
   ```

With IBM FORTRAN, it appears as:

```
$ 9. 1
```

Actually, this is a consequence of the fact that WATFIV's formatting routines assume the buffer to be blanked before any filling of it occurs, i.e., only significant characters are moved into the buffer.

9. REAL*4 values are printed with a maximum of 7 significant digits. If the output format specification calls for more, i.e., E20.10, zeroes are supplied on the right.

10. WATFIV treats FORTRAN-supplied functions differently than IBM FORTRAN as follows:

    (a) WATFIV makes no distinction between 'in-line' and 'out-of-line' functions; all functions are out-of-line and thus no code is generated at compile time.

    (b) WATFIV evaluates all functions that require approximation formulae in double precision, i.e.,

    ```
    SQRT(X)
    ```

    is calculated as, essentially,

    ```
    SNGL(DSQRT(DBLE(X))).
    ```

11. WATFIV handles FORMAT statements differently than G and H as follows:

    (a) WATFIV allows more than the maximum number of continuation cards for FORMAT statements.

    (b) WATFIV does not allow group or field counts to be zero.

12. Execution-time data cards read on the standard card reader unit by WATFIV-compiled programs may not contain a $ in column 1 or C$ in columns 1-2.

13. With WATFIV, a particular labelled COMMON block can be initialized in more than one BLOCK DATA subprogram. This allows undetected violations of rule 6, page 112 of GC28-6515-10.

15. APPENDIX

15.1 WATFIV ERROR MESSAGES

```
'ASSEMBLER LANGUAGE SUBPROGRAMMES'
AL-0   'MISSING END CARD ON ASSEMBLY LANGUAGE OBJECT DECK'
AL-1   'ENTRY-POINT OR CSECT NAME IN AN OBJECT DECK WAS PREVIOUSLY
       DEFINED.FIRST DEFINITION USED'

'BLOCK DATA STATEMENTS'
BD-0   'EXECUTABLE STATEMENTS ARE ILLEGAL IN BLOCK DATA SUBPROGRAMS'
BD-1   'IMPROPER BLOCK DATA STATEMENT'

'CARD FORMAT AND CONTENTS'
CC-0   'COLUMNS 1-5 OF CONTINUATION CARD ARE NOT BLANK.
       PROBABLE CAUSE:STATEMENT PUNCHED TO LEFT OF COLUMN 7'
CC-1   'LIMIT OF 5 CONTINUATION CARDS EXCEEDED'
CC-2   'INVALID CHARACTER IN FORTRAN STATEMENT.A'$' WAS INSERTED IN THE
       SOURCE LISTING'
CC-3   'FIRST CARD OF A PROGRAM IS A CONTINUATION CARD.
       PROBABLE CAUSE:STATEMENT PUNCHED TO LEFT OF COLUMN 7'
CC-4   'STATEMENT TOO LONG TO COMPILE (SCAN-STACK OVERFLOW)'
CC-5   'A BLANK CARD WAS ENCOUNTERED'
CC-6   'KEYPUNCH USED DIFFERS FROM KEYPUNCH SPECIFIED ON JOB CARD'
CC-7   'THE FIRST CHARACTER OF THE STATEMENT WAS NOT ALPHABETIC'
CC-8   'INVALID CHARACTER(S) ARE CONCATENATED WITH THE FORTRAN KEYWORD'
CC-9   'INVALID CHARACTERS IN COLUMNS 1-5.STATEMENT NUMBER IGNORED.
       PROBABLE CAUSE:STATEMENT PUNCHED TO LEFT OF COLUMN 7'
CC-A   'CONTROL CARDS MAY NOT BE CONTINUED'
CC-B   'CONTROL CARDS MUST BE IN PROGRAM SEGMENT'

'COMMON'
CM-0   'THE VARIABLE IS ALREADY IN COMMON'
CM-1   'OTHER COMPILERS MAY NOT ALLOW COMMONED VARIABLES TO BE INITIALIZED IN
       OTHER THAN A BLOCK DATA SUBPROGRAM'
CM-2   'ILLEGAL USE OF A COMMON BLOCK OR NAMELIST NAME'

'FORTRAN TYPE CONSTANTS'
CN-0   'MIXED REAL*4,REAL*8 IN COMPLEX CONSTANT;REAL*8 ASSUMED FOR BOTH'
CN-1   'AN INTEGER CONSTANT MAY NOT BE GREATER THAN 2,147,483,647 (2**31-1)'
CN-2   'EXPONENT ON A REAL CONSTANT IS GREATER THAN 2 DIGITS'
CN-3   'A REAL CONSTANT HAS MORE THAN 16 DIGITS.IT WAS TRUNCATED TO 16'
CN-4   'INVALID HEXADECIMAL CONSTANT'
CN-5   'ILLEGAL USE OF A DECIMAL POINT'
CN-6   'CONSTANT WITH MORE THAN 7 DIGITS BUT E-TYPE EXPONENT,ASSUMED TO BE
       REAL*4'
CN-7   'CONSTANT OR STATEMENT NUMBER GREATER THAN 99999'
CN-8   'AN EXPONENT OVERFLOW OR UNDERFLOW OCCURRED WHILE CONVERTING A CONSTANT
       IN A SOURCE STATEMENT'
```

