THE COMPILER FOR TRANSACT.

PERFORMANCE SOFTWARE GROUP
12 Hillview Drive
Baltimore, Md. 21228
(301) 242-6777
Telex: 887764

QUIT ABXCC

CCC = error
A = TYPE
1 - USER
2 - PROGRAMMER
3 - SYSTEM

NOTICE

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

PERFORMANCE SOFTWARE GROUP MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Performance Software Group shall not be responsible for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

PRINTING HISTORY

First Edition	Jan. 1984
Second Edition	Jul. 1984
Third Edition	Feb. 1985
Fourth Edition	Mar. 1986
Fifth Edition	Oct. 1987
Sixth Edition	Nov. 1988
Seventh Edition	May 1991

PREFACE

This manual is a reference for using the FASTRAN compiler to compile and execute programs written in Hewlett-Packard's Transact programming language on the HP-3000 computer system. It assumes a working knowledge of Transact, the HP-3000 and the MPE operating system.

This manual is not intended as a reference for the Transact language. As such, it confines its discussions to the differences between FASTRAN and Transact, and to the special features of FASTRAN. Hewlett-Packard's Transact/3000 Reference Manual (HP Part No. 32247-90001) should be consulted for any general questions regarding the Transact language.

This manual contains the following sections:

Section 1: **INTRODUCTION TO FASTRAN,** describes the major advantages of FASTRAN as well as its limitations.

Section 2: **COMPILING PROGRAMS WITH FASTRAN,** describes in detail how to use the FASTRAN compiler.

Section 3: **FASTRAN COMPILER CONTROL OPTIONS,** describes the effect of each of the FASTRAN compiler options.

Section 4: **PREPARING AND EXECUTING FASTRAN PROGRAMS,** tells how to prepare a FASTRAN program for execution and how to control its execution at run-time.

Section 5: **USING CALL WITH FASTRAN,** describes the implementation of the CALL statement in FASTRAN and discusses the various techniques that can be used with CALL.

Section 6: **USING THE FASTRAN/SEGMENTER,** describes the special segmenter supplied with FASTRAN.

Appendix A explains the error messages issued by the FASTRAN compiler.

Appendix B explains the error messages issued by a FASTRAN program at run time.

Appendix C describes the optional run-time statistics generated by the compiler.

CONTENTS

SECTION 1 INTRODUCTION TO FASTRAN

FASTRAN is a compiler for Hewlett-Packard's Transact programming language used on the HP-3000 series of computers. The main difference between FASTRAN and Transact is that FASTRAN produces HP-3000 object code which can be directly executed by the hardware under control of the MPE operating system. The Transact compiler produces "intermediate processor code" which must be executed interpretively by the Transact processor program.

The major advantage of FASTRAN over Transact is a dramatic reduction in both CPU and elapsed time. A number of features contribute to this high level of performance:

- FASTRAN is compiled, not interpreted. The overhead of interpreting the intermediate processor code is eliminated. In addition, FASTRAN is able to employ special machine instructions (in particular, the COBOL II microcode) that greatly speed certain FASTRAN functions.
- FASTRAN data structures are designed for fast access. Thus FASTRAN is able to eliminate such time-consuming operations as list register searches when data items are referenced, table look-ups when child items are referenced and "garbage collection" in the work register. FASTRAN accomplishes this while still maintaining all of Transact's capabilities for dynamic list and data register allocation and efficient re-use of work register space.
- Where interpretive techniques cannot be avoided, the FASTRAN interpretive procedures have been coded for the maximum level of performance. For example, FASTRAN match register evaluation, while essentially interpretive, is about four times faster than it is with Transact.

Programs compiled with FASTRAN typically require less data space than with Transact, often dramatically less. This is because FASTRAN can use code segments for much of what Transact must store in its data stack. Some of the data that FASTRAN stores in code segments include:

- The program code itself.
- Data item tables (names, aliases, attributes, headings, edit pictures and entry text).
- VPLUS form and field tables.
- Command and sub-command tables.
- Text and control strings.

FASTRAN LIMITATIONS

There are a few features of Transact which are not supported by FASTRAN:

- *Run-time* access to the data dictionary is not supported. Definitions for all data items in your program must be available at compile time, either from the data dictionary or via DEFINE(ITEM) statements in your program. *Compile-time* access to DICTIONARY/3000 is fully supported. System Dictionary access is not supported.
- Test mode is not supported, nor is the TEST built-in command.
- The INITIALIZE built-in command (which allows Transact to initiate a new Transact program without exiting the processor) is not supported.
- A number of limitations apply to the CALL statement:
  - CALLs to Transact programs are supported only if both the called and the calling programs have been compiled with FASTRAN. In addition, the programs must be linked to one another in one of the ways described in Section 5.
  - CALLs cannot be made to programs residing in different groups or accounts (except for process-handling calls, which have certain other limitations – see Section 5).
  - The SWAP option of the CALL statement is not supported and is ignored by FASTRAN. Since FASTRAN uses much less data stack space than Transact, this option is not likely to be needed.

Programs which do not use any of these unsupported features can normally be compiled and executed successfully by FASTRAN with no changes to the Transact source code. Occasionally a program with one or more very large segments may need to be resegmented to be compiled with FASTRAN.

IS FASTRAN A SUBSTITUTE FOR TRANSACT?

FASTRAN is a production-oriented compiler designed for optimum run-time performance. It is not a substitute for Transact – Transact is still the better choice for the development phase of a program's life cycle for several reasons:

- Program compilation is significantly faster with Transact than with FASTRAN, since much of what Transact defers to run time is done by FASTRAN at compile time.
- The program development features of Transact (test mode and run-time data dictionary access) are not supported by FASTRAN.

SYSTEM REQUIREMENTS FOR FASTRAN

The object code generated by FASTRAN includes instructions from the Language Extension instruction set (the COBOL-II microcode). Any machine which is to execute a FASTRAN program must include the COBOL-II firmware (standard on all HP-3000's produced since December 1982). The COBOL-II compiler is not required.

Although the data stacks required by FASTRAN are normally smaller than with Transact, large data stacks may still be required. In addition, FASTRAN may generate large MPE code segments. Therefore, any machine which is to execute compiled FASTRAN programs should be configured for the largest permissible values for both maximum data stack size and maximum code segment size. (See the HP System Manager's manual for more information on these configuration parameters).

SECTION 2 COMPILING PROGRAMS WITH FASTRAN

Compiling a Transact source program with FASTRAN is very similar to using the Transact compiler. You can run the compiler interactively by entering the following command:

```
:RUN  FASTRAN.PUB.FASTRAN
```

The FASTRAN compiler will prompt you as follows:

```
SOURCE FILE>
LIST FILE>
CONTROL>
```

In response to the SOURCE FILE> prompt you should enter the name of your Transact source program.

In response to the LIST FILE> prompt you can respond in any of the following ways:

- Enter a carriage return (or $STDLIST) to direct the listing to your terminal.
- Enter NULL, $NULL or N to suppress the listing.
- Enter LP to direct the listing to the line printer.
- Enter a file name to direct the listing to a new disc file. If the named file already exists, the FASTRAN compiler will ask if you want to purge it.
- Enter a back reference to a file equation (beginning with *).

In response to the CONTROL> prompt you may enter any control options you wish to apply to the compilation. If you respond with a carriage return, the default control options are used. You can reverse the effect of any default control option by preceding it with NO. Some of the FASTRAN control options are different from the Transact options. The FASTRAN control options are discussed in Section 3.

Like Transact, FASTRAN allows you to bypass the compiler prompts by using the PARM= and/or the INFO= options of the :RUN command for FASTRAN.

The PARM= option allows you to identify your source file and/or your list file with file equations, as follows:

PARM=1 FASTRAN uses formal-file-designator FSTTEXT for your source file and the SOURCE FILE> prompt is suppressed.

PARM=2 FASTRAN uses formal-file-designator FSTLIST for your list file and the LIST FILE> prompt is suppressed.

PARM=3 Combines the effect of PARM=1 and PARM=2.

FASTRAN also allows you to control the destination file for your compiled object code with the PARM= option. Normally the object code is written to a USL (user subroutine library) file named $OLDPASS. If you want to direct the object code to a different USL file, you can use PARM=4.

FASTRAN will then write the object code to formal-file-designator SPLUSL. You can use a file equation to equate SPLUSL to your USL file, for example:

```
:FILE  SPLUSL=MYUSL
:RUN  FASTRAN.PUB.FASTRAN;PARM=4
```

Although FASTRAN no longer uses the SPL compiler, the USL formal-file-designator is still SPLUSL to maintain compatibility with customers' existing job streams. You can combine the effect of PARM=4 with PARM=1, 2 or 3 by using PARM=5, 6 or 7, respectively.

The INFO= option allows you to supply control options directly to the FASTRAN compiler, bypassing the CONTROL> prompt. The control options are separated by commas, just as they would be in response to the CONTROL> prompt. If you want FASTRAN to use the default options and to bypass the CONTROL> prompt, set the INFO= parameter to INFO=" ".

FASTRAN can access a data dictionary during compilation. Like Transact, FASTRAN uses DICT.PUB as the formal designator for the data dictionary. If you want FASTRAN to use a different dictionary, a file equation is needed.

The FASTRAN compiler uses three additional files which are normally of no concern since the default assignments are usually appropriate.

- **FSTOUT** is the formal-file-designator for prompts and error messages from the compiler. The default assignment for FSTOUT is $STDLIST.
- **FSTIN** is the formal-file-designator for responses to the prompts issued by the compiler. The default assignment for FSTIN is $STDINX.
- **FSTRN000.PUB.FASTRAN** is the formal-file-designator for the FASTRAN compile-time message catalog. In a normal FASTRAN installation no file equation will be needed. (Note that the FASTRAN compile-time and run-time message catalogs are in different files.)

USING UDC'S FOR COMPILING WITH FASTRAN

The FASTRAN account contains a UDC file named UDC.PUB.FASTRAN which contains user-defined commands for compiling with FASTRAN. Each of these commands is described below:

:FASTRAN The :FASTRAN command simply executes the following MPE command:

```
:RUN  FASTRAN.PUB.FASTRAN
```

:FASTCOMP *source-file [,[usl-file] [,[list-file] [,control-option]...]]*

The :FASTCOMP command runs the FASTRAN compiler and allows you to designate your *source-file, usl-file, list-file* and *control-options* in the same line.

Only the *source-file* parameter is required. The default *usl-file* is $OLDPASS and the default *list-file* is $STDLIST. If you want to suppress the listing, you must use $NULL and not simply NULL.

Up to five options may be entered. The options must begin with the fourth parameter. You may need to use extra commas to indicate missing parameters, for example:

```
:FASTCOMP  MYPROG,,,DEFN,OPTS
```

This command will compile MYPROG with options DEFN and OPTS. The extra commas indicate that defaults are to be used for the *usl-file* and the *list-file*.

