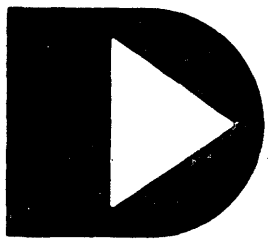
CHAINPLS
CHAIN FILE COMPILER AND EXECUTOR

User's Guide

Version 3

October, 1980

Document No. 50386



DATAPOINT

CHAINPLS
CHAIN FILE COMPILER AND EXECUTOR

User's Guide

Version 3

October, 1980

Document No. 50386

NOTICE

Datapoint strongly recommends that its customers use Datapoint Customer supplies. These disks, diskettes, cassettes, ribbons and other products are certified by Datapoint to meet all Datapoint Hardware specifications for consistent optimum performance.

PREFACE

CHAINPLS is designed to improve and enhance the capabilities of the CHAIN utility. It significantly improves the compile-time and execution-time capabilities of CHAIN while accepting CHAIN-format input files without significant alteration. This manual contains documentation on the features and run-time characteristics of CHAINPLS.

TABLE OF CONTENTS

CHAPTER 1. INTRODUCTION

1.1 CHAINPLS Data Flow

The execution characteristics of CHAINPLS are similar to CHAIN. A single file is presented to CHAINPLS containing internal tailoring commands and an output file is generated containing the DOS command lines and keyin responses necessary to execute a desired job stream. CHAINPLS is not intended to be a total replacement to CHAIN. However it will execute CHAIN files with a minimum of changes while allowing extended capabilities during file compilation for arithmetic computation, character string manipulation, assignment of default values, structuring and looping, and execution-time system status information. During execution, additional CHAINPLS features include system logging capabilities and versatile restart control following system failure.

CHAINPLS has two basic phases. The first phase is the compilation or "tailoring" phase in which the input file is read and the control statements are modified according to the directives found in the file. The output records (command lines and keyin responses) are written into a control file whose default name is SYSJOB/SYS. If logging of console display/keyin activity is requested, a log file whose default name is SYSLOG/SYS is created.

The second phase of CHAINPLS consists of actually executing the command lines in the SYSJOB/SYS file. All keyin responses requested by programs executing under CHAINPLS control are taken from the SYSJOB/SYS file. At the end of the SYSJOB/SYS file, control is returned back to the Disk Operating System.

CHAINPLS may be run on any 5500-level processor on any of the DOS. series operating systems. It is A.R.C. compatible. CHAINPLS may not, however, be run in the remote partition of PS66 or the fixed partition of PS or U.P.S.

1.2 "Execute" and "Compile-Only" Modes

The normal mode of operation for CHAINPLS is to tailor the input file and immediately execute the results. This is "execute" mode. On occasion, however, the user may wish to verify the results of a CHAINPLS compilation. The "compile-only" mode causes the result of the compilation phase to be written into a GEDIT standard text file. This file will be completely tailored and will contain any comment lines which would normally have been displayed on the screen. Note that this output text file can usually be executed using the normal DOS CHAIN utility. This feature can be of value when transporting software to a Datapoint 2200 or any time when the released DOS CHAIN program must be used.

1.3 The CHAINPLS Command Line

The command line for the execution of CHAINPLS is:

```
CHAINPLS <input file spec>,<output file spec>,<print file spec>,
  <log file spec>;<OPTIONS=options>,<chain replacement equate>,
  <replacement equate>,...
```

The <input file spec> is a DOS-standard file specification describing the input text file containing the untailored text. It may contain a member name if the input is from a library. If the input is from a text file the default extension is /TXT. If the input is from a library the default is /LIB.

The <output file spec> describes the output file desired. If this entry is given for normal executions, it will be used as an override name for the system job file SYSJOB/SYS. If CHAINPLS is being run in "compile-only" mode, the output file generated is a DOS-standard text file. The default extension of the <output file spec> is /CHN if "compile-only" and /SYS if executing.

The <print file spec> is used if the user desires a print file listing of the compilation. The default extension of the print file is /PRT.

If compiling, the default name of the output file is the name of the input file. The default name of the print file is the name of the output file. The compile-only output file is in DOS-standard, non-space-compressed format.

The <log file spec> is used to name the system log file if a name other than SYSLOG/SYS is desired. (See "Execution Logging".)

The reserved word "OPTIONS" is used to assign compile-time and execution-time options to CHAINPLS. The word "OPTIONS" may be replaced by "OPTN" or "OP". For example:

```
CHAINPLS FILE1,FILE2;OPTIONS=LI,A=100,B=XYZ
```

specifies "LI" to be the execution options; A and B are data items with values of 100 and XYZ respectively. The options available are:

L	list results on the local printer
P	printer output goes to a print file
S	printer output goes to a servo printer
D	display output and additional lines on console
I	display/list the input file
R	display/list input lines after replacement
O	display output lines
T	activate any imbedded DEBUG statements
C	compile-only; generate output file and stop
N	no output file or execution desired
G	log all console messages during execution
Q	log messages into existing log file
U	unstack all CHAINPLS recursion

The D option enables the display driver and will cause the input and output line numbers to be displayed on the bottom of the console screen. The P, L, and S options enable the appropriate printer driver and will automatically cause all output lines to be printed. For further information on the printer capabilities, see the chapter on "Display and Print Options".

The I, R, and O options control the data to be listed or displayed by the drivers selected. The I option will cause all input statements to be listed or displayed. The R option will cause all input to be listed or displayed after line replacement. The O option will cause all output lines to be displayed.

The C option causes CHAINPLS to compile (tailor) the input file completely, but instead of actually executing the output, the results are written to a user-specified file for later use. This option is useful for testing a complex compilation or generating multiple CHAINPLS files for later use.