```
'COMPILER ERRORS'
CP-0   'COMPILER ERROR - LANDR/ARITH'
CP-1   'COMPILER ERROR.LIKELY CAUSE:MORE THAN 255 DO STATEMENTS'
CP-2   'COMPILER ERROR'
CP-4   'COMPILER ERROR - INTERRUPT AT COMPILE TIME,RETURN TO SYSTEM'

'CHARACTER VARIABLE'
CV-0   'A CHARACTER VARIABLE IS USED WITH A RELATIONAL OPERATOR'
CV-1   'LENGTH OF A CHARACTER VALUE ON RIGHT OF EQUAL SIGN EXCEEDS THAT ON
       LEFT. TRUNCATION WILL OCCUR'
CV-2   'UNFORMATTED CORE-TO-CORE I/O NOT IMPLEMENTED'

'DATA STATEMENT'
DA-0   'REPLICATION FACTOR IS ZERO OR GREATER THAN 32767.
       IT IS ASSUMED TO BE 32767'
DA-1   'MORE VARIABLES THAN CONSTANTS'
DA-2   'ATTEMPT TO INITIALIZE A SUBPROGRAM PARAMETER IN A DATA STATEMENT'
DA-3   'OTHER COMPILERS MAY NOT ALLOW NON-CONSTANT SUBSCRIPTS IN DATA
       STATEMENTS'
DA-4   'TYPE OF VARIABLE AND CONSTANT DO NOT AGREE.  (MESSAGE ISSUED ONCE FOR
       AN ARRAY)'
DA-5   'MORE CONSTANTS THAN VARIABLES'
DA-6   'A VARIABLE WAS PREVIOUSLY INITIALIZED.THE LATEST VALUE IS USED.
       CHECK COMMONED AND EQUIVALENCED VARIABLES'
DA-7   'OTHER COMPILERS MAY NOT ALLOW INITIALIZATION OF BLANK COMMON'
DA-8   'A LITERAL CONSTANT HAS BEEN TRUNCATED'
DA-9   'OTHER COMPILERS MAY NOT ALLOW IMPLIED DO-LOOPS IN DATA STATEMENTS'

'DEFINE FILE STATEMENTS'
DF-0   'THE UNIT NUMBER IS MISSING'
DF-1   'INVALID FORMAT TYPE'
DF-2   'THE ASSOCIATED VARIABLE IS NOT A SIMPLE INTEGER VARIABLE'
DF-3   'NUMBER OF RECORDS OR RECORD SIZE IS ZERO OR GREATER THAN 32767'

'DIMENSION STATEMENTS'
DM-0   'NO DIMENSIONS ARE SPECIFIED FOR A VARIABLE IN A DIMENSION STATEMENT'
DM-1   'THE VARIABLE HAS ALREADY BEEN DIMENSIONED'
DM-2   'CALL-BY-LOCATION PARAMETERS MAY NOT BE DIMENSIONED'
DM-3   'THE DECLARED SIZE OF ARRAY EXCEEDS SPACE PROVIDED BY CALLING ARGUMENT'

'DO LOOPS'
DO-0   'THIS STATEMENT CANNOT BE THE OBJECT OF A DO-LOOP'
DO-1   'ILLEGAL TRANSFER INTO THE RANGE OF A DO-LOOP'
DO-2   'THE OBJECT OF THIS DO-LOOP HAS ALREADY APPEARED'
DO-3   'IMPROPERLY NESTED DO-LOOPS'
DO-4   'ATTEMPT TO REDEFINE A DO-LOOP PARAMETER WITHIN THE RANGE OF THE LOOP'
DO-5   'INVALID DO-LOOP PARAMETER'
DO-6   'ILLEGAL TRANSFER TO A STATEMENT WHICH IS INSIDE THE RANGE OF A DO-LOOP'
```

```
DO-7   'A DO-LOOP PARAMETER IS UNDEFINED OR OUT OF RANGE'
DO-8   'BECAUSE OF ONE OF THE PARAMETERS,THIS DO-LOOP WILL TERMINATE AFTER THE
       FIRST TIME THROUGH'
DO-9   'A DO-LOOP PARAMETER MAY NOT BE REDEFINED IN AN INPUT LIST'
DO-A   'OTHER COMPILERS MAY NOT ALLOW THIS STATEMENT TO END A DO-LOOP'