:FASTPREP *source-file [,[program-file] [,[list-file] [,control-option]...]]*

The :FASTPREP command runs the FASTRAN compiler and then prepares the compiler output, producing an executable program file. The program file is PREPed with MAXDATA=32000 using the RL (relocatable library) file RL.PUB.FASTRAN.

Only the *source-file* parameter is required. The default *program-file* is $OLDPASS and the default *list-file* is $STDLIST.

Note that both :FASTPREP and :FASTGO (below) use the FASTRAN/SEGMENTER, rather than the MPE segmenter, to prepare the program file.

:FASTGO *source-file [,[list-file] [,control-option]...]]*

The :FASTGO command runs the FASTRAN compiler, uses the FASTRAN/SEGMENTER to prepare the program file, and then executes it.

Only the *source-file* parameter is required. The default *list-file* is $STDLIST. The program file is always $OLDPASS.

SECTION 3 FASTRAN COMPILER CONTROL OPTIONS

Like Transact, FASTRAN allows you to control certain features of compilation by supplying control options, either in response to the interactive CONTROL> prompt or via the INFO= parameter of the compiler :RUN command.

Many of the FASTRAN control options are the same as the Transact options. There are a few Transact options which are not relevant to FASTRAN and are therefore not supported. There are also a number of additional options which are unique to FASTRAN.

There are six Transact options which are not supported by FASTRAN. They are:

OBJT This option tells Transact to produce a listing of the intermediate processor code. FASTRAN does not produce intermediate processor code.

OPTE This option tells Transact not to store the edit text for a data item in the data stack tables. FASTRAN never stores this information in the data stack.

OPTH This option tells Transact not to store the heading text for a data item in the data stack tables. FASTRAN never stores this information in the data stack.

OPTP This option tells Transact not to store the prompt text for a data item in the data stack tables. FASTRAN never stores this information in the data stack.

OPT@ This option tells Transact not to store the edit text, heading text, textual name and prompt text for a data item in the data stack tables. FASTRAN never stores this information in the data stack.

XERR This option tells Transact to create a code file even if there are errors in the compilation. FASTRAN does not allow you to create object code if there are compilation errors.

The remaining nine Transact options are supported by FASTRAN. These options all have essentially the same effect in FASTRAN as in Transact. The default options are marked by an asterisk (*). The FASTRAN defaults are the same as the Transact defaults, except where noted:

***CODE** Creates a USL file containing the compiled object code, unless any errors occurred during the compilation.

DEFN Produces an alphabetized listing of all data items referenced in the program, including the definition of each.

***DICT** Tells the compiler to use the data dictionary (DICT.PUB) to resolve data item definitions.

***ERRS** Lists compilation errors on $STDLIST even if the listing is suppressed or directed elsewhere.

***LIST** Generates a listing of the compiled source code. With FASTRAN this listing is produced during the compiler's second pass. Therefore, if any errors are detected during the first pass, no listing will be produced.

OPTI Optimizes the storage of data item names in FASTRAN's internal tables. When this option is used, any data items which were defined in the program with the OPT option will not have their names stored in the compiler-generated data item tables. As with Transact, OPT should not be used with any data item whose name is needed for a prompt string, a display heading, a LIST= option for IMAGE or a WINDOW= option for VPLUS.

Note that FASTRAN does not store the data item tables in the data stack, but rather in code segments. Therefore, this option will have no effect on the size of the FASTRAN data stack.

Normally the OPTI option will not be required with FASTRAN, even if the program you are compiling requires it with Transact. This is because there is much more space available for these tables with FASTRAN. We recommend that you use OPTI only in the event of compile-time errors 16 or 22, which indicate an overflow of these tables.

***OPTS** Optimizes segment transfers in a segmented program. When you use this option the list, match and update registers are not checked for local segment items when a segment transfer occurs. Use of this option speeds segment transfers considerably.

With Transact, the OPTS option is normally off. This is because these checks are important when you are developing and debugging a Transact program. However, since FASTRAN is intended as a high-performance production compiler, OPTS is a default option for FASTRAN.

STAT Generates statistics on run-time storage allocation for the compiled program. However, due to the great differences in the run-time environments of Transact and FASTRAN, the statistics are presented in a completely different format. Appendix C describes the format of a FASTRAN STAT listing.

XREF Generates a cross-reference listing of label definitions and their references.

In addition, FASTRAN provides nine new control options. These are described below. As before, an asterisk (*) denotes a default option:

***CHEK** Tells the FASTRAN compiler to generate code for certain run-time checks, namely verifying that a referenced data item is in the list register and verifying that there is a pending PERFORM when a RETURN statement is executed.

Since the performance penalty for performing these checks is relatively small (no more than a few percent), CHEK is a default option for FASTRAN. However,

you can significantly reduce the amount of code generated (usually about 15%) by specifying NOCHEK. This can be a handy alternative to segmentation if you have a FASTRAN segment which is just a little too large.

Be aware that your program will produce unpredictable results if you compile it with NOCHEK and it happens to reference a data item which is not in the list register, or if it attempts to RETURN when there is no pending PERFORM.

CLST Lists the generated code (in assembly language) immediately following each program statement.

DCAL Generates dynamic calls for all CALL statements in your program. This will allow your program to be executed even if some of the programs which it calls are not available at load-time. Dynamic (and static) calls and the use of the DCAL option are discussed in detail in Section 5.

DDBO Defers data base opens until the data base is first referenced. Normally, all data bases are opened at the beginning of your program. When DDBO is specified, your program begins executing immediately and each data base is opened when it is first referenced in a data management statement.

FLST Forces the compiler to generate a listing for all compiled source code, even if it contains !NOLIST statements. This can be useful if there are sections of code that you normally want to suppress on the listing, but occasionally you want to list them.

***OPTX** Optimizes expression evaluation based on the declared size of the data items in an expression. When you specify this option, FASTRAN assumes that no data item will ever contain a value larger than its declared size. When FASTRAN generates code for an intermediate calculation, it uses these assumed maximum values to determine the maximum range of the intermediate result. It then chooses the data type and size to accommodate the maximum range.

If you specify NOOPTX, FASTRAN assumes that a data item can contain any value within the range of the underlying data type regardless of the declared size. This can produce much less efficient calculations.

For example, consider the following Transact statements:

```
DEFINE(ITEM)  ITEM1  I(4):  ITEM2  I(4):  ITEM3  I(4):
    RESULT  I(4);
LET  (RESULT)  =  [(ITEM1)+(ITEM2)]+(ITEM3);
```

With OPTX in effect, FASTRAN will assume that all three operands on the right are in the range -9999 to 9999. When it generates code for the intermediate calculation (ITEM1)+(ITEM2) FASTRAN can use single integer arithmetic since the result must be in the range -19998 to 19998. The final addition can also be performed in single integer arithmetic since the range of the result is -29997 to 29997.

With NOOPTX the situation is much different. FASTRAN must be prepared for operand values in the range -32768 to 32767. Since the range of the intermediate result is now -65536 to 65534, FASTRAN must generate code to convert both operands to double integers, add them with double integer arithmetic, convert the third operand to a double integer, add the third operand to the intermediate result and finally convert the result to a single integer. This is obviously a much more time-consuming calculation. The performance difference is even more dramatic when the magnitudes of the operands force the calculations from double integers into packed decimal.

Because of the performance objectives of FASTRAN, OPTX is a default option. Experience has shown that the vast majority of Transact programs can be successfully executed using the assumptions that OPTX makes. However, if a FASTRAN program should terminate with an integer overflow (program error 51) or a decimal overflow (program error 47), you should recompile the program with NOOPTX to see if the problem disappears.

If a program experiences an overflow with OPTX but executes successfully with NOOPTX, you can simply continue to use NOOPTX whenever you recompile the program. However, a preferable solution would be to locate the data item which is causing the overflow and change its declaration to reflect its true range of values. The program location of the statement in which the overflow occurred will be indicated in the overflow error message.

SSEG Split segment option. Causes the compiler to generate all code for data management statements (FIND, REPLACE, etc.) and for VPLUS statements in a separate procedure from the main line code. This option permits FASTRAN to compile larger program segments. It can be used in combination with the NOCHEK option if you encounter compile-time error 202: *Too much code in this segment*.

SUBP This option is used when you are compiling a sub-program which will be called by another FASTRAN program. No outer block is generated when the SUBP option is specified. The processing which is normally done by the outer block is done by the calling program. The SUBP option is discussed in more detail in Section 5.

***USLI** Causes the USL file to be initialized (cleared) before compilation begins. You would normally use NOUSLI only when compiling a called program to the same USL file as a previously-compiled calling program. The use of NOUSLI is discussed in more detail in Section 5.

FASTRAN COMPILER DIRECTIVES

FASTRAN supports all of the Transact compiler directives (!COPYRIGHT, !INCLUDE, !LIST, !NOLIST, !PAGE and !SEGMENT), treating them the same as Transact. In addition, FASTRAN supports four additional compiler directives:

!CALLTABLE= Establishes an internal table to keep track of dynamic calls, so that dynamic calls by one program to the same called program will only incur the overhead of the LOADPROC intrinsic for the first call.

To request that FASTRAN establish a table to save the LOADPROC information for called programs and to re-use that information the next time it is called, use a directive like:

```
<<!CALLTABLE=20>>
```

This sets up a table of 20 entries. If more than 20 different dynamic calls are issued, only the first 20 are retained. The maximum size of this table is 100 entries.

!DCAL= Dynamic calls can be resolved out of the SL's in the group and account where the program file resides, or out of the logon group and account.

To use the SL's in the account and group where the program file resides, include the following directive in your source code prior to the dynamic call:

```
<<!DCAL=PROGRAM>>
```

To use the SL's in the logon group and account, use the following directive:

```
<<!DCAL=LOGON>>
```

The !DCAL compiler directive takes effect at the point it appears in the source code and remains in effect until another !DCAL appears. Thus, you can use both sets of libraries in the same program.

The default is !DCAL=LOGON. The !DCAL directive has no effect on static or PH calls.

!PH Used in conjunction with the CALL verb to define a process-handling call. Process-handling calls are described in Section 5.

!SORTSTACK= Used to override FASTRAN's default allocation of stack space for sorting. Whenever a SORT= option appears in a FIND or OUTPUT statement, FASTRAN must allocate space on the data stack for SORT/3000 to use. FASTRAN's default allocation is 6000 words. This normally provides a good balance between sort speed and conserving stack space. There are two situations where you may wish to override the default allocation with the !SORTSTACK= compiler directive:

- If your program uses nested sorts (a statement with a SORT= option which PERFORM's a paragraph containing another SORT= option) you may need to reduce the stack space for each sort so as not to run out of stack. This is normally necessary only if your sorts are nested three or more deep.
- If your program has extra stack space available and you are sorting a large number of records, you can improve the performance of your program by increasing the !SORTSTACK= beyond the default of 6000 words.