The G and Q options cause all execution-time console traffic (displays and keyins) to be logged into the disk file specified by the <log file spec> (which defaults to SYSLOG/SYS). This file is in DOS-standard text file format. The G option initiates logging

and causes a new log file to be prepared. The Q option notifies CHAINPLS that the log file file already exists and new output should be written onto the end of the existing data. Note that if CHAINPLS is invoked by CHAINPLS and logging is active at the outer (prior) level of chaining, those options are automatically passed through to the inner (newer) level. This is explained more fully during discussion of the execution phase.

The U option alters the behavior of CHAINPLS when the execution phase of CHAINPLS has called the compile phase of CHAINPLS. This "CHAINPLS calling CHAINPLS" is called "recursion". Normally, as with CHAIN, if file "A" is chained, and file "A" contains a command to chain another file, say "B", then the completion of the commands in file "B" will result in control returning to file "A" at the command line just following the command to chain file "B". There is no practical limit to the number of times CHAINPLS can call CHAINPLS (recurse). The U option, when set, causes CHAINPLS to ignore all prior levels of recursion and behave as though its command line came directly from the keyboard. This can be useful, for example, in allowing an error procedure to run to completion and then return control back to the DOS without completing prior levels of chain files.

It is important to note that if CHAINPLS calls CHAINPLS, any override names for the system job file (SYSJOB/SYS) and the system log file (SYSLOG/SYS) specified for the second CHAINPLS execution are ignored. The output and log data will go into the same files as were used during the first execution.

The command line may be continued across many keyin lines by using the dash (-) as a line continuation marker. The dash may only appear, however, in the same position normally occupied by a comma.

Since the comma (,) and the dash (-) cannot normally be entered as part of a value string on the command line, special provision has been made for a "forcing character." The pound sign (#), if used on the command line, tells CHAINPLS that the character immediately following is part of the value of the data item and should not be examined for any other purpose. NOTE: the forcing character is only applicable to the command line.

1.4 Compile Phase Activity

During the compile phase, the input file is read, all statements are selected and tailored, and an output file is written. If CHAINPLS is running in its normal mode (compile and execute), the output is written into a file called SYSJOB/SYS. If CHAINPLS was invoked by CHAINPLS, the new output is added onto the end of the existing SYSJOB/SYS file. This added data is called an "extent". There is no practical limit to the number of extents which can be written to the SYSJOB/SYS file. Each extent in the SYSJOB/SYS file is the result of a compilation phase of CHAINPLS. During execution, the user will be notified as each extent is completed and the name of the input file used to create the extent is displayed on the system console. Note that only command lines, keyin response lines, and execution-time CHAINPLS directives are allowed in the SYSJOB/SYS file.

If CHAINPLS is being run in compile-only mode, all statement selection and tailoring is done normally, but the output is written into the output file specified by the user and is not executed.

If logging is specified, the SYSLOG/SYS file is either created or searched for its current end-of-file depending on the option used on the command line.

1.5 Execution Phase Activity

At the end of the compilation phase, CHAINPLS makes certain modifications to the resident DOS to allow keyin responses to be read from the SYSJOB/SYS file and delivered to the requesting program. If logging was specified, additional modifications are made to cause all displayed messages (including keyin lines) to be written to the file SYSLOG/SYS. It is common to note some degradation in system performance when logging is active, but the overhead imposed by the logging facility is only present during actual console display. The overhead is due to the disk operations necessary to support the logging facility.

The execution phase of CHAINPLS requires the presence of CHAINPLS/OV1 and CHAINPLS/OV2 on a common on-line drive. CHAINPLS/OV1 is the command handler and is invoked between each program execution. CHAINPLS/OV2 is the logger and is loaded whenever necessary to create new records in the SYSLOG/SYS file.

CHAPTER 2. COMPILE PHASE DIRECTIVES OF CHAINPLS

2.1 The Data Types of CHAINPLS

2.1.1 Definition of a Data Item

A data item as used by CHAINPLS is a name consisting of eight characters or less beginning with an alphabetic character and containing no imbedded special characters or blanks. Associated with any data item is a Boolean value indicating whether the item was specified in the command line to CHAINPLS or was subsequently SET to TRUE or used as the receiving field in an ASSIGN statement. Also, all TRUE data items have an associated value which may be null. The value types are CHARACTER, DECIMAL, or OCTAL. This associated value is in addition to the Boolean value of the data item; whether the Boolean value of a data item or its associated value is used in a particular expression depends upon the particular operation being performed. Note that lower case alphabetic characters are accepted as any part of data item name.

As with CHAIN, the use of the standard replacement character (#) surrounding a data item name will cause the value associated with the item to be inserted into any line of the input file; this is true regardless of statement type.

2.1.2 Boolean Data

All data types used by CHAINPLS possess, at all times, a Boolean value. If a data item is not specified in the command line, its initial Boolean value is FALSE; if it was specified its initial value is TRUE. Subsequent ASSIGNment of any numeric value or character value to the data item will cause it to assume a TRUE Boolean value.

2.1.3 Arithmetic Data

The association of arithmetic data with a data item causes the data item to posess a data type of numeric. If the equated value on the command line for the data item is numeric (consisting only of digits), or if subsequent ASSIGNments of values to the data item are numeric, the data type for the item is numeric. If the number on the command (or in the ASSIGNment) line is OCTAL (composed of digit 0-7 and preceded by a zero) then the item is considered OCTAL, otherwise it is considered DECIMAL. Note that a minus sign (-) may precede any numeric value. This data type association can change during a single execution of CHAINPLS by use of the ASSIGN and SET statements. The number base type (DECIMAL or OCTAL) is preserved and utilized when the value of the data item is formatted into a CHAINPLS statement. That is, a DECIMAL value is automatically converted to decimal, an OCTAL value is automatically converted to OCTAL.

All internal arithmetic values are maintained as signed 23-bit binary values. This allows bit manipulation operations to be performed. The 23-bit length limits numeric range to -8388608, +8388607. In octal the range is -040000000, +037777777.