'EQUIVALENCE AND/OR COMMON'
EC-0   'EQUIVALENCED VARIABLE APPEARS IN A COMMON STATEMENT'
EC-1   'A COMMON BLOCK HAS A DIFFERENT LENGTH THAN IN A PREVIOUS
       SUBPROGRAM:GREATER LENGTH USED'
EC-2   'COMMON AND/OR EQUIVALENCE CAUSES INVALID ALIGNMENT.
       EXECUTION SLOWED.REMEDY:ORDER VARIABLES BY DECREASING LENGTH'
EC-3   'EQUIVALENCE EXTENDS COMMON DOWNWARDS'
EC-4   'A SUBPROGRAM PARAMETER APPEARS IN A COMMON OR EQUIVALENCE STATEMENT'
EC-5   'A VARIABLE WAS USED WITH SUBSCRIPTS IN AN EQUIVALENCE STATEMENT BUT HAS
       NOT BEEN PROPERLY DIMENSIONED'

'END STATEMENTS'
EN-0   'MISSING END STATEMENT:END STATEMENT GENERATED'
EN-1   'AN END STATEMENT WAS USED TO TERMINATE EXECUTION'
EN-2   'AN END STATEMENT CANNOT HAVE A STATEMENT NUMBER. STATEMENT NUMBER
       IGNORED'
EN-3   'END STATEMENT NOT PRECEDED BY A TRANSFER'

'EQUAL SIGNS'
EQ-0   'ILLEGAL QUANTITY ON LEFT OF EQUALS SIGN'
EQ-1   'ILLEGAL USE OF EQUAL SIGN'
EQ-2   'OTHER COMPILERS MAY NOT ALLOW MULTIPLE ASSIGNMENT STATEMENTS'
EQ-3   'MULTIPLE ASSIGNMENT IS NOT IMPLEMENTED FOR CHARACTER VARIABLES'
EQ-4   'ILLEGAL QUANTITY ON RIGHT OF EQUALS SIGN'

'EQUIVALENCE STATEMENTS'
EV-0   'ATTEMPT TO EQUIVALENCE A VARIABLE TO ITSELF'
EV-2   'A MULTI-SUBSCRIPTED EQUIVALENCED VARIABLE HAS BEEN INCORRECTLY
       RE-EQUIVALENCED.REMEDY:DIMENSION THE VARIABLE FIRST'

'POWERS AND EXPONENTIATION'
EX-0   'ILLEGAL COMPLEX EXPONENTIATION'
EX-1   'I**J WHERE I=J=0'
EX-2   'I**J WHERE I=0, J.LT.0'
EX-3   '0.0**Y WHERE Y.LE.0.0'
EX-4   '0.0**J WHERE J=0'
EX-5   '0.0**J WHERE J.LT.0'
EX-6   'X**Y WHERE X .LT. 0.0, Y IS NOT TYPE INTEGER OR .NE. 0.0'
```

```
'ENTRY STATEMENT'
EY-0   'ENTRY-POINT NAME WAS PREVIOUSLY DEFINED'
EY-1   'PREVIOUS DEFINITION OF FUNCTION NAME IN AN ENTRY IS INCORRECT'
EY-2   'THE USAGE OF A SUBPROGRAM PARAMETER IS INCONSISTENT WITH A PREVIOUS
       ENTRY-POINT'
EY-3   'A PARAMETER HAS APPEARED IN A EXECUTABLE STATEMENT BUT IS NOT A
       SUBPROGRAM PARAMETER'
EY-4   'ENTRY STATEMENTS ARE INVALID IN THE MAIN PROGRAM'
EY-5   'ENTRY STATEMENT INVALID INSIDE A DO-LOOP'

'FORMAT'
   SOME FORMAT ERROR MESSAGES GIVE CHARACTERS IN WHICH ERROR WAS DETECTED
FM-0   'IMPROPER CHARACTER SEQUENCE OR INVALID CHARACTER IN INPUT DATA'
FM-1   'NO STATEMENT NUMBER ON A FORMAT STATEMENT'
FM-2   'FORMAT CODE AND DATA TYPE DO NOT MATCH'
FM-4   'FORMAT PROVIDES NO CONVERSION SPECIFICATION FOR A VALUE IN I/O LIST'
FM-5   'AN INTEGER IN THE INPUT DATA IS TOO LARGE.
         (MAXIMUM=2,147,483,647=2**31-1)'
FM-6   'A REAL NUMBER IN THE INPUT DATA IS OUT OF MACHINE RANGE (1.E-78,1.E+75)'
FM-7   'UNREFERENCED FORMAT STATEMENT'
FT-0   'FIRST CHARACTER OF VARIABLE FORMAT IS NOT A LEFT PARENTHESIS'
FT-1   'INVALID CHARACTER ENCOUNTERED IN FORMAT'
FT-2   'INVALID FORM FOLLOWING A FORMAT CODE'
FT-3   'INVALID FIELD OR GROUP COUNT'
FT-4   'A FIELD OR GROUP COUNT GREATER THAN 255'
FT-5   'NO CLOSING PARENTHESIS ON VARIABLE FORMAT'
FT-6   'NO CLOSING QUOTE IN A HOLLERITH FIELD'
FT-7   'INVALID USE OF COMMA'
FT-8   'FORMAT STATEMENT TOO LONG TO COMPILE (SCAN-STACK OVERFLOW)'
FT-9   'INVALID USE OF P FORMAT CODE'
FT-A   'INVALID USE OF PERIOD(.)'
FT-B   'MORE THAN THREE LEVELS OF PARENTHESES'
FT-C   'INVALID CHARACTER BEFORE A RIGHT PARENTHESIS'
FT-D   'MISSING OR ZERO LENGTH HOLLERITH ENCOUNTERED'
FT-E   'NO CLOSING RIGHT PARENTHESIS'
FT-F   'CHARACTERS FOLLOW CLOSING RIGHT PARENTHESIS'
FT-G   'WRONG QUOTE USED FOR KEY-PUNCH SPECIFIED'
FT-H   'LENGTH OF HOLLERITH EXCEEDS 255'
FT-I   'EXPECTING COMMA BETWEEN FORMAT ITEMS'