The !SORTSTACK=> compiler directive takes effect at the point it appears in the source code and remains in effect unless another !SORTSTACK= appears. Thus you can specify different !SORTSTACK= values for different sorts. In the case of nested sorts, this allows you to set aside more stack for sorts with a large number of records, and less stack for sorts with only a few records. For example:

```
.
.
<<!SORTSTACK=8000>>
FIND(SERIAL)   BIG-FILE,...,SORT=...,
    PERFORM=SORT-2;
.
.
SORT-2:
<<!SORTSTACK=2500>>
FIND(CHAIN)   LITTLE-FILE,...,SORT=...,
    PERFORM=...;
.
.
```

Note that if you set !SORTSTACK= too low (below about 2500 words) or too high, you will get a run-time failure of the sort.

The four special FASTRAN compiler directives (!CALLTABLE=, !DCAL=, !PH and !SORTSTACK=) are always enclosed as comments (between << and >>, with no space between << and !). This is to provide backward compatibility with the Transact compiler.

SECTION 4 PREPARING AND EXECUTING FASTRAN PROGRAMS

This section discusses how to prepare and execute Transact programs which have been compiled with FASTRAN. Programs which use the CALL verb require special treatment and are discussed separately in Section 5. The material in this section pertains to programs which do not use CALL.

PREPARING A FASTRAN PROGRAM FOR EXECUTION

Because the Transact compiler produces a special intermediate processor code, a Transact code file needs no further preparation in order to be interpreted by the Transact processor.

FASTRAN, like most other compilers on the HP-3000, produces executable object code in the form of a USL (user subprogram library) file. A USL file must be prepared before it can be executed. The result of preparing a USL file is a program file which can then be executed by the MPE :RUN command.

The standard method of preparing a program file on the HP-3000 is to use the MPE :PREP command. However, a limitation of the :PREP command is that all RL (relocatable library) code must fit into a single MPE code segment. Whichever of FASTRAN's run-time library procedures your program requires are included from an RL file (RL.PUB.FASTRAN), and these frequently will require more than one MPE code segment to contain them. In such a case the :PREP command will fail with the message: *ERROR #40, RL SEGMENT, CODE SEGMENT OVERFLOW.* Therefore, you should avoid using :PREP with FASTRAN programs.

The FASTRAN/SEGMENTER overcomes this limitation and should always be used for preparing FASTRAN programs (note: other functions of the FASTRAN/SEGMENTER are described in Section 6). The easiest way to compile and prepare a stand-alone program is to use the :FASTPREP or :FASTGO commands (described in Section 2), which use the FASTRAN/SEGMENTER. A sample job stream for compiling the source file MYSOURCE and producing a program file MYPROG follows:

```
:JOB <<log-on information>>
:PURGE MYPROG
:FASTPREP MYSOURCE,MYPROG
:SAVE MYPROG
```

The :PURGE command gets rid of any existing copy of MYPROG. The :SAVE command is necessary because the program file is initially created as a temporary file.

SPECIAL CAPABILITIES

With Transact, special capabilities are not normally of concern. This is because when you execute a Transact program, you are actually running the program file TRANSACT.PUB.SYS. This program file comes with all special capabilities and, since it resides in PUB.SYS, automatically confers these capabilities on any Transact user. (Note that Transact actually defeats MPE security as far as special capabilities are concerned.)

Normally, no special capabilities are required to execute a FASTRAN program. However, your program may contain PROC statements that call intrinsics or user-written procedures which require special capabilities. Or your program may require multiple-RIN (MR) capability because it locks two or more data bases or files simultaneously. In such cases, the program file must be prepared with special capabilities in order to execute. In addition, the group and account in which your prepared FASTRAN program is to reside must also have any required special capabilities.

The :FASTPREP command does not allow you to specify special capabilities directly in the command itself. However, after the :FASTPREP command you can use the ALTCAP program which is supplied on the FASTRAN distribution tape (ALTCAP.PUB.FASTRAN). This program allows you to alter the capabilities of an existing program file.

ALTCAP uses the INFO= parameter to designate the program file to be altered and the new list of capabilities. A semicolon should separate the program file name from the capability list, and commas should be used to separate the individual capabilities. For example, if your program uses a PROC statement to call the CREATEPROCESS intrinsic (which requires process-handling capability), your job stream should include the following statement:

```
:RUN  ALTCAP.PUB.FASTRAN;INFO="MYPROG;IA,BA,PH"
```

This gives PH (process-handling) capability (as well as interactive and batch access) to MYPROG. The complete job stream follows:

```
:JOB <<log-on information>>
:PURGE MYPROG
:FASTPREP MYSOURCE,MYPROG
:SAVE MYPROG
:RUN  ALTCAP.PUB.FASTRAN;INFO="MYPROG;IA,BA,PH"
:EOJ
```

You can also run ALTCAP interactively by omitting the INFO= parameter. ALTCAP will prompt for the program file name and capabilities.

No special capabilities are required to run ALTCAP. However, in order to execute the altered program file, the account and group in which the program file resides must have all required capabilities.

EXECUTING FASTRAN PROGRAMS

FASTRAN programs are executed with the MPE :RUN command. For example, if the name of your program file is MYPROG, you would simply enter:

```
:RUN MYPROG
```

Two additional parameters *may* be required with the :RUN command:

- If you want to supply a default mode to be used in opening your program's data bases, use the PARM= parameter. For example:

  ```
  :RUN MYPROG;PARM=5
  ```

  This command causes any data bases to be opened in mode 5, unless a different mode was specified in the SYSTEM statement of your program.

  Using the PARM= parameter with a FASTRAN program is equivalent to entering a mode as the second parameter in response to the SYSTEM NAME>prompt from the Transact processor.

- If you are using PROC statements to call procedures which reside in group or account SL's (segmented libraries), you must use the LIB= parameter on your :RUN command. For example, if your program uses PROC to call procedures in a group SL, you would enter

  ```
  :RUN MYPROG;LIB=G
  ```

  There is a difference between Transact and FASTRAN concerning where the libraries must be located. Transact uses the SL in your *log-on* group as its group SL and it uses the SL in the PUB group of your *log-on* account as its account SL. FASTRAN uses the SL in the group *where your program file resides* as its group SL and uses the SL in the PUB group of the account *where your program file resides* as its account SL. This makes a difference only if you are running a program with PROC calls which resides in a different account or group than you are signed on to.

  The reason for this difference is that Transact uses the LOADPROC intrinsic at run-time to locate procedures called via PROC, whereas FASTRAN uses procedure call (PCAL) instructions which are resolved by the system loader at load-time. LOADPROC uses the *log-on* libraries to locate external references and the loader uses the libraries which *accompany the program file.*

THE FASTRAN RUN-TIME MESSAGE CATALOG

Whenever you execute a program compiled with FASTRAN, the FASTRAN run-time message catalog (CATALOG.PUB.FASTRAN) should be present. If you transport a compiled FASTRAN program to a machine without a FASTRAN compiler, you should make sure this file is available. (Your FASTRAN license permits you to copy and transport the message catalog to other machines.) If you want to rename the catalog, you must either set a file equation for CATALOG.PUB.FASTRAN at run-time, or have your program issue the file equation when it initiates.

FASTRAN RUN-TIME FILE EQUATIONS

Most of the formal file designators that FASTRAN uses at run time are the same as those used by Transact. This permits programs which issue file equations programmatically to execute under FASTRAN with no modifications. Each of the formal file designators is described below:

TRANIN is the formal file designator for responses to prompts issued by your program. The default assignment is $STDINX.

There is one difference between Transact and FASTRAN regarding the input file TRANIN. The Transact processor issues a SYSTEM NAME> prompt and reads the system name from TRANIN. Since FASTRAN programs are executed directly, there is no SYSTEM NAME> prompt. Therefore, if a program is set up to read TRANIN from a disc file (or from $STDIN in a job stream), the record containing the system name must be removed from the disc file or job stream before running with FASTRAN.

TRANOUT is the formal file designator for output from your program that is normally sent to your terminal. The default assignment is $STDLIST.

TRANLIST is the formal file designator for output from your program that is normally sent to the line printer. The default assignment is DEV=LP.

TRANVPLS is the formal file designator used by VPLUS to open the terminal. The default is $STDIN.

TRANSORT is used by Transact as a temporary file during sort operations. FASTRAN uses input and output procedures and does not require a work file. However, FASTRAN will allow you to use a TRANSORT file equation with a DISC= parameter in order to specify the maximum number of sort records at run-time.

TRANDUMP is not used by FASTRAN. Transact uses TRANDUMP for test mode output. Test mode is not supported by FASTRAN.

BUILT-IN PROCESSOR COMMANDS

FASTRAN supports all Transact built-in commands and command qualifiers with the following two exceptions:

- The INITIALIZE built-in command is not supported. This command tells the Transact processor to begin processing a new Transact program. Since FASTRAN programs are executed directly by MPE, the equivalent function would be provided by an EXIT command followed by an MPE :RUN command for the new FASTRAN program.
- The TEST built-in command is not supported since FASTRAN does not support test mode.

SECTION 5 USING CALL WITH FASTRAN

The Transact CALL statement is used to initiate execution of another Transact program from within an executing Transact program. When the Transact processor interprets a CALL statement, it opens and reads the code file for the called program and begins interpreting the IP (intermediate processor) code in the new code file.

HOW FASTRAN HANDLES CALLS

Since FASTRAN programs execute directly under MPE, a different method of initiating called programs must be used. One approach that the FASTRAN compiler could take would be to use the CREATEPROCESS intrinsic (or CREATE and ACTIVATE). However, this approach has several drawbacks:

- Since the called program would be executing as a separate MPE process, it would not be possible to share data files, form files and data bases in the same way that Transact permits.
- There is considerable overhead associated with the CREATEPROCESS intrinsic.

An alternative approach would be for FASTRAN to implement the CALL verb simply as a procedure call. This permits the required data and file sharing and meets the performance requirements of FASTRAN. However, this approach also has several drawbacks:

- Because of the linkage requirements for procedure calls, special attention must be given when you compile, prepare and execute programs which use CALL. Called programs must either be prepared into the same program file as the calling program, or they must be placed in a segmented library (SL) which is available to the calling program at run time. If the called programs are in an SL and the calling program is run from different accounts and/or groups, multiple copies of the SL may be required.
- Since the CALL verb references a procedure rather than a file, there is no way to qualify the CALL with an account and/or group name.
- Because the maximum number of MPE code segments is limited to 255 for a program file and to 254 for a segmented library, there is a limit to the total number of programs that can be linked together using procedure calls.

Since the lack of file sharing using process handling would introduce a major incompatibility between Transact and FASTRAN, the procedure call approach is the one normally used by FASTRAN to implement the CALL verb.

However, occasionally there are situations where the process-handling approach is appropriate. At the end of this section there is a description of these situations and the special cautions pertaining to the use of *process-handling calls*.

The remainder of this section discusses normal (non-process-handling) calls, the different methods of linking calling and called programs together, and how to choose among these methods.

STATIC AND DYNAMIC CALLS

FASTRAN can generate either of two different types of code for non-process-handling calls, referred to as *static calls* and *dynamic calls*. *Static calls* are direct procedure calls to the called program. The major characteristics of static calls are:

- The name of the called program must be available to FASTRAN at compile time.
- The object code for all called programs must either be included in the USL file at prep time or must be in an SL at load time (even if the CALLs will not actually be executed at run-time).