2.1.4 Character String Data

The association of character string data to a data item causes the data item to posess a data type of CHARACTER. If the equated value on the command line for the data item is not numeric or if subsequent assignments of values to the data item are character strings, the data type for the item will be CHARACTER. All character string values are maintained internally in standard ASCII characters with an associated string length indicator. The maximum length of a character string is 80 bytes. This limit also applies to input lines (from KEYIN or READ). The end-of-line character counts as one of the 80 characters on input so the actual maximum number of displayable characters is 79.

2.1.5 Literals

Three types of literals are allowed in CHAINPLS. These literals may appear in any expression anywhere a data item name could normally be used. Any character string enclosed in double quotes (") is considered type CHARACTER and its length is determined by the length of the enclosed string. Null length character literals are allowed (""). Any numeric string is considered a numeric literal. The data type of a numeric literal is determined by the presence or absence of a leading zero. If a

leading zero is present, the literal is considered to be OCTAL; otherwise is it considered DECIMAL. The use of a literal zero is a special case. If a single zero is used as a literal, it is considered to be a decimal literal; two or more zeroes is considered to be an octal zero.

2.1.6 Redefinition of Data Item Types

At any time during an execution, the type of a data item may be altered by use of the ASSIGN statement. This allows any data item to be defined as a certain type of data, and redefined as another type later in the program. There is no limit to the number of times a data item may be redefined, but care must be taken that the type of the item is clearly understood by the programmer. This feature could be of benefit should the user encounter memory size constraints during execution. The redefinition of a data item does not allocate any new memory to the item unless necessary. Each data item is given an initial allocation of 14 bytes. If character string operations or reassignments lengthen the data items, new memory will be allocated up to 80 bytes.

2.2 The Expressions of CHAINPLS

An expression in CHAINPLS can consist of any of four types of operators: logical operators, arithmetic operators, binary operators, or character string operators. The logical operators all generate a Boolean result based on the truth or falsity of a statement. The arithmetic operators are the standard mathematical operators used in most languages. The binary operators perform bit manipulation upon the values associated with a data item. The character string operators are used to create new character strings from previously existing strings, literals and the values of other data items.

The arithmetic operators are:

+	addition
-	subtraction
*	multiplication
/	division

The binary operators are:

&&	binary "and"
¦¦	binary "or"
!	exclusive binary "or"
>>	binary shift right
<<	binary shift left

The logical operators are:

¦	logical "or"
&	logical "and"
~	logical unary "not"
=	logical equal
<	logical less than
>	logical greater than
<=	logical less than or equal to
>=	logical greater than or equal to
~=	logical not equal to

The character string operators are:

\\	concatenation
^	sub-stringing
:	sub-string control
.	string length function
¦	pattern match
[[	string scan
~¦[	string scan unequal
\	string replacement

Operations are evaluated in the order of their occurrence, but parentheses to any depth may be used to alter the order of evaluation.

If a data item with a data type of character string is used in an arithmetic expression, the high-order 24 bits (first three bytes) of the character string are used as a binary value.

2.3 The CHAINPLS Compilation Directives

The general form for a directive statement in CHAINPLS is:

```
// <directive> <data items or expression>
```

Any number of spaces can occur anywhere in the statement; the "//", however, must occur in column 1.

2.3.1 The program block

The conditional statements (IF/ELSE/XIF, DO/END, WHILE <condition>/END) and the BEGIN/END statements create a program block. All of the statements in a block are one logical entity. There is a limit on the maximum number of blocks open at one time. In the current CHAINPLS the limit is 42.

2.3.2 The IF/ELSE Directive

The basic syntax of an IF directive is:

```
// IF <any expression>
....any number of statements
// ELSE
....any number of statements
// XIF
```

The first group of statements is processed if the expression evaluated in the IF statement is TRUE; the second group of statements is processed if the expression is FALSE. The ELSE statement and its associated statement block is optional. Note that IFs may be nested to any depth up to the number of number of free block levels, and BEGINs are not necessary for stacking of IFs. This differs from CHAIN requirements.

2.3.3 The IFS/IFC Directives

The basic syntax of an IFS directive is:

```
// IFS <any expression>
```

The basic syntax of an IFC directive is:

```
// IFC <any expression>
```

The IFS and IFC directives are similar to the IF directive but **are only provided for compatibility with old CHAIN files. It is important to note that it is not intended that they be used in new CHAINPLS files.**

The IFS and IFC directives are defined exactly as they were in the old CHAIN; that is, the IFS is TRUE if the <expression> is true, but the IFC is TRUE if the <expression> is false.

IMPORTANT: For compatibility, ALL commas (",") and periods (".") in any IFS or IFC statment are TRANSLATED to "¦" and "&" respectively. This translation is internal to CHAINPLS and does NOT APPLY TO ANY STATEMENTS OTHER THAN IFS AND IFC! Due to the compatibility problem, IFS and IFC should be assiduously avoided in new CHAINPLS files. Any logical statement is possible with an IF statement. Note that the logical not ("~") operation may be performed to invert the value of any expression, thus eliminating the need for an IFC directive.

2.3.4 The XIF Directive

The XIF directive is used to terminate a block created by an IF or ELSE statement. Note that, unlike the older CHAIN, any XIF only closes one block; that is, an XIF statement only affects the block created by the last IF or ELSE statment. For example:

```
// IF <expression>        first IF
....statement
// IF <expression>        second IF
....statement
// ELSE                   inversion of second IF
....statement
// XIF                    terminate second IF/ELSE
// XIF                    terminate first IF
```

It should be carefully noted that, for compatibility with the old CHAIN, the behavior of XIF is different with the IFS and IFC instructions. If an old-style CHAIN file containing IFS and IFC statements is processed by CHAINPLS, the XIF directive associated with the IFS and IFC statements (and their ELSEs) will close all existing blocks created by IFS and IFC statements; that is, XIF will behave as it did in CHAIN. This feature is provided only for compatibility with the CHAIN and should not be used in new CHAINPLS input files.