'FUNCTIONS AND SUBROUTINES'
FN-1   'A PARAMETER APPEARS MORE THAN ONCE IN A SUBPROGRAM OR STATEMENT
       FUNCTION DEFINITION'
FN-2   'SUBSCRIPTS ON RIGHT-HAND SIDE OF STATEMENT FUNCTION.
        PROBABLE CAUSE:VARIABLE TO LEFT OF EQUAL SIGN NOT DIMENSIONED'
FN-4   'ILLEGAL LENGTH MODIFIER'
FN-5   'INVALID PARAMETER'
FN-6   'A PARAMETER HAS THE SAME NAME AS THE SUBPROGRAM'
```

```
'GO TO STATEMENTS'
GO-0   'THIS STATEMENT COULD TRANSFER TO ITSELF'
GO-1   'THIS STATEMENT TRANSFERS TO A NON-EXECUTABLE STATEMENT'
GO-2   'ATTEMPT TO DEFINE ASSIGNED GOTO INDEX IN AN ARITHMETIC STATEMENT'
GO-3   'ASSIGNED GOTO INDEX MAY BE USED ONLY IN ASSIGNED GOTO AND ASSIGN
       STATEMENTS'
GO-4   'INDEX OF AN ASSIGNED GOTO IS UNDEFINED OR OUT OF RANGE,OR INDEX OF
       COMPUTED GOTO OR CASE IS UNDEFINED'
GO-5   'ASSIGNED GOTO INDEX MAY NOT BE AN INTEGER*2 VARIABLE'

'HOLLERITH CONSTANTS'
HO-0   'ZERO LENGTH SPECIFIED FOR H-TYPE HOLLERITH'
HO-1   'ZERO LENGTH QUOTE-TYPE HOLLERITH'
HO-2   'NO CLOSING QUOTE OR NEXT CARD NOT A CONTINUATION CARD'
HO-3   'UNEXPECTED HOLLERITH OR STATEMENT NUMBER CONSTANT'

'IF STATEMENTS (ARITHMETIC AND LOGICAL)'
IF-0   'AN INVALID STATEMENT FOLLOWS THE LOGICAL IF'
IF-1   'ARITHMETIC OR INVALID EXPRESSION IN LOGICAL IF OR WHILE'
IF-2   'LOGICAL,COMPLEX OR INVALID EXPRESSION IN ARITHMETIC IF'

'IMPLICIT STATEMENT'
IM-0   'INVALID DATA TYPE'
IM-1   'INVALID OPTIONAL LENGTH'
IM-3   'IMPROPER ALPHABETIC SEQUENCE IN CHARACTER RANGE'
IM-4   'A SPECIFICATION IS NOT A SINGLE CHARACTER.THE FIRST CHARACTER IS USED'
IM-5   'IMPLICIT STATEMENT DOES NOT PRECEDE OTHER SPECIFICATION STATEMENTS'
IM-6   'ATTEMPT TO DECLARE THE TYPE OF A CHARACTER MORE THAN ONCE'
IM-7   'ONLY ONE IMPLICIT STATEMENT PER PROGRAM SEGMENT ALLOWED. THIS ONE
       IGNORED'

'INPUT/OUTPUT'
IO-0   'I/O STATEMENT REFERENCES NON-FORMAT STATEMENT. PROBABLE CAUSE :
       STATEMENT DEFINED AS NON-FORMAT'
IO-1   'A VARIABLE FORMAT MUST BE AN ARRAY NAME'
IO-2   'INVALID ELEMENT IN INPUT LIST OR DATA LIST'
IO-3   'OTHER COMPILERS MAY NOT ALLOW EXPRESSIONS IN OUTPUT LISTS'
IO-4   'ILLEGAL USE OF END= OR ERR= PARAMETERS'
IO-5   'INVALID UNIT NUMBER'
IO-6   'INVALID FORMAT'
IO-7   'ONLY CONSTANTS,SIMPLE INTEGER*4 VARIABLES,AND CHARACTER VARIABLES ARE
       ALLOWED AS UNIT'
IO-8   'ATTEMPT TO PERFORM I/O IN A FUNCTION WHICH IS CALLED IN AN OUTPUT
       STATEMENT'
IO-9   'UNFORMATTED WRITE STATEMENT MUST HAVE A LIST'
IO-A   'EXPECTING STATEMENT TO BE A FORMAT. PREVIOUSLY REFERENCED IN I/O
       STATEMENT'