FASTRAN generates a static call whenever a literal program name is used in the CALL statement and the DCAL (dynamic call) option is off. *Dynamic calls* use the MPE LOADPROC intrinsic to load the called program at run time. The major characteristics of dynamic calls are:

- The name of the called program is not required until run time.
- The object code for any programs which are called must be in an SL. However, only those programs which are actually called at run time need to be present in the SL.

FASTRAN generates dynamic calls whenever a variable program name is used in the CALL statement, or if the DCAL option is on.

Table 5-1 shows the type of call which FASTRAN will generate under each set of circumstances.

DCAL option	Program Name	
	Literal: CALL PROG;	Literal: CALL (PROG);
OFF	Static	Dynamic
ON	Dynamic	Dynamic

Table 5-1. Types of CALLs generated.

There are advantages and disadvantages to each type of call. The primary advantage of static calls over dynamic calls is superior run-time performance. *Dynamic calls* must use LOADPROC whenever a CALL statement is executed. Although LOADPROC uses very little CPU time, it does require several seconds of elapsed time while it searches the various libraries and loads the requested program. With *static calls* the called programs are loaded when the main program is loaded and the run-time overhead is negligible for most applications.

However, there are several advantages of *dynamic calls* which can make them preferable in certain situations:

- Dynamic calls do not require the called program name to be known at compile time. Therefore, CALL statements which use a variable for the called program name are always compiled as dynamic calls.
- Dynamic calls do not require the object code for the called programs to be available until the CALL statement is actually executed. Therefore, dynamic calls allow a main program to be executed even if some of the called programs it references have not yet been compiled (or even written). Of course, an error will occur if your program actually tries to call a missing program.
- In general, each program called using a static call requires an entry in the calling program's segment transfer table and in the operating system's code segment table (CST). Both of these tables have a maximum size, so there is a theoretical limit to the number of different called programs that a main program can reference via static calls. This number will vary depending on the number of other external references in your program, the configured size of the code segment table and the CST requirements of the other programs executing at the time.

CONTROLLING THE TYPE OF CALLS GENERATED BY FASTRAN

A program can contain both *static* and *dynamic* calls. If you do not select the DCAL option at compile time, FASTRAN will generate *static* calls for all CALL statements which use a *literal* program name and will generate dynamic calls for all CALL statements which use a *variable* program name. If you do specify the DCAL option, *dynamic* calls are generated in all cases.

If you want to control the type of call which FASTRAN will generate, you can use a *variable* program name for calls you want to be *dynamic*, and a *literal* program name for calls you want to be *static*, and then compile the program with DCAL off. If you have a system which uses a large number of different called programs, you may wish to use static calls (which are faster) for the most frequently called programs and dynamic calls for the remainder.

LIMITATIONS ON THE CALL STATEMENT WITH FASTRAN

A few limitations apply to the CALL statement when you are using FASTRAN:

- Calls can only be made to Transact programs that have been compiled with FASTRAN, and the called program must be linked to the calling program in one of the ways described below.
- Calls cannot be made to programs residing in a different group or account. If a called program name is qualified with an account and/or a group name, the qualification is ignored by FASTRAN (except for process-handling calls, described at the end of this section).
- The SWAP option is not supported and is ignored if it appears on a CALL statement. Since FASTRAN uses far less stack space than Transact, this option is unlikely to be needed.

COMPILING, PREPARING AND EXECUTING WITH CALL

The following cases will demonstrate compilation, preparation and execution of FASTRAN programs that use CALL:

CASE 1: STATIC CALLS WITH PREP-TIME LINKAGE

In this case we have a main program MAIN which calls three other programs PROG1, PROG2 and PROG3 using static calls. All four programs will be compiled into a single USL file and then prepared into a program file which will contain the object code for all four programs.

This is the method you should use whenever possible because of two important advantages:

- Since all program linkage is performed at prep time, systems compiled in this manner will have the best load-time and run-time performance.
- Since all called programs are included in the program file, the program can be run stand-alone, that is, without any group or account SL's to worry about.

The following job stream will compile and prepare the example system of four programs:

```
:JOB <log-on information>
:FASTCOMP  MAIN
:FASTCOMP  PROG1,,,NOUSLI,SUBP
:FASTCOMP  PROG2,,,NOUSLI,SUBP
:FASTPREP  PROG3,,,NOUSLI,SUBP
:SAVE  $OLDPASS,MYPROG
:EOJ
```

The result will be a program file named MYPROG.

When you are compiling a system of programs using this method, keep the following points in mind:

- The main program should be compiled first, using the :FASTCOMP command, with the USLI and NOSUBP options (both defaults) in effect.
- The called programs should then be compiled. The order of the called programs is not important, nor is it important whether they are called directly by the main program or if they call each other. Each called program should be compiled with NOUSLI (so that the previously-compiled object code is not cleared from the USL) and with SUBP (to suppress generation of an outer block). Use the :FASTCOMP command for all but the last called program (note: FASTRAN versions A.03.F00 and later allow you to omit the SUBP option).
- The last called program should be compiled with the :FASTPREP command so that the USL file is prepared into a program file.

To run the program file you need only enter:

```
:RUN MYPROG
```

You can also invoke the FASTRAN/SEGMENTER directly to handle this type of call structure. See the example in Section 6.

CASE 2: SETTING UP A SEGMENTED LIBRARY (SL) FOR FASTRAN

Any programs which are called with *dynamic* calls *must* be located in an SL (segmented library). Programs which are called with *static* calls can also be placed in an SL if you choose to use load-time linkage. An SL is not required if you are using only static calls with prep-time linkage, as in case 1 (or if you are not using CALL at all). This case will demonstrate how to create and initialize an SL for use with FASTRAN.

Two steps are required to set up an SL for FASTRAN:

- You must create an SL file (unless you intend to use an existing SL).
- You must add the FASTRAN run-time library procedures to the new SL. Both steps can be performed using the FASTRAN/SEGMENTER by entering the following commands:

```
:RUN FASTSEG.PUB.FASTRAN {or :FASTSEG}
=BUILDSL SL,10000,20
=UPDATESL SL
=EXIT
```

The :RUN command invokes the FASTRAN/SEGMENTER.

The =BUILDSL command creates a new SL file (named SL) with a total size of 10,000 sectors allocated in 20 extents. Only 1 extent (500 sectors) will be initially allocated. You may want to use a different space allocation. If you are using an existing SL, you should omit this command.

The =UPDATESL command adds the FASTRAN run-time library segments to the new SL. The library segments only need to be added to an SL the first time you use it with FASTRAN.

Whenever you install a new release of FASTRAN, you must replace these segments in your FASTRAN SL's. The same =UPDATESL command can be used to replace the old run-time library segments with the new version.

The group and account location of your SL files is important, particularly if you will be executing FASTRAN programs from groups or accounts other than where the program file resides.

- For any called programs that are accessed by *dynamic* calls, the SL may be in either the *log-on* account (in the PUB group or the log-on group), or it may be in the same account as the program file (in the PUB group or the same group as the program file). See the !DCAL= compiler directive in Section 3 for a discussion of how to control which set of SL's is used.
- If any called programs will be accessed via *static calls using load-time linkage*, the SL must be in the same group as the program file (or in the PUB group of the same account as the program file).

Note that a single FASTRAN program could access up to five different SL's:

- The SL in your log-on group (used for dynamic calls when the !DCAL=LOGON compiler directive is in effect)
- The SL in the PUB group of your log-on account (used for dynamic calls when the !DCAL=LOGON compiler directive is in effect, if they were not resolved from the SL in your log-on group)
- The SL in the group where your program file resides (used for dynamic calls when the !DCAL=PROGRAM compiler directive is in effect, for static calls, and for PROC statements)
- The SL in the PUB group of the account where your program file resides (used for dynamic calls when the !DCAL=PROGRAM compiler directive is in effect, for static calls, and for PROC statements, if they were not resolved above)
- SL.PUB.SYS (used for any calls or PROC statements not resolved elsewhere)

CASE 3: ADDING A FASTRAN PROGRAM TO AN SL

In this case we will compile a FASTRAN program and add it to an SL using the FASTRAN/SEGMENTER. The source file for the program is MYSOURCE and the name of the program (in the SYSTEM statement of the source program) is PROG. The following commands will accomplish this:

```
:FASTCOMP  MYSOURCE
:FASTSEG
=REPLACE  PROG,SL,$OLDPASS
=EXIT
```

The :FASTCOMP command compiles the program. Since no *usl-file* is specified, the compiled code will be in $OLDPASS. The SUBP option could have been used, but is not required when using the FASTRAN/SEGMENTER.

The :FASTSEG command invokes the FASTRAN/SEGMENTER.

The =REPLACE command adds the program to the SL. If a previous version of the program already exists in the SL, it is replaced. The three parameters are the *system name* of the program to be added or replaced (PROG), the *name of the segmented library* (SL) and the *name of the usl-file* containing the compiled code ($OLDPASS).

CASE 4: STATIC CALLS WITH LOAD-TIME LINKAGE

Case 1 described how to use static calls with prep-time linkage and the advantages of that method. This case will show how to use load-time linkage with the same set of four programs, a main program and three called programs.

The primary advantage of using this method is that the individual programs in a system can be recompiled separately and (except for the main calling program) no PREP step is required, while still maintaining the high performance provided by static calls. The disadvantage is that the called programs must be placed in an SL, and the SL must be available at run time.

To set up this system of programs, you would first compile the three called programs and add them to your SL as described in Cases 2 and 3. You would then compile and prepare the main program as if it were a stand-alone FASTRAN program:

```
:FASTPREP  MAIN
:SAVE  $OLDPASS,PROG
```

To run the MAIN program, the SL file containing the three called programs must be available in the same group as the program file (or in the PUB group of the same account). You would then execute the program as follows:

```
:RUN  PROG;LIB=G
```

The LIB=G parameter tells the MPE loader to use the group SL (and, if required, the account SL) to resolve external references at load time. This method will require slightly more load time than Case 1 but will provide the same run-time performance.

CASE 5: DYNAMIC CALLS

No special techniques or parameters are required to compile, prepare or execute the *main program* in a system which uses only dynamic calls. However, any programs which are *called* must be available in an SL at run time. See Case 2 for a discussion of where the SLs must be located.

Table 5-2 summarizes the features of each of the three different methods for handling CALL statements with FASTRAN.

Type of CALL and linkage	Static with prep-time linkage	Static with load-time linkage	Dynamic (run-time linkage)
Run-time performance	Excellent	Excellent	Several seconds for LOADPROC on each call
Load-time performance	Excellent	Slightly longer load time	Excellent
Location of SL	Not required	Same group and account as program file	Log-on group/acct or same group/acct as program file
Lib=G (or P) required on :RUN	No	Yes	No
All called programs required to run main program?	No	Yes	No

Table 5-2. Features of Different Types of FASTRAN CALLs.