2.3.5 The BEGIN Directive

The BEGIN directive has the same definition as that used in CHAIN; that is, the BEGIN directive starts a block of statements which, from the level in which the BEGIN statement occurs, is to be considered one statement. The format of the BEGIN directive is:

```
// BEGIN
```

Once again, please note that **BEGIN statements are NOT necessary to create multiple levels of IF statments as they were in the old CHAIN. The BEGIN statement is thus unnecessary and is provided for compatibility only.**

2.3.6 The WHILE Directive

The WHILE directive is used to conditionally process a block of statements repeatedly until a condition becomes false. The format of the while statement is:

```
// WHILE <any expression>
```

2.3.7 The END Directive

The END directive has the same definition as that used in CHAIN; that is, the END directive terminates a block of statements which, from the outer lexic level, are considered to be one statement. END statments are used to terminate the blocks created by BEGIN and WHILE. The format of the END directive is:

```
// END
```

2.3.8 The DO Directive

The DO directive begins a block of statements which are always executed once and will be repeated if the expression associated with the terminating UNTIL statement is found to be FALSE. The format of the DO directive is:

```
// DO
```

2.3.9 The UNTIL Directive

The UNTIL directive is used to terminate a block of statements begun by a DO directive. If the expression associated with the UNTIL directive is TRUE, the block will not be reprocessed. If it is FALSE, execution will resume with the DO statement. The format of the UNTIL directive is:

```
// UNTIL <any expression>
```

2.3.10 The Command Line as Assignment Directive

When a data item is entered on the command line, its associated Boolean value is set to TRUE. If the entry is equated to a character string, decimal number, or octal number, that string or number determines the type and value of the data value. If no value is equated to the data item, its data type is set to CHARACTER and its value to NULL.

2.3.11 The ASSIGN Directive

The ASSIGN directive is used to assign or alter the value or the data type of a data item. The format of the ASSIGN statement is:

```
// ASSIGN <data item name>=<expression>
```

If the <expression> consists only of one data item (e.g., "A=B"), the data type of the receiving data is set to the type of the sending data item and the value of the sending data item is transferred to the receiving data item. To prevent data type alteration by reassigment, a trivial expression may be used. For example:

```
// ASSIGN FIELD1=FIELD2
```

will cause FIELD1 to assume all characteristics of FIELD2. However:

```
// ASSIGN FIELD1=(FIELD2)
```

will reassign only the value FIELD2 to FIELD1.

Numeric literals are allowed in any expression. If the first character of a numeric literal is a zero (0), the literal is considered to be in OCTAL form; otherwise it is interpreted as

decimal. Alphanumeric literals are allowed in any expression and must be enclosed in quotes. The maximum lenth of a literal is 80 bytes. Null-length alphanumeric literals are allowed. A null length literal is two adjacent double quotes ("") and is typically used to reset the length of a data item value to zero.

The ASSIGN directive can also be used to create a data item that did not previously exist. The Boolean value of a newly declared data item is set to TRUE and the value and type of the data associated with the data item is determined by the evaluation of the expression. Since the value type of intermediate arithmetic results is always DECIMAL if the <data item name> is being created by the ASSIGN statement and is ASSIGNed the value of an arithmetic expression, the data item will receive a value type of DECIMAL. The reserved word "NULL" may be used to assign a null value to any data item. Note that the result field of any assign statement is always Boolean TRUE after the execution of the statement. To change the Boolean value of a variable, the SET statement must be used. Note that the Boolean value of any undeclared data items is implicitly FALSE.

2.3.12 **The SET Directive**

The SET directive is the Boolean equivalent of the ASSIGN directive. The format of the SET directive is:

```
// SET <data item name>=<expression>
```

The SET directive, like the ASSIGN directive, can be used to create new data items. As with command line entry, the creation of a new data item involves setting the new item's data type to CHARACTER and its value to NULL. The SETting of an existing data item to a non-existing data item causes a value of FALSE to be transferred to the receiving data item. The reserved words FALSE, NO, and OFF may be used to indicate a FALSE value in the expression while the reserved words TRUE, YES, and ON maybe be used to indicate TRUE.

Note that any previous value associated with the result variable is lost after a SET statement.

2.3.13 The DEBUG Directive

The DEBUG directive is intended to provide the user with a means of determining the source of error in a CHAINPLS execution. The format of the DEBUG statement is:

```
// DEBUG <data item name> <data item name>...
```

When, during execution, the DEBUG statement is encountered, the contents and flags of the symbol table entries specified will be listed or display on the media specified. The DEBUG statement has no effect unless the "T" option is specified on the command line. This allows the DEBUG statements to remain imbedded in the text file.

2.3.14 The CLICK Directive

The CLICK directive allows the user to provide a click as audible output from the execution of CHAINPLS. The format of the CLICK statement is:

```
// CLICK
```

2.3.15 The BEEP Directive

The BEEP directive allows the user to provide a beep as audible output from the execution of CHAINPLS. The format of the BEEP statement is:

```
// BEEP
```

2.3.16 The ABORT Directive

The ABORT directive allows the user to discontinue the execution of CHAINPLS due to a detected internal error condition. The format of the ABORT statement is:

```
// ABORT
```

2.3.17 The KEYIN Directive

The KEYIN directive allows the user to display data to and request input from the console operator. The format of the KEYIN statement is:

```
// KEYIN <operand or alpha literal> <operand or literal> ....
```

Any alpha literals (character strings enclosed in double quotes) which are encountered in the statement are displayed. Any operand names encountered cause the cursor to appear in the character position following the last displayed item. The console operator must then keyin the data requested. The data entered is checked against the value type of the operand. If the value type is DECIMAL, only the characters 0 through 9 are allowed. If the value type is OCTAL, only the characters 0 through 7 are allowed. If the value type is CHARACTER, any characters may be entered. Negative numbers are allowed. Up to 80 characters may be entered into a CHARACTER string. However, the enter character counts as one of the 80 characters.