'JOB CONTROL CARDS'
JB-0   'CONTROL CARD ENCOUNTERED DURING COMPILATION;
       PROBABLE CAUSE:MISSING C$ENTRY CARD'
JB-1   'MIS-PUNCHED JOB OPTION'
```

```
'JOB TERMINATION'
KO-0   'SOURCE ERROR ENCOUNTERED WHILE EXECUTING WITH RUN=FREE'
KO-1   'LIMIT EXCEEDED FOR FIXED-POINT DIVISION BY ZERO'
KO-2   'LIMIT EXCEEDED FOR FLOATING-POINT DIVISION BY ZERO'
KO-3   'EXPONENT OVERFLOW LIMIT EXCEEDED'
KO-4   'EXPONENT UNDERFLOW LIMIT EXCEEDED'
KO-5   'FIXED-POINT OVERFLOW LIMIT EXCEEDED'
KO-6   'JOB-TIME EXCEEDED'
KO-7   'COMPILER ERROR - EXECUTION TIME:RETURN TO SYSTEM'
KO-8   'TRACEBACK ERROR. TRACEBACK TERMINATED'
KO-9   'CANNOT OPEN WATFIV.ERRTEXTS. RUN TERMINATED'
KO-A   'I/O ERROR ON TEXT FILE'

'LOGICAL OPERATIONS'
LG-0   '.NOT. WAS USED AS A BINARY OPERATOR'

'LIBRARY ROUTINES'
LI-0   'ARGUMENT OUT OF RANGE DGAMMA OR GAMMA. (1.382E-76 .LT. X .LT. 57.57)'
LI-1   'ABS(X) .GE. 175.366 FOR SINH,COSH,DSINH OR DCOSH OF X'
LI-2   'SENSE LIGHT OTHER THAN 0,1,2,3,4 FOR SLITE OR 1,2,3,4 FOR SLITET'
LI-3   'REAL PORTION OF ARGUMENT .GT. 174.673, CEXP OR CDEXP'
LI-4   'ABS(AIMAG(Z)) .GT. 174.673 FOR CSIN, CCOS, CDSIN OR CDCOS OF Z'
LI-5   'ABS(REAL(Z)) .GE. 3.537E15 FOR CSIN, CCOS, CDSIN OR CDCOS OF Z'
LI-6   'ABS(AIMAG(Z)) .GE. 3.537E15 FOR CEXP OR CDEXP OF Z'
LI-7   'ARGUMENT .GT. 174.673, EXP OR DEXP'
LI-8   'ARGUMENT OF CLOG OR CDLOG IS ZERO'
LI-9   'ARGUMENT IS NEGATIVE OR ZERO, ALOG, ALOG10, DLOG OR DLOG10'
LI-A   'ABS(X) .GE. 3.537E15 FOR SIN, COS, DSIN OR DCOS OF X'
LI-B   'ABSOLUTE VALUE OF ARGUMENT .GT. 1, FOR ARSIN, ARCOS, DARSIN OR DARCOS'
LI-C   'ARGUMENT IS NEGATIVE, SQRT OR DSQRT'
LI-D   'BOTH ARGUMENTS OF DATAN2 OR ATAN2 ARE ZERO'
LI-E   'ARGUMENT TOO CLOSE TO A SINGULARITY, TAN, COTAN, DTAN OR DCOTAN'
LI-F   'ARGUMENT OUT OF RANGE DLGAMA OR ALGAMA. (0.0  .LT. X .LT. 4.29E73)'
LI-G   'ABSOLUTE VALUE OF ARGUMENT .GE. 3.537E15, TAN, COTAN, DTAN, DCOTAN'

'MIXED MODE'
MD-0   'RELATIONAL OPERATOR HAS LOGICAL OPERAND'
MD-1   'RELATIONAL OPERATOR HAS COMPLEX OPERAND'
MD-2   'MIXED MODE - LOGICAL OR CHARACTER WITH ARITHMETIC'
MD-3   'OTHER COMPILERS MAY NOT ALLOW SUBSCRIPTS OF TYPE COMPLEX,LOGICAL OR
       CHARACTER'

'MEMORY OVERFLOW'
MO-0   'INSUFFICIENT MEMORY TO COMPILE THIS PROGRAM.REMAINDER WILL BE ERROR
       CHECKED ONLY'
MO-1   'INSUFFICIENT MEMORY TO ASSIGN ARRAY STORAGE. JOB ABANDONED'
MO-2   'SYMBOL TABLE EXCEEDS AVAILABLE SPACE,JOB ABANDONED'
MO-3   'DATA AREA OF SUBPROGRAM EXCEEDS 24K -- SEGMENT SUBPROGRAM'
MO-4   'INSUFFICIENT MEMORY TO ALLOCATE COMPILER WORK AREA OR WATLIB BUFFER'
```