PROCESS-HANDLING CALLS

The discussion so far in this section has focused on the the way FASTRAN normally handles the CALL verb – via a procedure call. In most situations this method provides the greatest compatibility with the Transact implementation of the CALL verb.

The rest of this section discusses the *process-handling call* (PH-call) which uses MPE's process-handling capability to implement the CALL verb. The main disadvantage of using this form of CALL is that data bases, data files and form files cannot be shared between programs in the same way that Transact permits. **YOU SHOULD CAREFULLY CONSIDER THIS INCOMPATIBILITY IF YOU USE PH-CALLS, AND YOUR PROGRAM WILL PROBABLY REQUIRE SOME MODIFICATIONS.** (Note however that sharing of the data register between called and calling programs is supported and data can be passed between programs in this way.)

There are several advantages to PH-calls, however, and these may outweigh the disadvantages for some applications:

- No special prep-time or run-time linkage need to be used and there are no SL's to be concerned with.
- There is no limit to the number of programs in a system of called programs.
- The same copy of a program can be used for stand-alone execution and for called execution.
- Programs in different groups or accounts can be called.

You tell FASTRAN to generate a process-handling call by inserting a *pseudo-comment* containing the !PH compiler directive into the CALL statement. For example:

```
CALL PROG1, DATA=ITEM1 <<!PH>>;
```

The pseudo-comment <<!PH>> may appear anywhere in the CALL statement after the program name and before the semicolon. Any CALL statement containing this pseudo-comment will be compiled as a PH-call, regardless of the DCAL option, or whether the program name is a literal or a variable. Any CALL without the pseudo-comment will be compiled as a normal (non-PH) call, either static or dynamic as discussed in above. Therefore, all three types of CALL (PH, static and dynamic) can be freely intermixed within any program.

Any program which is to be called using the PH-call should be separately compiled and prepared as if it were a stand-alone program. Both the called programs as well as the calling program must be prepared with process handling (PH) capability. The group and account where such programs are to reside must also have PH capability.

SECTION 6 USING THE FASTRAN/SEGMENTER

The FASTRAN/SEGMENTER is an interface to the MPE Segmenter designed to simplify and enhance the use of the segmenter with FASTRAN, particularly in situations involving called programs. Since the FASTRAN/SEGMENTER operates by generating and passing commands to the MPE Segmenter, any error messages are actually MPE Segmenter messages. You should consult the MPE Segmenter Manual for error message descriptions.

The major capabilities of the FASTRAN/SEGMENTER are:

- Allowing FASTRAN programs to be compiled which contain more relocatable library (RL) code than will fit into one code segment.
- Initializing and maintaining a segmented library for use with FASTRAN.
- Adding or replacing a called program in a segmented library.
- Preparing a single program file from separately-compiled programs in separate USL files.

To execute the FASTRAN/SEGMENTER, enter the following command:

```
:RUN  FASTSEG.PUB.FASTRAN
```

or use the following UDC:

```
:FASTSEG
```

The FASTRAN/SEGMENTER will display an identifying banner and then will prompt for a command with an equal sign (=).

If you want to use the FASTRAN/SEGMENTER from your own UDC or from a job stream, you can use the INFO= parameter of the :RUN command to enter commands. For example:

```
:RUN  FASTSEG.PUB.FASTRAN;INFO="UPDATESL  SL.MYGROUP:EXIT"
```

would cause the FASTRAN/SEGMENTER to execute the two commands UPDATESL SL.MYGROUP and EXIT. Successive commands in the INFO= parameter are separated by a colon. If the last command is not EXIT, the FASTRAN/SEGMENTER will prompt for additional commands after executing those in the INFO= parameter.

Each FASTRAN/SEGMENTER command is described on the following pages. Following the command descriptions is an example showing how to use the FASTRAN/SEGMENTER to prepare a system of calling and called programs. Examples 2 and 3 in Section 5 also illustrate the use of the FASTRAN/SEGMENTER.

FASTRAN/SEGMENTER COMMANDS

=BUILDSL *sl-file,records,extents*

The =BUILDSL command will create a new segmented library (SL) file. This command has three parameters, all required, as follows:

sl-file: the name to be given to the new SL file, usually 'SL'. The name may be qualified with an account and/or group name, if desired.

records: the maximum number of 128-word records to be allowed in the new SL file.

extents: the maximum number of extents into which the records are to be divided.

For example:

```
=BUILDSL  SL.MYGROUP,2000,20
```

will build a new SL file called SL.MYGROUP with 2000 records divided into 20 extents of 100 records each. This command is identical to the =BUILDSL command in the MPE Segmenter.

=EXIT

The EXIT command terminates the FASTRAN/SEGMENTER.

=INCLUDE *segment,usl-file*

The =INCLUDE command is used prior to a =PREP command to include non-FASTRAN code in the program file, such as procedures called via the PROC verb. This command has two parameters, both required, as follow:

segment: the name of a segment to be included, which may contain one or more procedures.

usl-file: the name of a USL file containing the segment to be included.

=LUSL *library-usl-file*

The =LUSL command tells the FASTRAN/SEGMENTER where to find the library USL (user subprogram library) file. This file is used by the =UPDATESL command (and sometimes by the =PREP command).

Normally this command will not be required. If no =LUSL command has been entered, the FASTRAN/SEGMENTER will use LUSL.PUB.FASTRAN as its library USL file.

=MAIN *program-name,usl-file*

The =MAIN command is used prior to a =PREP command to designate the main program to be prepared. In a system of called programs, the main program is the one where execution is to begin.

This command has two parameters, both required, as follows:

program-name: the name of the main program as it appears in the SYSTEM statement.

usl-file: the name of the USL file which contains the compiled code for the program.

For example:

```
=MAIN  MENU,$OLDPASS
```

will include the main program MENU from the USL file $OLDPASS for a subsequent =PREP command.

=PREP *program-file* [;CAP=*capability-list*] [;PMAP]

The =PREP command prepares a program file from one or more FASTRAN programs. This command has three parameters, one required and two optional, as follows:

program-file: the name to be given to the newly-prepared program file.

CAP=capability-list: a list of capabilities to be assigned to the program. Valid capabilities are: BA, DS, IA, MR, PH and PM. If neither IA nor BA are included, or if the CAP= parameter is omitted, both are assigned. Your MPE user name must have any capabilities that you are assigning to the program file.

PMAP: a procedure map of the prepared program is written to formal file designator SEGLIST. If no file equation has been set, $STDLIST is used.

The =PREP command must always be preceded by a =MAIN command to designate the main program being prepared. One or more =SUBP commands may also be used to designate called programs.

For example:

```
=PREP  MYPROG
```

will prepare the code which was included by any previous =MAIN or =SUBP commands. The program file will be called MYPROG.

The =PREP command in the FASTRAN/SEGMENTER differs from the MPE :PREP command in the following ways:

- Code from more than one USL file may be included in a single program file.
- The program is automatically assigned a MAXDATA of 32000.
- The relocatable library RL.PUB.FASTRAN is automatically used to resolve references to the FASTRAN run-time library (unless the RL command was used to designate a different relocatable library).
- If the program requires more run-time library code than can fit into one MPE code segment, the FASTRAN/SEGMENTER will use two segments. This situation can occur if the program uses a large number of different features of the Transact language. If you try to use the MPE Segmenter to prepare a FASTRAN program and it encounters this situation, it will fail with the message: *RL SEGMENT, CODE SEGMENT OVERFLOW.*

=REPLACE *program-name,sl-file,usl-file*

The =REPLACE command will add or replace a called program in a segmented library. This command has three parameters, all required, as follows:

program-name: the name of the program to be added or replaced in the SL, as contained in the program's SYSTEM statement.

sl-file: the name of the segmented library where the program is to be added or replaced.

usl-file: the name of the USL (user subprogram library) containing the compiled code for the program. The program must be one that will be called by another program via either a dynamic call or a static call with load-time linkage (see section 5). Note that the =REPLACE command will work even if the program was not compiled with the SUBP option.

For example:

```
=REPLACE  MYPROG,SL.MYGROUP,MYUSL
```

will replace the program MYPROG in SL.MYGROUP (or add it if it's not already there). MYUSL is the USL file into which MYPROG was compiled.

=RL *library-rl-file*

The =RL command tells the FASTRAN/SEGMENTER where to find the RL (relocatable-library) file that contains the FASTRAN run-time library procedures. This file is used by the =PREP command.

Normally this command will not be required. If no =RL command has been entered, the FASTRAN/SEGMENTER will use RL.PUB.FASTRAN as its relocatable library file.

=SUBP *program-name,usl-file*

The =SUBP command is used prior to a =PREP command to designate a called program to be included in a system of programs. Any such program must be referenced via a static call (see section 5).

This command has two parameters, both required, as follows:

program-name: the name of the called program as it appears in the SYSTEM statement.

usl-file: the name of the USL file which contains the compiled code for the called program. The =SUBP command will work even if the program was not compiled with the SUBP option.

For example:

```
=SUBP  MYPROG,$OLDPASS
```

will include the called program MYPROG from the USL file $OLDPASS for a subsequent =PREP command.

=UPDATESL *sl-file*

The =UPDATESL command is used to add or replace the FASTRAN run-time library procedures in a segmented library. The run-time library procedures are required in any SL which is to contain FASTRAN called programs. The current =LUSL file (default LUSL.PUB.FASTRAN) is used as the source of the procedures.

This command is used both to initialize an SL for use with FASTRAN and to replace the run-time library procedures when a new version of FASTRAN is released. It requires one parameter, the name of the SL file to be updated.

EXAMPLE: PREPARING A SYSTEM OF PROGRAMS

This example uses the FASTRAN/SEGMENTER directly to accomplish the same result as Case 1 in Section 5. A main program calls three other programs using static calls. All four programs will be prepared into a single program file. The primary differences when using the FASTRAN/SEGMENTER are:

- The individual programs do not need to be compiled into a single USL file. You can keep separate USL files for each program. Then, if a change is required to one of the programs, only that one program must be recompiled. The FASTRAN/SEGMENTER can then re-prep the entire system using the existing USL files. Also, a program which is called as part of more than one system needs to be compiled only once.
- The FASTRAN/SEGMENTER does not require called programs to be compiled with the SUBP option. Thus a program which is run both stand-alone and as part of a system of called programs can be compiled just once to create a USL file. This file can then be used for both purposes by the FASTRAN/SEGMENTER.

The first step is to compile all four programs and save the USL files. We will use the :FASTCOMP command to compile the programs:

```
:FASTCOMP  MAINS,MAINUSL       {MAINS, PROG1S, PROG2S  and   }
:FASTCOMP  PROG1S,PROG1USL  {PROG3S are source file names}
:FASTCOMP  PROG2S,PROG2USL
:FASTCOMP  PROG3S,PROG3USL
```

Next, we use the FASTRAN/SEGMENTER to prepare a program file from the USL files:

```
:FASTSEG
=MAIN  MAIN,MAINUSL          {MAIN, PROG1, PROG2 and}
=SUBP  PROG1,PROG1USL        {PROG3 are system names}
=SUBP  PROG2,PROG2USL
=SUBP  PROG3,PROG3USL
=PREP  PROG
=EXIT
:SAVE  PROG
```

The :FASTSEG command invokes the FASTRAN/SEGMENTER.