Any data item can de displayed by using the replacement operator e.g.

```
// KEYIN "#VAR#"
```

would cause the current value of var to be displayed. It is important to remember that replacement is done before the directive is executed. This means that "VAR" in the above example must not contain a quote character. There is no forcing character in the KEYIN directive.

2.3.18 The INCLUDE Directive

The INCLUDE directive is used to copy in other text files and cause them to be interpreted as part of the input file. The general format of the INCLUDE directive is:

```
// INCLUDE <data item name or alpha literal>
```

The value of the <data item> or literal used in the include statement is a DOS file specification. The default extension is /TXT if the file specification does not contain a member name. The file specification may contain a member name. If a member name is given the extension will default to /LIB. If the CHAINPLS input file is a library member, other members of that library may be included by having the value of the data item or literal contain only the member name (eg. ".MEMBER").

An INCLUDEd file may INCLUDE still more text files, but at no time can the total number of outstanding inclusions exceed 14. It is important to note that blocks created in an outer level source text file (such as by IF, WHILE, ELSE, or BEGIN) cannot be completed (with END or XIF or ELSE) by any statements within the included source text file. The INCLUDEd source text file can create blocks of its own, but these blocks must be closed (with END or XIF) before the end-of-file record in the INCLUDEd file is reached. At end of file on the inclusion, processing is resumed with the statement following the INCLUDE statement in the earlier file. In other words, included files are used in a last-in, first-out (LIFO) stack basis.

2.3.19 The EXIT Directive

The EXIT directive is designed to provide a means of selecting the program to be executed next after CHAINPLS returns to the operating system at the end of the job. This statement is only effective if CHAINPLS is being run in "compile-only" mode.

The format of the EXIT directive is:

```
// EXIT <data item name or literal>
```

The <data item> used in the EXIT statement must have an associated data type of CHARACTER. The value of the data item or literal is placed in the Monitor Communications Region (MCR$) of the DOS and flagged to indicate that the command line is to be interpreted before returning to the keyboard for operator instructions.

Multiple EXIT statements can be used in a single CHAINPLS execution, but only the last one encountered prior to end-of-job is active.

2.3.20 The STOP Directive

The STOP directive is used to terminate interpretation of the input file at the current line. It also allows the user to select the program to be executed after CHAINPLS returns control to the operating system. The format of the STOP directive is:

```
// STOP <data item name or literal>
```

The <data item or literal> is optional, and, if specified, its value will be placed in MCR$ in the DOS. The <data item name or literal> is only utilized by CHAINPLS if it is being run in

"compile-only" mode. The primary function of the STOP statement is to facilitate program termination without the necessity of creating specialized block exits to clear all declared blocks; no checking is done to determine if the user has properly closed all blocks.

2.3.21 The DISCARD Directive

The DISCARD directive is used to eliminate a previously used data item from the symbol table. In addition, the memory areas dedicated to it are released and become available for reuse by other data item declarations. The format of the DISCARD directive is:

```
// DISCARD <data item name>(,<data item name>...)
```

Any number of data item names may appear on the statement line.

2.3.22 The OPTIONS Directive

The OPTIONS directive is used during compilation to set or reset certain display and print capabilities of CHAINPLS. Of the possible execution options available to the user from the command line, only the "D" (display), "I" (input), "R" (replace), "O" (output), and "T" (debug) options may be used in the OPTIONS statement. The format of the options statement is:

```
// OPTIONS <char data item or alpha literal>
```

The modified options appear in a character string or an alpha literal. If the option is to be set on, the character alone appears; if the option is to be set off, the character is preceded by a minus sign (-). For example:

```
// OPTIONS "D-OT"
```

will set the "D" option on, cancel the "O" option, and set the "T" option on. The use of a character variable is demonstrated as follows:

```
// ASSIGN OPZ="-I-TR"
// OPTIONS OPZ
```

In this case, the "I" and "T" options are set off, and the "R" option is set on.

2.3.23 The USERPROG Directive

The USERPROG directive is used to create new statement types at user request. Basically, the USERPROG directive causes a relocatable assembly language module to be dynamically loaded into memory. The format of the USERPROG directive is:

```
// USERPROG <statement name>(<library name>,<member name>)
```

where the <library name> and <member name> are data item names or literals.

The USERPROG directive will cause the <library name> to be opened and searched for the <member name>. The relocatable member is then loaded into memory and the <statement name> is inserted into the symbol table and marked as a directive. When a statement is encountered which contains the <statement name> as its directive, control is passed to the relocatable subroutine. For example, if the USERPROG statement were:

```
// USERPROG STARTIME("CHAINPLS/REL","MODULE1")
```

the program "MODULE1" would be located in the library "CHAINPLS/REL" and loaded into memory. A symbol table entry for the name "STARTIME" would be created and marked as though it were a standard statement directive. The address of the processing (parsing) routine for the statement would be set to the entry point of "MODULE1". If a statement such as:

```
// STARTIME XTIMER
```

were encountered, then statement would be broken into symbols and control would be passed to the relocatable subroutine "MODULE1". Further information on this feature can be found in the chapter on "USER-WRITTEN SUBROUTINES".

2.3.24 The User File Input/Output Directives

CHAINPLS provides the ability for a user to read and write text disk files during the execution of CHAINPLS. Any standard TXT-format file can be read as input. The input files can be space-compressed or non-space-compressed, record-compressed or non-record compressed. The output files written by CHAINPLS are record-compressed, non-space-compressed text files.