```
'NAMELIST STATEMENTS'
NL-0   'NAMELIST ENTRY MUST BE A VARIABLE,NOT A SUBPROGRAM PARAMETER'
NL-1   'NAMELIST NAME PREVIOUSLY DEFINED'
NL-2   'VARIABLE NAME TOO LONG'
NL-3   'VARIABLE NAME NOT FOUND IN NAMELIST'
NL-4   'INVALID SYNTAX IN NAMELIST INPUT'
NL-6   'VARIABLE INCORRECTLY SUBSCRIPTED'
NL-7   'SUBSCRIPT OUT OF RANGE'
NL-8   'NESTED BLANKS ARE ILLEGAL IN NAMELIST INPUT'

'PARENTHESES'
PC-0   'UNMATCHED PARENTHESIS'
PC-1   'INVALID PARENTHESIS NESTING IN I/O LIST'

'PAUSE, STOP STATEMENTS'
PS-0   'OPERATOR MESSAGES NOT ALLOWED:SIMPLE STOP ASSUMED FOR STOP,
       CONTINUE ASSUMED FOR PAUSE'

'RETURN STATEMENT'
RE-1   'RETURN I, WHERE I IS OUT OF RANGE OR UNDEFINED'
RE-2   'MULTIPLE RETURN NOT VALID IN FUNCTION SUBPROGRAM'
RE-3   'VARIABLE IS NOT A SIMPLE INTEGER'
RE-4   'A MULTIPLE RETURN IS NOT VALID IN THE MAIN PROGRAM'

'ARITHMETIC AND LOGICAL STATEMENT FUNCTIONS'
    PROBABLE CAUSE OF SF ERRORS - VARIABLE ON LEFT OF = WAS NOT DIMENSIONED
SF-1   'A PREVIOUSLY REFERENCED STATEMENT NUMBER APPEARS ON A STATEMENT
       FUNCTION DEFINITION'
SF-2   'STATEMENT FUNCTION IS THE OBJECT OF A LOGICAL IF STATEMENT'
SF-3   'RECURSIVE STATEMENT FUNCTION DEFINITION:NAME APPEARS ON BOTH SIDES OF
       EQUAL SIGN.LIKELY CAUSE:VARIABLE NOT DIMENSIONED'
SF-4   'A STATEMENT FUNCTION DEFINITION APPEARS AFTER THE FIRST EXECUTABLE
       STATEMENT'
SF-5   'ILLEGAL USE OF A STATEMENT FUNCTION NAME'

'STRUCTURED PROGRAMMING BLOCKS'
SP-0   'AT END STATEMENT MUST FOLLOW IMMEDIATELY AFTER A READ'
SP-1   'AT END FOLLOWS CORE TO CORE, DIRECT ACCESS OR INVALID READ STATEMENT'
SP-2   'AT END NOT VALID WHEN 'END=' SPECIFIED IN THE READ STATEMENT'
SP-3   'MISSING OR INVALID DO CASE, WHILE, AT END, OR  IF-THEN STATEMENT'
SP-4   'IMPROPER NESTING OF BLOCK OR CONSTRUCT'
SP-5   'IMPROPER NESTING OF DO-LOOP'
SP-6   'IMPROPER NESTING WITH DO-LOOP'
SP-7   'MISSING END CASE, END WHILE, END AT END, OR END IF STATEMENT'
SP-8   'OTHER COMPILERS MAY NOT ALLOW IF-THEN-ELSE, DO CASE, WHILE, EXECUTE,
       REMOTE BLOCK OR AT END STATEMENTS'
SP-9   'IF NONE BLOCK ALREADY DEFINED FOR CURRENT DO CASE CONSTRUCT'
SP-A   'IF NONE BLOCK MUST FOLLOW ALL CASE BLOCKS'
SP-B   'ATTEMPT TO TRANSFER CONTROL ACROSS REMOTE BLOCK BOUNDARIES'
SP-C   'REMOTE BLOCK NOT PRECEDED BY A TRANSFER'
```

```
SP-D   'REMOTE BLOCK PREVIOUSLY DEFINED'
SP-E   'REMOTE BLOCK STATEMENT MISSING OR INVALID'
SP-F   'LAST REMOTE BLOCK NOT COMPLETED'
SP-G   'REMOTE BLOCK IS NOT DEFINED'
SP-H   'REMOTE BLOCK IS NOT REFERENCED'
SP-I   'ATTEMPT TO NEST REMOTE BLOCK DEFINITIONS'
SP-J   'MISSING OR INVALID REMOTE BLOCK NAME'
SP-K   'ATTEMPT TO EXECUTE A REMOTE BLOCK RECURSIVELY'
SP-L   'NUMBER OF REMOTE BLOCKS EXCEEDS 255'