The =MAIN command includes program MAIN from USL file MAINUSL.

The three =SUBP commands include called programs PROG1, PROG2 and PROG3 from USL files PROG1USL, PROG2USL and PROG3USL, respectively.

The =PREP command prepares the included programs into program file PROG. The FASTRAN/SEGMENTER automatically sets MAXDATA to 32000 and uses RL.PUB.FASTRAN to resolve external references to the FASTRAN run-time library.

The =EXIT command terminates the FASTRAN/SEGMENTER.

The :SAVE> command makes the new program file permanent.

Beginning with version A.03.F00 of FASTRAN, the FASTRAN compiler automatically gener-

THE FASTRAN/SEGMENTER COMPILER INTERFACE

ates a FASTRAN/SEGMENTER command for each program it compiles. If the compiler is run with the SUBP option on, a =SUBP command (containing the program's SYSTEM name and the USL file name) is generated. If the SUBP option is off (the default), a =MAIN command is generated.

These commands are written to a temporary file called XXFSEGXX. FASTRAN will create this file if none exists. If the compiler is run with the USLI option on (the default), the temporary file is overwritten. If the USLI option is off, the new command is appended to the existing file.

The FASTRAN/SEGMENTER will read and execute the commands contained in this file if it is run with PARM=1. These commands are executed prior to any commands contained in the INFO= parameter.

This enhancement is included primarily to enable the :FASTPREP and :FASTGO commands to use the FASTRAN/SEGMENTER. However, you can examine the definitions of these two UDCs if you want to use this feature in your own UDCs or job streams.

APPENDIX A COMPILE-TIME ERROR MESSAGES

Most language errors in your source program will be detected by FASTRAN during its first pass. Since the program listing is generated during the second pass, most compile errors will suppress your program listing.

Compile errors are listed immediately below the source line in which the error was detected. Even if no source listing is being produced (during pass 1 or if the !NOLIST compiler option is selected), lines containing errors are listed.

Each error message is accompanied by a caret (^) which points to the position where the error was detected. An error number also accompanies each error.

The following table lists all compiler error messages. An explanation follows any message which is not self-explanatory.

NO.	MESSAGE	EXPLANATION
4	INVALID VERB	
5	INVALID ITEM TYPE	The type code for a data item is invalid.
6	MULTIPLE LABEL DEFINITION	
7	INVALID MODIFIER	The verb modifier is not appropriate for the indicated verb, or a required verb modifier is missing.
14	INVALID OPTION	
15	EXPECTING ITEM NAME	
16	ITEM NAME LONGER THAN 16 CHARACTERS	
17	SET NAME LONGER THAN 16 CHARACTERS	An IMAGE data set name is too long.
18	INVALID SYSTEM NAME	
19	MULTIPLE SYSTEM DEFINITION	The SYSTEM statement can only be used once in aprogram.
20	MULTIPLE BASE DEFINITION	A data base name (other than the 'HOME' base) appears more than once in the SYSTEM statement.
21	EXPECTED A COMMAND LABEL	A sub-command label appears before the first command label.
22	EXPECTING A SYSTEM DEFINITION	The first statement in the source program must be a SYSTEM statement.
24	INVALID NUMBER	The number is out of the allowable range for the option in which it appears.
30	SYNTAX ERROR	The compiler could not identify the syntax.
31	INVALID BASE NAME	A data base name does not fit the IMAGE rules for such names.
33	EXPECTING A CHARACTER STRING	A quoted character string is expected.

NO.	MESSAGE	EXPLANATION
34	LABEL LONGER THAN 32 CHARACTERS	
37	STORAGE BYTE COUNT TOO SMALL	The requested storage length will not hold the requested number of digits.
39	DATA TYPE LENGTH NOT SUPPORTED	
42	PASSWORD LONGER THAN 8 CHARACTERS	
46	MULTIPLE OPTION DEFINITION	An option was repeated.
47	MULTIPLE ITEM DEFINITION	A data item name appears in more than one DEFINE(ITEM) statement.
48	MULTIPLE FILE DEFINITION	The same file name appears in more than one FILE and/or KSAM definition.
49	EXPECTING A FILE NAME	
50	INVALID FILE NAME	A name specified in the FILE= or KSAM= option of the SYSTEM statement does not follow the proper syntax for an MPE file name.
51	INPUT STRING LONGER THAN 80 CHARACTERS	
52	EXPECTING A LABEL REFERENCE	
54	MULTIPLE VALUES ONLY VALID FOR COMPARE EQUAL	A condition clause with more than one value on the right of the relational operator can only specify an equal (=) relational operator.
55	NON-PRINTING CHARACTER IN TEXT FILE	Non-printing characters are not permitted in the source file except within a quoted string.
58	CONFLICTING OPTION IGNORED	
60	TOO MANY SORT KEYS	No more than 32 keys may be specified in a SORT option.
62	ITEM REFERENCED TO ITSELF	
65	UNEXPECTED BLOCK TERMINATOR	

NO.	MESSAGE	EXPLANATION
66	UNEXPECTED ELSE STATEMENT	
73	INCOMPLETE BLOCK STRUCTURE IN PRIOR SEQUENCE	
74	VPLUS FORM NAME LONGER THAN 15 CHARACTERS	
75	INVALID VPLUS FORM NAME	
79	SOURCE FILE READ ERROR	The file information display which accompanies this message should indicate the cause of the cause of the problem.
83	EXPECTING SET NAME	An IMAGE data set name is expected.
93	SEGMENT TABLE FULL	No more than 127 segments are allowed.
100	DATA DICTIONARY REQUIRED AND NOT AVAILABLE	The VPLS= option of the SYSTEM statement or the LIST(AUTO) statement could not gain access to the data dictionary.
102	TOO MANY ITEMS IN FILE LIST	No more than 128 data items may appear in the LIST= option.
113	DATA DICTIONARY, CANNOT FIND FILE	
117	CANNOT OPEN INCLUDE FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
118	TOO MANY INCLUDE FILES	More than 5 INCLUDE files have been nested.
120	VPLUS FILE/FORM NOT FOUND IN DICTIONARY	
122	MISSING 'RECNO' OPTION	A data management statement with the DIRECT modifier has no RECNO= option.
125	UNEXPECTED UNTIL STATEMENT	
166	NESTED !IF'S ARE NOT ALLOWED	
167	!ELSE MUST BE PRECEEDED BY AN !IF	
168	!ENDIF MUST BE PRECEEDED BY AN !IF	

NO.	MESSAGE	EXPLANATION
1004	CANNOT OPEN ASSEMBLER SOURCE FILE	The file used to pass the assembly language code to the FASTRAN assembler cannot be opened.
1005	WRITE ERROR ON ASSEMBLER SOURCE FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
1006	CANNOT CLOSE ASSEMBLER SOURCE FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
1007	CANNOT OPEN COMPILER WORK FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
1008	WRITE ERROR ON COMPILER WORK FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
1010	READ ERROR ON COMPILER WORK FILE	The file information display which accompanies this message should indicate the cause of the cause of the problem.
1012	UNABLE TO CREATE ASSEMBLER PROCESS	FASTRAN could not create a process for the FASTRAN assembler.
1013	ABNORMAL TERMINATION OF ASSEMBLER	Internal FASTRAN problem. Contact Performance Software Group.
1014	INTERNAL PROCEDURE OVERFLOW - CIOT	Too much code has been generated for the CIOT (Child Item Offset Table) internal procedure. There are too many child items used in your program.
1015	INTERNAL PROCEDURE OVERFLOW - ATTR	Too much code has been generated for the ATTR (Data Item Attribute) internal procedure. There are too many data items used in your program.

NO.	MESSAGE	EXPLANATION
1016	INTERNAL PROCEDURE OVERFLOW - INAME	Too much code has been generated for the INAME (Item Name) internal procedure. You must limit your program to fewer or shorter data item names or use the OPTI option.
1017	INTERNAL PROCEDURE OVERFLOW - ANAME	Too much code has been generated for the ANAME (Alias Name) internal procedure. You must limit your program to fewer or shorter alias and/or synonym names.
1018	INTERNAL PROCEDURE OVERFLOW - HEAD	Too much code has been generated for the HEAD (Heading Text) internal procedure. You must limit your program to fewer or shorter data item headings.
1019	INTERNAL PROCEDURE OVERFLOW - EDIT	Too much code has been generated for the EDIT (Edit Picture) internal procedure. You must limit your program to fewer or shorter edit pictures.
1020	INTERNAL PROCEDURE OVERFLOW - INIT	Too much code has been generated for the INIT (Program Initialization) internal procedure. Contact Performance Software Group.
1021	INTERNAL PROCEDURE OVERFLOW - OB	Too much code has been generated for the OB (Outer Block Initialization) internal procedure. Contact Performance Software Group.
1022	INTERNAL PROCEDURE OVERFLOW - ITMNO	Too much code has been generated for the ITMNO (Item Number) internal procedure. You must limit your program to fewer or shorter data item names or use the OPTI option.

NO.	MESSAGE	EXPLANATION
1023	INTERNAL PROCEDURE OVERFLOW - VFLD	Too much code has been generated for the VFLD (VPLUS Field) internal procedure. You must limit your program to fewer total form fields.
1024	INTERNAL PROCEDURE OVERFLOW - VFORM	Too much code has been generated for the VFORM (VPLUS Form) internal procedure. You must limit your program to fewer VPLUS forms.
1025	INTERNAL PROCEDURE OVERFLOW - VMOVE	Too much code has been generated for the VMOVE (VPLUS buffer movement) internal procedure. You must limit your program to fewer total form fields.
1026	INTERNAL PROCEDURE OVERFLOW - CMD	Too much code has been generated for the CMD (Command/Sub-Command) internal procedure. You must limit your program to fewer or shorter command and sub-command names and/or passwords.
1098	INTERNAL EXPRESSION EVALUATION ERROR	An internal error has occurred in FASTRAN during expression evaluation. Contact Performance Software Group.
1099	INTERNAL ERROR: TCODE OVERFLOW	An internal error has occurred in FASTRAN during expression evaluation. Contact Performance Software Group.
1105	EXPECTING A CLOSING PARENTHESIS [)]	
1106	EXPECTING A RELATIONAL OPERATOR [=,<>,>,<,>=,<=]	
1108	INVALID SYNTAX IN COMPILER DIRECTIVE	An unidentified compiler directive (beginning with !) was encountered.
1109	EXPECTING A COLON [:]	

NO.	MESSAGE	EXPLANATION
1113	EXPECTING 'THEN'	The condition clause in an IF statement must be followed by THEN.
1129	EXPECTING AN EQUAL SIGN [=]	
1140	DUPLICATE FORM NAME	The same form name appears more than once in the VPLS= option of the SYSTEM statement.
1142	INVALID SYSTEM VARIABLE	An unknown system variable (beginning with $) was encountered.
1144	ITEM(S) NOT DEFINED AND DICTIONARY NOT AVAILABLE	This message is generated at the end of a segment with undefined data items if either the NODICT option was used or if the dictionary (DICT.PUB) could not be opened. The message is followed by a list of the undefined data items.
1145	ITEM(S) NOT FOUND IN DICTIONARY:	This message is generated at the end of a segment with undefined data items if the items were not found in the data dictionary. The message is followed by a list of the undefined data items.
1146	INVALID DICTIONARY ENTRY:	The dictionary entry for the indicated item contains an invalid data item definition.
1147	DUPLICATE ENTRY DEFINITION	A label appears in more than one DEFINE(ENTRY) statement.
1153	EXPECTING AN OPENING PARENTHESIS [(]	
1160	MARKER MAY NOT BE INITIALIZED	
1164	MARKER NOT VALID IN THIS CONTEXT	
1170	UNDEFINED LABEL(S):	This message appears at the end of any segment with undefined labels. It is followed by a list of the undefined labels.