2.3.24.1 The OPEN Directive

The OPEN directive is used to initialize the processing of any input or output user files. The format of the OPEN directive is:

// OPEN <data item name> (<data item name or literal>)

The first <data item name> is the name of the data item to be used by CHAINPLS as the internal name of the file. After the OPEN statement is executed, this data item possesses a data type of "file" (a sub-class of Boolean) and any previous value it may have possessed is lost. This file data item name is used to reference the file in READ, WRITE and CLOSE statements. If the file data item is subsequentely used in an expression, it is interpreted as a Boolean whose value is TRUE if the file is OPEN and FALSE if the file is CLOSEd.

The <data item name or literal> enclosed in parentheses is used as the name of the file to be opened. Thus, if a data item is used, it must be a character string. The default extension is /TXT unless a library member name is supplied in the data item name or literal (see below). If a file of that name (on the selected drive, if specified) exists, it is opened, if not, a file of that name is PREPared. The same OPEN directive is used for both input and output files; subsequent use of the file determines action to be taken at CLOSE time.

Library members can be OPEN'ed and read. It is only necessary to include the member name in the <data item name or literal>. If a member name is specified the file extension will default to /LIB instead of /TXT. If the CHAINPLS input file is a library member other members of that library may be OPEN'ed as user files by having <data item name or literal> contain only the member name (eg. ".MEMBER").

Great care should be taken to insure that a user file opened to a library member is never used in a WRITE statememt. Any attempt to WRITE to a library member will destroy the library structure leaving the file in a undetermined state.

2.3.24.2 The CLOSE Directive

The CLOSE directive is used to terminate processing of the specified file. The format of the CLOSE directive is:

```
// CLOSE <file data item name>
```

The <file data item> must have been used in an OPEN statement or an error will result. If any WRITE statements have been issued to the file, a DOS-standard end-of-file mark will be written into the proper sector of the file. Any subsequent use of the file data item name in an expression will cause a value of Boolan FALSE to be used.

2.3.24.3 The READ Directive

The READ directive is used to bring data from an input file into the value area of a character string data item for processing. The format of the READ directive is:

```
// READ <file data item name>,<data item name>,<data item>,...
```

The <file data item name> must have been used in a successful OPEN statement. The data from the next sequential record in the file becomes the value of the <data item name>; this data item always assumes a data type of CHARACTER after a READ operation. The length of the data item value is set to the length of the record. If the record is longer than 80 bytes, the following bytes in the record will be stored in the additional data items in the data item list. If insufficient data items are listed to contain the entire record, the remaining part of the record is discarded. If too many data items are listed, their length is set to NULL.

At end-of-file on the input file, the variables in the list are FALSE; that is, end-of-file may be detected by checking the Boolean value of the first data item for FALSE. Any subsequent attempts to read the file after end-of-file has been detected will also return FALSE values.

2.3.24.4 The WRITE Directive

The WRITE directive is used to place data from the values of various data items onto an output file. The format of the WRITE directive is:

// WRITE <file data item name>,<data item name>,<d.i. name>...

The <file data item name> must have been previously used in a successful OPEN statement. There is no limit to the number of <data item names> which may be included in a WRITE statement; in fact, none are required (this would generate a null length record). Character literals are also allowed in place of data item names. The data items may be of any type and may be of NULL (0) length. There is no limit to the size of the output record. Care must be taken when using NUMERIC or OCTAL data items in an output record as the length of the values written out will depend on their current numeric size. The occurrence of a WRITE directive to a file data item will cause an end-of-file to be written to the proper sector at CLOSE time. Note that FALSE data items are effectively ignored during WRITEs.

It may be useful to note that both READs and WRITEs may be done to the same file. A file may be read to its current end-of-file and have additional records appended to it by means of WRITE statements. Care must be taken if this feature is to be used, however, since programs written in this manner are not easily restarted.

2.3.24.5 The ENQUEUE Directive

The ENQUEUE and DEQUEUE directives are used under the Attached Resource Computer (A.R.C.) system to allow exclusive use of user files. The capabilities and limitations of the A.R.C. "enqueue/dequeue" mechanism are fully discussed in the A.R.C. User's Guide and should be carefully understood before using these directives. The format of the ENQUEUE Directive is:

// ENQUEUE <num literal>,<file data item>,(<f.d.i.>...)

The <numeric literal> is the "enqueue level" for the request and must be either a 2 or 3; if no <numeric literal> is specified, 3 will be used. Any <file data item> used in an ENQUEUE statement must be currently OPEN. No more than 16 files may be included in a single ENQUEUE statement.

2.3.24.6 The DEQUEUE Directive

The DEQUEUE directive is companion to the ENQUEUE directive and is used to release resources obtain for exclusive use. The format of the DEQUEUE directive is:

```
// DEQUEUE
```

The DEQUEUE directive will automatically release all file previously ENQUEUEd. No harm can result from the use of a DEQUEUE directive at any time, regardless of whether files have previously been ENQUEUEd or not.

2.3.25 The Special Statement Directives

The special statement directives are designed to allow commenting of CHAINPLS files, and to provide a means of creating CHAIN and CHAINPLS files during the execution of CHAINPLS in compile only mode. These directives disable certain fuctions of the CHAINPLS input scanner causing data which would normally be modified or interpreted as control statements to be ignored or modified only slightly.

2.3.25.1 The Compile Phase Comment Directive

Any record beginning with a period (".") is considered to be a compile phase comment. If CHAINPLS is being run in compile-only mode, these records are passed directly to the output file; otherwise the are displayed on the system console and not passed to the SYSJOB/SYS file. Any indicated replacements will be performed on a comment line before it is written or displayed.