'SUBPROGRAMS'
SR-0   'MISSING SUBPROGRAM'
SR-1   'SUBPROGRAM REDEFINES A CONSTANT,EXPRESSION,DO-PARAMETER OR ASSIGNED
       GOTO INDEX'
SR-2   'THE SUBPROGRAM WAS ASSIGNED DIFFERENT TYPES IN DIFFERENT PROGRAM
       SEGMENTS'
SR-3   'ATTEMPT TO USE A SUBPROGRAM RECURSIVELY'
SR-4   'INVALID TYPE OF ARGUMENT IN REFERENCE TO A SUBPROGRAM'
SR-5   'WRONG NUMBER OF ARGUMENTS IN A REFERENCE TO A SUBPROGRAM'
SR-6   'A SUBPROGRAM WAS PREVIOUSLY DEFINED. THE FIRST DEFINITION IS USED'
SR-7   'NO MAIN PROGRAM'
SR-8   'ILLEGAL OR MISSING SUBPROGRAM NAME'
SR-9   'LIBRARY PROGRAM WAS NOT ASSIGNED THE CORRECT TYPE'
SR-A   'METHOD FOR ENTERING SUBPROGRAM PRODUCES UNDEFINED VALUE FOR
       CALL-BY-LOCATION PARAMETER'
SR-B   'MAINLINE PROGRAM NOT IN LIBRARY'

'SUBSCRIPTS'
SS-0   'ZERO SUBSCRIPT OR DIMENSION NOT ALLOWED'
SS-1   'ARRAY SUBSCRIPT EXCEEDS DIMENSION'
SS-2   'INVALID SUBSCRIPT FORM'
SS-3   'SUBSCRIPT IS OUT OF RANGE'
SS-4   'SUBSCRIPTS EXCEED BOUNDS OF ACTUAL ARRAY'

'STATEMENTS AND STATEMENT NUMBERS'
ST-0   'MISSING STATEMENT NUMBER'
ST-1   'STATEMENT NUMBER GREATER THAN 99999'
ST-2   'STATEMENT NUMBER HAS ALREADY BEEN DEFINED'
ST-3   'UNDECODEABLE STATEMENT'
ST-4   'UNNUMBERED EXECUTABLE STATEMENT FOLLOWS A TRANSFER'
ST-5   'STATEMENT NUMBER IN A TRANSFER IS A NON-EXECUTABLE STATEMENT'
ST-6   'ONLY CALL STATEMENTS MAY CONTAIN STATEMENT NUMBER ARGUMENTS'
ST-7   'STATEMENT SPECIFIED IN A TRANSFER STATEMENT IS A FORMAT STATEMENT'
ST-8   'MISSING FORMAT STATEMENT'
ST-9   'SPECIFICATION STATEMENT DOES NOT PRECEDE STATEMENT FUNCTION DEFINITIONS
       OR EXECUTABLE STATEMENTS'
ST-A   'UNREFERENCED STATEMENT FOLLOWS A TRANSFER'
ST-B   'STATEMENT NUMBER MUST END WITH COLON. STATEMENT NUMBER WAS IGNORED'
```

```
'SUBSCRIPTED VARIABLES'
SV-0   'THE WRONG NUMBER OF SUBSCRIPTS WERE SPECIFIED FOR A VARIABLE'
SV-1   'AN ARRAY OR SUBPROGRAM NAME IS USED INCORRECTLY WITHOUT A LIST'
SV-2   'MORE THAN 7 DIMENSIONS ARE NOT ALLOWED'
SV-3   'DIMENSION OR SUBSCRIPT TOO LARGE (MAXIMUM 10**8-1)'
SV-4   'A VARIABLE USED WITH VARIABLE DIMENSIONS IS NOT A SUBPROGRAM PARAMETER'
SV-5   'A VARIABLE DIMENSION IS NOT ONE OF SIMPLE INTEGER VARIABLE,SUBPROGRAM
       PARAMETER,IN COMMON'
SV-6   'PSEUDO VARIABLE DIMENSIONING ASSUMED FOR ARRAY '

'SYNTAX ERRORS'
SX-0   'MISSING OPERATOR'
SX-1   'EXPECTING OPERATOR'
SX-2   'EXPECTING SYMBOL'
SX-3   'EXPECTING SYMBOL OR OPERATOR'
SX-4   'EXPECTING CONSTANT'
SX-5   'EXPECTING SYMBOL OR CONSTANT'
SX-6   'EXPECTING STATEMENT NUMBER'
SX-7   'EXPECTING SIMPLE INTEGER VARIABLE'
SX-8   'EXPECTING SIMPLE INTEGER VARIABLE OR CONSTANT'
SX-9   'ILLEGAL SEQUENCE OF OPERATORS IN EXPRESSION'
SX-A   'EXPECTING END-OF-STATEMENT'
SX-B   'SYNTAX ERROR'

'TYPE STATEMENTS'
TY-0   'THE VARIABLE HAS ALREADY BEEN EXPLICITLY TYPED'
TY-1   'THE LENGTH OF THE EQUIVALENCED VARIABLE MAY NOT BE CHANGED.
       REMEDY: INTERCHANGE TYPE AND EQUIVALENCE STATEMENTS'