NO.	MESSAGE	EXPLANATION
1171	UNDEFINED ENTRY(S):	This message appears at the end of your program if any labels declared in a DEFINE(ENTRY) statement were not defined.
1176	EXPECTING 'UNTIL'	
1179	EXPECTING A CLOSING BRACKET (])	
1180	NOT A CHILD ITEM	A child item was expected but a parent item was found.
1183	FUNCTION RETURN PARAMETER MUST BE 2, 4 OR 8 BYTES	
1184	VALUE PARAMETER MUST BE 2, 4 OR 8 BYTES	
1185	INVALID PROCEDURE NAME	
1186	CHILD ITEM NOT PERMITTED HERE	
1190	MPE FILE INVALID FOR THIS VERB	An MPE file may not be the object of this verb.
1193	SORT KEY NOT IN ITEM LIST	A SORT= option was used on a data management statement with a fragmented item list and the item list does not contain all of the keys in the sort option.
1104	'KEY' MODIFIER ONLY VALID FOR IMAGE DATA SET	
1200	TOO MANY PARAMETERS	No more than 32 parameters are permitted in a PROC statement.
1201	INVALID SORT SPECIFICATION	
1202	TOO MUCH CODE IN THIS SEGMENT	If the amount of code in the segment cannot be reduced, the segment must be divided into two or more segments. (See the discussion of the CHEK and SSEG options in Section 3 for work arounds.)

NO.	MESSAGE	EXPLANATION
1203	TOO MUCH DISPLAY CODE IN THIS SEGMENT	The code generated by DISPLAY and FORMAT statements in this segment is too large. Either reduce the number of such statements in the segment or divide the segment into two or more segments.
1207	ONLY A RANGE LIST IS VALID IN THIS CONTEXT	
1215	UNDEFINED ITEM(S) IN VPLUS FORM	
1216	EXPECTING A COMMA	
1217	COMMAND LABEL EXCEEDS 16 CHARACTERS	
1220	TOO MUCH SSEG CODE IN THIS SEGMENT	The code generated for data management and VPLUS statements in a program compiled with the SSEG option was too large. The segment must be divided into two or more segments.
1292	CANNOT SUBSCRIPT AN ITEM NOT DEFINED AS AN ARRAY	
1294	TOO MANY SUBSCRIPTS	

APPENDIX B RUN-TIME ERROR MESSAGES

The run-time error messages generated by a FASTRAN program parallel those generated by Transact as closely as possible. As with Transact, the format of a run-time error message is:

*ERROR: *error-message (error-info)*

The *error-info* contains the following fields:

(type,number,program.segment.location[,file])

Each of these fields is described below:

type:

USER: The error is probably the result of an invalid data entry response by the user and can be corrected by re-entering the response.

PROG: The error is probably the result of an error in the program and should be corrected by the programmer.

SYSTEM: The error is probably the fault of the system environment, for example insufficient disc space or an improper file equation for a data base.

As with Transact, the following FASTRAN error types are derived from one of the HP-3000 subsystems. The appropriate subsystem reference manual should be consulted for information about any of these errors.

IMAGE: An IMAGE data base error occurred.

KSAM: A file system error occurred on a KSAM file.

MPEF: A file system error occurred on an MPE file.

VPLUSA: VPLUS subsystem error occurred.

number:

For USER, PROG or SYSTEM errors, this number refers to an error message listed in this appendix. For IMAGE, KSAM, MPEF or VPLUS errors, this is a subsystem error number.

program:

The SYSTEM-statement name of the program in which the error occurred. The program name is useful in tracing errors which occur in systems which use CALL.

segment:

The segment number within the program in which the error occurred.

location:

The code location within the segment in which the error occurred. The code location will appear as the second column of numbers on the program compilation listing and is in octal.

file:

For IMAGE, KSAM and MPEF errors, this is the name of the data set or file on which the error occurred.

USER ERRORS

NO.	MESSAGE	EXPLANATION
1	ENTRY NOT NUMERIC	Non-numeric characters were entered in response to a prompt for a numeric value.
2	INPUT FIELD LONGER THAN *n*	The length of a data entry response exceeds the defined size of the data item.
4	NUMERIC INTEGER PART LONGER THAN *n*	The length of the integer part of a numeric response exceeds the maximum implied by the definition of the data item.
5	NUMERIC DECIMAL PART LONGER THAN *n*	The length of the decimal part of a numeric response exceeds the maximum implied by the definition of the data item.
7	INVALID COMMAND/OPTION: *command/option*	The command or qualifier entered is not defined in the program and is not one of the built-in commands or command qualifiers.
8	INVALID/MISSING SUB-COMMAND: *sub-command*	The sub-command entered is not defined in the program, or no sub-command was entered for a command which requires one.
12	INVALID COMMAND PASSWORD	
13	INVALID SEQUENCE PASSWORD	
16	ATTEMPT TO ASSIGN NEGATIVE VALUE TO ITEM: *item-name*	A LET statement attempted to assign a negative value to a data item declared as positive-only.
17	INVALID ARITHMETIC FIELD	An arithmetic field in a LET statement contained non-numeric data.
97	" "	ITEM UNIDENTIFIED

NO.	MESSAGE	EXPLANATION
18	ENTRY CANNOT BE NEGATIVE	A negative value was entered for a data item declared as positive-only.
19	INVALID LOGICAL CONNECTOR	In response to a DATA(MATCH) prompt, the match expression contains an invalid connector. Valid connectors are 'and', 'or' and 'to'.
20	INVALID PRECEEDING RELATIONAL	In response to a DATA(MATCH) prompt, the match expression contains an invalid relational operator. Valid operators are =, <>, >, <, >= and <=.
21	UNDELIMITED TEXT STRING	In response to a DATA(MATCH) prompt, the match expression contains a quoted string with no closing quote.
22	INVALID PASSWORD FOR DATA BASE: *base-name*	An invalid password was entered in response to a data base password prompt.
100	ITEM NOT FOUND IN LIST REGISTER: *item-name*	In response to a DATA(ITEM) prompt, an item name was entered which was not in the list register.

PROGRAMMER ERRORS

NO.	MESSAGE	EXPLANATION
1	ITEM NOT IN LIST REGISTER: *item-name*	An data item was referenced and was not in the list register.
4	INVALID LIST START POSITION	A range-type data item list was specified, but the start position did not preceed the end position.
7	DATA BASE BUFFER NOT ON WORD BOUNDARY	
16	INVALID RETURN OPERATION	A RETURN statement was executed but there was no pending PERFORM.
18	ARITHMETIC CONVERSION	A data item of type X, U, 9, Z or P contains non-numeric data. Does your program initialize all such data items?
20	INVALID/MISSING KSAM KEY	
21	LIST REGISTER IS EMPTY	An operation was requested which requires at least one data item in the list register and the list register was empty.
23	ITEM NOT FOUND IN VPLUS FORM: *item-name*	A VPLUS operation referenced a data item which was not contained in the current form.
24	VPLUS BUFFER CONVERSION FOR ITEM: *item-name*	An error occurred translating the ASCII value of a VPLUS field to internal format.
25	KEY REGISTER IS EMPTY	An operation was attempted which required a key value and the key register was empty.

NO.	MESSAGE	EXPLANATION
31	ITEM STACK FULL	There is no more room in the list register. You can increase the size of the list register by altering the second parameter of the DATA= option of the SYSTEM statement.
32	IMAGE LIST REGISTER FULL	The maximum size of an IMAGE data item list parameter was exceeded (2048 characters).
33	DATA REGISTER FULL	You can increase the size of the data register by altering the first parameter of the DATA= option of the SYSTEM statement.
34	WORKSPACE FULL	You can increase the size of the workspace by altering the first parameter of the WORK= option of the SYSTEM statement.
36	LEVEL STACK FULL	More than 10 LEVEL statements have been nested.
44	PRINT REGISTER TOO LONG	The work area used in constructing the output of a DISPLAY statement has been exceeded.
46	DECIMAL DIVIDE BY ZERO	
47	DECIMAL OVERFLOW	Could be caused by improper use of the OPTX compiler option. See the discussion of OPTX in Section 3.
48	EXTENDED PRECISION DIVIDE BY ZERO	
49	EXTENDED PRECISION UNDERFLOW	
50	EXTENDED PRECISION OVERFLOW	
51	INTEGER OVERFLOW	Could be caused by improper use of the OPTX compiler option. See the discussion of OPTX in Section 3.
52	FLOATING POINT OVERFLOW	
53	FLOATING POINT UNDERFLOW	
54	INTEGER DIVIDE BY ZERO	

NO.	MESSAGE	EXPLANATION
55	FLOATING POINT DIVIDE BY ZERO	
58	NO VPLUS FORM AVAILABLE FOR UPDATE	An UPDATE(FORM) statement was executed, but no form had yet been displayed.
62	VPLUS FORM NOT FOUND: *form-name*	The VPLUS form designated as the next form in the form file was not defined in the program or in the dictionary.
66	ITEM NAME NOT DEFINED: *item-name*	The item name designated for a VPLUS window operation was not defined in the program.
67	VPLUS FORM IS NOT CURRENT: *form-name*	The CURRENT option was used in a VPLUS statement but the form is not the current form.
68	SORT KEY NOT IN SORT FILE	In a sort, the designated sort key is not in the sort record.
70	ATTEMPTED VPLUS OPERATION WHILE VPLS OPTION SET	No FORM verbs may be executed while SET(OPTION) VPLS is in effect.
73	UNABLE TO CLOSE VPLUS PRINTFILE	
76	ATTEMPTED LN OR LOG FUNCTION ON A NUMBER THAT IS <= 0	
77	READ TERMINATED BY SOFTWARE TIMEOUT	
81	INVALID DECIMAL DIGIT	
84	ATTEMPTED SQRT FUNCTION ON A NUMBER THAT IS < 0.	
86	DBLOCK FAILED AS LOGTRAN LOCKS ARE STILL ACTIVE	A data management statement attempted to issue a lock while a LOGTRAN statement had the data base locked.
93	SUBSCRIPT IS OUT OF RANGE	
95	CANNOT DELETE ITEM IN MATCH REGISTER	
96	CANNOT DELETE ITEM IN UPDATE REGISTER	

NO.	MESSAGE	EXPLANATION
99	PARAMETER MISMATCH IN PROC STATEMENT	The number of parameters coded in a PROC statement does not agree with the number expected by the called procedure. Check your parameters carefully particularly if you are calling an 'OPTION VARIABLE' intrinsic.