2.3.25.2 The Transparent Replacement Directive

The transparent replacement directive is designed to allow a user to build normal CHAIN directives as output from CHAINPLS. The format of the transparent replacement directive is:

```
//% <any statement>
```

The transparent replacement statement causes all data items entered in the statement to be replaced by their associated values, the "//%" is replaced by the normal "//", but otherwise the statement is not compiled, but passed to the output file. Note that if CHAINPLS is <u>not</u> in compile only mode that when this

statement is encountered in the SYSJOB/SYS file it will cause an error.

2.3.25.3 The Total Transparency Directive

The total transparency directive is used to force statements to pass through CHAINPLS completely unmodified. The format of the of the total transparency directive is:

```
//$ <any statement>
```

CHAINPLS will convert the "//$" to "//" but no other alterations of any kind will be performed.

2.3.25.4 The Programmer Comment Directive

The programmer comment directive allows the programmer of a CHAINPLS file to comment the source code without affecting the execution of the program. This directive is completely ignored and is never printed or displayed during compilation or execution. The format of the programmer comment directive is:

```
& <any statement or data>
```

2.4 The Pre-defined Data Items in CHAINPLS

In CHAINPLS, certain words are reserved for use by the processing program and may not be redefined by the user. These "keywords" are generally used to indicate the status of the machine at execution time. Some of the keywords return just a Boolean value, and others return a Boolean and a data value.

KEYWORD	BOOLEAN	SECONDARY VALUE
ARC	TRUE=ARC running	none
ARCDATE	TRUE=ARC running	if TRUE, date as YY/MM/DD
ABTFLAG	TRUE=ABTIF flag was on at BOJ	
ARCTIME	TRUE=ARC running	if TRUE, time as HH:MM
BOOTDRV	always TRUE	numeric; DOS booted drive #
CHAINACT	TRUE=chaining already active (recursion)	
CHNCMDFL	always TRUE	char. name of the chain file being compiled.
DOSLEVEL	always TRUE	numeric; usually 2
DOSREV	always TRUE	numeric; usually 4
DOSLETTR	always TRUE	char, DOS. letter
FALSE	always FALSE	none
IS1800	TRUE=processor is 1800	none
IS6600	TRUE=processor is 6600	none
MEMSIZE	always TRUE	numeric; memory size in K as returned by DOSFUN 10.
NO	always FALSE	none
NULL	always TRUE	null character string
OFF	always FALSE	none
ON	always TRUE	none
PRTAVAIL	TRUE=line printer avail	none
PS	TRUE=PS running	none
QQUOTE	always TRUE	char = double quote (")
SUR0	TRUE=drive 0 on-line	char=subdirectory name
SUR1	TRUE=drive 1 on-line	char=subdirectory name
(SUR2 - SUR31 defined identically)		
TRUE	always TRUE	none
VOLID0	TRUE=drive 0 on-line	char=volume name if true
VOLID1	TRUE=drive 1 on-line	char=volume name if true
(VOLID2 - VOLID31 defined identically)		
YES	always TRUE	none

2.5 Special Replacement in CHAINPLS

The standard CHAIN replacement symbols #<data item name># are the primary means of data replacement in CHAINPLS. However, the results of the replacement are dependent upon the Boolean and associated data types of the <data item name>. If the Boolean value of a data item is FALSE, the replacement is NOT done. If the Boolean value is TRUE but the data value is null, the symbol is replaced with a zero length string and disappears from the statement. If the Boolean value is TRUE and the data value is not null, the replacement action taken depends upon the data type

associated with the data item.

If the data type is CHARACTER, the replacement is made by inserting the ASCII characters of the current value of the character string into the statement. If the data type is DECIMAL or OCTAL, the internal binary value of the data item is converted to ASCII decimal or octal representation. If the data item is numeric and its value is less than zero, a minus sign (-) will precede the digits of the value. The value type of an item is changed only by use of the "ASSIGN A=B" form of the ASSIGN statement.

Special provision has been made to allow any type of insertion allowed by the program to be directed.

2.5.1 Decimal Replacement

To force a symbol to be replaced by the decimal value of the data item, the format is:

```
$<data item name>$
```

For example:

```
// ASSIGN A=0101
// ASSIGN B=$A$
```

would result in B having a boolean value of TRUE and a data value of 65.

2.5.2 Octal Replacement

To force a symbol to be replaced by the octal value of the data item, the format is:

```
%<data item name>%
```

For example:

```
// ASSIGN A=65
// ASSIGN B=%A%
```

would result in B having a boolean value of TRUE and a data value of 0101.

2.5.3 Character String Replacement

To force a symbol to be replaced by the character value of the data item, the format is:

```
@<data item name>@
```

If the data type of the variable is character this option forces the exact **first three bytes** of a variable into the output line. If the variable is NUMERIC, the exact binary values are used. This can result in the insertion of unusable or unrecognizable characters into an output stream. Consequently, this option should not normally be used except when such binary output is desired.

For example:

```
// ASSIGN N=65
// ASSIGN B="@N@"
```

would result in B having a boolean value of TRUE and a data type of char and value of "A".

CHAPTER 3. EXECUTION PHASE DIRECTIVES OF CHAINPLS

The execution phase directives are statements which are processed by CHAINPLS and passed directly to the output file. They take effect only during actual execution of the command statements from a SYSJOB/SYS file. They are allowed only between program executions (job steps).

3.1 Execution Phase Comment Directive

The format of the execution phase comment is:

```
//.  <any comment>
```

This statement, when encountered in the SYSJOB/SYS file, will be displayed exactly as written onto the system console. Execution will continue without pause.

3.2 Operator Pause Directive

The format of the operator pause directive is:

```
//* <any comment>
```

This statement, when encountered in the SYSJOB/SYS file, will be displayed as is on the system console. Processing will then be suspended pending the depressing of the display key by the system operator.

3.3 Conditional Abort Directive

The format of the conditional abort directive is:

```
// ABTIF <string>
```

This statement, when encountered in the SYSJOB/SYS file, will test the "abort" bit in the DOS system flag byte. If the bit is on, the entire execution will terminate with an error message. Most Datapoint software will set the "abort" bit on if serious errors were encountered during their execution. The user should utilize the DOS "ABTONOFF" utility to guarantee the status of the abort bit to avoid spurious errors.