'I/O OPERATIONS'
UN-0   'CONTROL CARD ENCOUNTERED ON UNIT 5 AT EXECUTION.
       PROBABLE CAUSE:MISSING DATA OR INCORRECT FORMAT'
UN-1   'END OF FILE ENCOUNTERED (IBM CODE IHC217)'
UN-2   'I/O ERROR (IBM CODE IHC218)'
UN-3   'NO DD STATEMENT WAS SUPPLIED (IBM CODE IHC219)'
UN-4   'REWIND,ENDFILE,BACKSPACE REFERENCES UNIT 5, 6 OR 7'
UN-5   'ATTEMPT TO READ ON UNIT 5 AFTER IT HAS HAD END-OF-FILE'
UN-6   'AN INVALID VARIABLE UNIT NUMBER WAS DETECTED (IBM CODE IHC220)'
UN-7   'PAGE-LIMIT EXCEEDED'
UN-8   'ATTEMPT TO DO DIRECT ACCESS I/O ON A SEQUENTIAL FILE OR VICE VERSA.
       POSSIBLE MISSING DEFINE FILE STATEMENT (IBM CODE IHC231)'
UN-9   'WRITE REFERENCES 5 OR READ REFERENCES 6 OR 7'
UN-A   'DEFINE FILE REFERENCES A UNIT PREVIOUSLY USED FOR SEQUENTIAL I/O (IBM
       CODE IHC235)'
UN-B   'RECORD SIZE FOR UNIT EXCEEDS 32767,OR DIFFERS FROM DD STATEMENT
       SPECIFICATION (IBM CODES IHC233,IHC237)'
UN-C   'FOR DIRECT ACCESS I/O THE RELATIVE RECORD POSITION IS NEGATIVE,ZERO,OR
       TOO LARGE (IBM CODE IHC232)'
UN-D   'ATTEMPT TO READ MORE INFORMATION THAN LOGICAL RECORD CONTAINS (IBM CODE
       IHC213)'
UN-E   'FORMATTED LINE EXCEEDS BUFFER LENGTH (IBM CODE IHC212)'
UN-F   'I/O ERROR - SEARCHING LIBRARY DIRECTORY'
```

```
UN-G   'I/O ERROR - READING LIBRARY'
UN-H   'ATTEMPT TO DEFINE THE OBJECT ERROR FILE AS A DIRECT ACCESS FILE
       (IBM CODE IHC234)'
UN-I   'RECFM IS NOT V(B)S FOR I/O WITHOUT FORMAT CONTROL (IBM CODE IHC214)'
UN-J   'MISSING DD CARD FOR WATLIB.NO LIBRARY ASSUMED'
UN-K   'ATTEMPT TO READ OR WRITE PAST THE END OF CHARACTER VARIABLE BUFFER'
UN-L   'ATTEMPT TO READ ON AN UNCREATED DIRECT ACCESS FILE (IHC236)'
UN-M   'DIRECT ACCESS SPACE EXCEEDED'
UN-N   'UNABLE TO OPEN WATLIB DUE TO I/O ERROR; NO LIBRARY ASSUMED'
UN-P   'ATTEMPT TO WRITE ON A READ ONLY FILE'
UN-Q   'DIRECT ACCESS UNAVAILABLE IN DEBUG MODE'

'UNDEFINED VARIABLES'
UV-0   'VARIABLE IS UNDEFINED'
UV-3   'SUBSCRIPT IS UNDEFINED'
UV-4   'SUBPROGRAM IS UNDEFINED'
UV-5   'ARGUMENT IS UNDEFINED'
UV-6   'UNDECODABLE CHARACTERS IN VARIABLE FORMAT'

'VARIABLE NAMES'
VA-0   'A NAME IS TOO LONG.IT HAS BEEN TRUNCATED TO SIX CHARACTERS'
VA-1   'ATTEMPT TO USE AN ASSIGNED OR INITIALIZED VARIABLE OR DO-PARAMETER IN A
       SPECIFICATION STATEMENT'
VA-2   'ILLEGAL USE OF A SUBROUTINE NAME'
VA-3   'ILLEGAL USE OF A VARIABLE NAME'
VA-4   'ATTEMPT TO USE THE PREVIOUSLY DEFINED NAME AS A FUNCTION OR AN ARRAY'
VA-5   'ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A SUBROUTINE'
VA-6   'ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A SUBPROGRAM'
VA-7   'ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A COMMON BLOCK'
VA-8   'ATTEMPT TO USE A FUNCTION NAME AS A VARIABLE'
VA-9   'ATTEMPT TO USE A PREVIOUSLY DEFINED NAME AS A VARIABLE'
VA-A   'ILLEGAL USE OF A PREVIOUSLY DEFINED NAME'

'EXTERNAL STATEMENT'
XT-0   'A VARIABLE HAS ALREADY APPEARED IN AN EXTERNAL STATEMENT'
```