SYSTEM ERRORS

NO.	MESSAGE	EXPLANATION
1	SORT INITIALIZATION	An error occurred during execution of a SORTINIT intrinsic.
2	SORT FILE WRITE	An error occurred during execution of a SORTINPUT intrinsic.
3	SORT OUTPUT	An error occurred during execution of a SORTOUTPUT intrinsic.
5	CANNOT OPEN TRANLIST: *file-system-error*	
9	FILE SYSTEM ERROR ON TRANLIST: *file-system-error*	
12	CANNOT OPEN DATA BASE: *base-name*	
13	CANNOT LOAD CALLED PROGRAM:	FASTRAN was unable to load a called program at run-time from an SL. See Section 5 for a complete discussion of CALL.
79	LOG RECORD NOT ON WORD BOUNDARY	
501	CANNOT OPEN TRANIN: *file-system-error*	
502	FILE SYSTEM ERROR ON TRANIN: *file-system-error*	
503	CANNOT CLOSE TRANLIST: *file-system-error*	
505	CANNOT CREATE CALLED PROGRAM: *program-name*	FASTRAN could not create a process for the object of a process-handling (PH) call. Does the called program exist as a program file?
506	CANNOT INITIALIZE CALLED PROGRAM: *program-name*	A SENDMAIL or RECEIVEMAIL intrinsic call failed during initialization of a process-handling call.

APPENDIX C RUN-TIME STATISTICS

The table below illustrates the listing produced by FASTRAN when the STAT option is in effect:

```
*** RUN-TIME STATISTICS ***

CODE SEGMENT REQUIREMENTS:

OUTER BLOCK                          42
PROGRAM INITIALIZATION              384
SEGMENT 0 CODE                     5109
SEGMENT 0 FORMATS                    40
SEGMENT 1 CODE                     4425
SEGMENT 1 FORMATS                    40
SEGMENT 2 CODE                     1264
SEGMENT 2 FORMATS                    40
SEGMENT 3 CODE                     3396
SEGMENT 3 FORMATS                    40
SEGMENT 4 CODE                     1657
SEGMENT 4 FORMATS                  4670
SEGMENT 5 CODE                     2829
SEGMENT 5 FORMATS                  4197
SEGMENT 6 CODE                     1685
SEGMENT 6 FORMATS                    40
SEGMENT 7 CODE                     2247
SEGMENT 7 FORMATS                    40
SEGMENT 8 CODE                     3752
SEGMENT 8 FORMATS                  1219
SEGMENT 9 CODE                     4640
SEGMENT 9 FORMATS                    73
SEGMENT 10 CODE                    2170
SEGMENT 10 FORMATS                  390
SR0 (CIOT)                          152
SR1 (ATTR)                         2149
SR2 (INAME)                        3109
SR3 (ANAME)                        1303
SR4 (HEAD)                         1148
SR5 (EDIT)                         1050
SR6 (ITMNO)                        3559
SR7 (VFLD)                          170
SR8 (VFORM)                          72
SR9 (VMOVE)                         315

TOTAL CODE (EXCLUDING
 RUN-TIME LIBRARY)                57416

DATA STACK REQUIREMENTS:

PRIMARY DB                                  134
OUTER BLOCK
  DATA REGISTER                3075
  VPLUS COMAREA                  60
  MISCELLANEOUS                   6         3141
GLOBAL PROGRAM STORAGE
  DATABASE & FILE TBLS           89
  WORK AREA                     600
  LIST REGISTER                1137
  MISCELLANEOUS                 386         2212

TOTAL DATA STACK
  (EXCLUDING DYNAMIC
   RUN-TIME REQUIREMENTS)      5487
```

The program which generated this example contained 11 segments and approximately 7300 lines of source code. The program used the VPLUS interface but did not use a command/sub-command structure. The fields in this listing are described below and on the following pages.

CODE SEGMENT REQUIREMENTS:		Each entry under CODE SEGMENT REQUIREMENTS represents a single MPE procedure generated by FASTRAN. The maximum size for a FASTRAN-generated procedure is 16128 words.
OUTER BLOCK	42	The number of words of code generated to perform system initialization. Note that if the SUBP (sub-program) option is in effect, the outer block code is not actually generated.
PROGRAM INITIALIZATION	384	The number of words of code generated to perform program initialization. This code is generated and executed for both main and called programs.
SEGMENT 0 CODE	5109	The number of words of code generated for the root segment.
SEGMENT 0 FORMATS	40	The number of words of code generated for FORMAT and DISPLAY statements in the root segment.
SEGMENT n CODE	xxxx	The number of words of code generated for each local segment.
SEGMENT n FORMATS	xxxx	The number of words of code generated for FORMAT and DISPLAY statements in each local segment.
SR0 (CIOT)	152	The number of words of code generated for Service Routine 0 (SR0), which initializes the Child Item Offset Table (CIOT) at program initialization and during segment transfers. The size of this procedure increases by about 1 word for each additional child item referenced in the program.
SR1 (ATTR)	2149	The number of words of code generated for Service Routine 1, which contains the attributes of each data item in the program. The size of this procedure increases by about 5-6 words for each additional data item in the program.

SR2 (INAME)	3109	The number of words of code generated for Service Routine 2, which contains the item names of all data items referenced in the program. The size of this procedure is related both to the number and length of the data items. Each additional 8 character data item name adds about 6 words to this procedure.
SR3 (ANAME)	1303	The number of words of code generated for Service Routine 3, which contains the aliases and synonyms. The size of this procedure is related both to the number of aliases and the length of each alias name. Each additional 8 character alias (or synonym) adds about 6 words to this procedure.
SR4 (HEAD)	1148	The number of words of code generated for Service Routine 4, which contains the heading text for all data items which were defined with a HEAD= option or which had a heading defined in the data dictionary. The size of this procedure is related both to the number of headings and to their length. Each additional 15 character heading adds about 8 words to this procedure.
SR5 (EDIT)	1050	The number of words of code generated for Service Routine 5, which contains the edit pictures for all data items which were defined with an EDIT= option or which had an edit picture defined in the data dictionary. The size of this procedure is related both to the number of edit pictures defined and to their length. Each additional 8 character edit picture adds approximately 5 words to this procedure.
SR6 (ITMNO)	3559	The number of words of code generated for Service Routine 6, which contains the FASTRAN-assigned item number for each item name. The size of this procedure is related both to the number and length of the data item names. Each additional 8 character data item adds about 6 words to this procedure.

SR7 (VFLD)	170	The number of words of code generated for Service Routine 7, which relates the fields on each VPLUS form to the corresponding program item name. Each additional form field adds about 3 words to this procedure.
SR8 (VFORM)	72	The number of words of code generated for Service Routine 8, which contains information about each VPLUS form in the program. Each additional form adds about 8 words to this procedure.
SR9 (VMOVE)	315	The number of words of code generated for Service Routine 9, which contains coded information used to move data between the data register and the VPLUS form buffers. Each additional form field adds about 3 words to this procedure.
SR10 (CMD)	xxxx	The number of words of code generated for Service Routine 10, which contains the information necessary to decode commands and sub-commands defined in your program. The size of this procedure is related to the number of commands and sub-commands in the program, their lengths and the lengths of any command or sub-command passwords. Since the program in this example did not use a command/sub-command structure, no code was generated for SR10.
TOTAL CODE (EXCLUDING RUN-TIME LIBRARY)	57416	The total number of words generated by FASTRAN for the program. To this total must be added the run-time library procedures, which are linked to the program either at prep time from the RL or at load time from an SL. The number of words of run-time library code is usually between 8000 and 15000 words.

DATA STACK REQUIREMENTS:

PRIMARY DB	134	The number of words in the global table and pointer area used by FASTRAN. The size of this area may change with new releases of FASTRAN but cannot be controlled by the programmer. For a main program, the Primary DB area begins at location DB+0. Called programs use the same area as the main program but require a save area of the same size on the data stack.
OUTER BLOCK		The data areas in the outer block are only allocated for main programs. Called programs share this area with the main program.
DATA REGISTER	3075	The number of words in the data register, controlled by the DATA= option on the SYSTEM statement.
VPLUS COMAREA	60	The number of words in the VPLUS Comarea, not under programmer control.
MISCELLANEOUS	6	The number of words of additional storage required in the outer block, not under programmer control.
	3141	The second value following the MISCELLANEOUS entry gives the total number of words in the outer block area.
GLOBAL PROGRAM STORAGE		The number of words of global storage required for the program. For a called program, this area is distinct from the global program storage area of the main program. It is allocated whenever the program is called and is released when the called program is exited.
DATABASE & FILE TBLS	89	The number of words used for IMAGE, KSAM, MPE and VPLUS file tables. The size of this area can be computed as follows: Each IMAGE database requires 15 words plus 1 word for each 2 characters in the data base name. Each MPE file requires 16 words plus 1 word for each 2 characters in the file name. Each KSAM file requires 7 words plus 1 word for each 2 characters in the file name.

		Each VPLUS form file requires 2 words plus 1 word for each 2 characters in the form file name. Each different IMAGE data set referenced in the program requires 1 word.
WORK AREA	600	The number of words required by the work register. This will be twice the value specified on the WORK= option of the SYSTEM statement. Note that FASTRAN uses twice the area as Transact. The extra storage allows FASTRAN to use a different work area allocation algorithm which eliminates the need for run-time garbage collection in the work area.
LIST REGISTER	1137	The number of words required by the list register. This requirement can be determined as follows: Multiply 3 times the number of list register entries requested in the second parameter of the DATA= option of the SYSTEM statement, and add 1. Add one word for each different data item referenced in the root segment. For a segmented program, an additional word is required for each local parent item in the segment with the largest number of parent items, plus one word for each child item in the segment with the largest number of child items.
MISCELLANEOUS	386	The number of additional words required in the global program area. 366 words of this area are fixed and are not under programmer control, though the requirement may change with new releases of FASTRAN. Two additional words are required for each local segment in the program, and one word is required for each two characters in the BANNER= option of the SYSTEM statement.
	2122	The second value following the MISCELLANEOUS entry gives the total number of words in the global program storage area.

TOTAL DATA STACK (EXCLUDING DYNAMIC RUN-TIME REQUIREMENTS)	5487	Total fixed data stack requirement of the program when run stand-alone. The dynamic run-time requirements are difficult to predict for a given program and include data base and file buffers, VPLUS buffers, the RETURN stack, sort work area, local storage for called procedures and various miscellaneous work areas. These areas are allocated only when needed and are released as soon as they are no longer needed.