The string can be any sequence of characters. It is treated as a comment by CHAINPLS. However, it must not contain any CHAINPLS reserved words.

3.4 Log Suspension Directive

The format of the log suspension directive is:

```
// LOGOFF
```

This statement will suspend logging until a log restart directive is encountered. This allows a user to selectively log only parts of a CHAINPLS execution. This feature can be useful when running utilites which display large amounts of non-critical data or which are virtually always error-free.

This statement has no effect if logging is not active or has already been suspended.

3.5 Log Restart Directive

The format of the log restart directive is:

```
// LOGON
```

This statement will restart logging after a temporary suspension. It has no effect if logging is not active or has not been suspended.

3.6 Date/Time Stamp Directive

The format for the date/time stamp directive is:

```
// STAMP
```

The date/time stamp directive is used to display the current calendar date and time on the system console. It has no effect if the Attached Resource Computer system is not being used or if no File Processor possesses an available ARCCLOCK/TXT file. The message displayed on the console screen appears as:

```
STAMP: yy/mm/dd hh:mm
```

For example:

STAMP: 79/03/21 10:04

is March 21, 1979, 10:04 AM. If logging is active, the date/time stamp record will be written to the system log file.

3.7 The Operator SIGNAL Directive

The format of the operator signal directive is:

// SIGNAL <noise time> <wait time> <noise time> <wait time>...

The SIGNAL directive causes the processor click to sound rapidly, producing a distinctive tone. The numbers which occur in the statement are considered in pairs: the first number specifies the number of seconds to produce the noise, the second number specifies the number of seconds of silence. For example:

// SIGNAL 5 30 10 30 1 1 1 1

will produce the following:

5 seconds of noise
30 seconds of silence
10 seconds of noise
30 seconds of silence
1 second of noise
1 second of silence
1 second of noise
1 second of silence

The use of the keyboard or display keys on the processor will cause the execution of the SIGNAL statement to terminate following the current time segment. Note that any nonnumeric data beyond the keyword SIGNAL is ignored.

CHAPTER 4. BASIC USAGE

4.1 Symbol Replacement

It is often necessary to execute a particular set of programs with only minor modifications from execution to execution. As with CHAIN, CHAINPLS may be used to specify such variables as date and time of execution or print disposition in a single keyin line. For example, consider this "compile-only" CHAINPLS command line:

```
CHAINPLS INCHAIN,OUTCHAIN;OPTIONS=C,DATE=25FEB78,TIME=20:23
```

This command will cause CHAINPLS to read a file called INCHAIN/TXT and create a file call OUTCHAIN/CHN. If INCHAIN/TXT consisted of these lines:

```
DBCMP TESTPROG;C
TEST PROGRAM COMPILATION #DATE# #TIME#
```

the output file OUTCHAIN/CHN would appear as follows:

```
DBCMP TESTPROG;C
TEST PROGRAM COMPILATION 25FEB78 20:23
```

When OUTCHAIN/CHN is CHAINed, the resulting compiler listing will have the correct date on the heading.

If a data item is specified on the command line but no value is assigned to it, all references to the item disappear from the generated output. For example, if the command line in the example above had been:

```
CHAINPLS INCHAIN,OUTCHAIN;OP=C,DATE=25FEB78,TIME=20:23,PRINT
```

and the file INCHAIN/TXT had appeared as:

```
DBCMP TESTPROG;#PRINT#
TEST PROGRAM COMPILATION #DATE# #TIME#
```

the output file would appear:

```
DBCMP TESTPROG;
TEST PROGRAM COMPILATION 25FEB78 20:23
```

In other words, the data item would have been replaced with nothing.

4.2 Statement Selection

4.2.1 The IF/ELSE/XIF Statement

CHAINPLS also allows a user to determine if a data item was specified on the command line and include or exclude statements from the output file based on the result. The primary means of selecting statements to be included is the IF statement. Consider the command line:

```
CHAINPLS INFILE,OUTFILE;OP=C,DAILY
```

The IF statement could be used as follows:

```
// IF DAILY     (if DAILY is TRUE)
statement       (statments to be included if DAILY)
statement
// ELSE         (otherwise, if DAILY is FALSE)
statement       (statements to be included if not DAILY)
statement
// XIF          (end of conditional selection)
```

IF statements may be used inside the range of other IF statments to further control the selection of lines for the output file.

4.2.2 The WHILE Statement

The WHILE statement is similar to the IF statement in that a conditional expression controls inclusion of a set of statements; however the WHILE statement allows for a set of statements to be included repeatedly as long as the control expression is TRUE. For example, if the command line entered were:

```
CHAINPLS INFILE,OUTFILE;OP=C,DAY=12
```

and the input file were:

```
// WHILE DAY > 5          (statment to evaluate each time)
LISTING FOR DAY #DAY#     (statements included)
// ASSIGN DAY=DAY-1       (assign a new value to DAY)
// END                    (end of group; go back to // WHILE)
```

the output file would be:

```
LISTING FOR DAY 12
LISTING FOR DAY 11
LISTING FOR DAY 10
LISTING FOR DAY 9
LISTING FOR DAY 8
LISTING FOR DAY 7
LISTING FOR DAY 6
```

In other words, CHAINPLS will go back to the WHILE statement and re-evaluate it over and over again until it is FALSE. Any statements within the WHILE block will be included as many times as the statement is evaluated and found to be TRUE. The use of WHILE should be carefully tested. An erroneous WHILE statement could be evaluated as always TRUE and thus cause an infinite loop.

4.2.3 The DO Statement

The DO and UNTIL statements are very similar to the WHILE and END statements. The basic difference is that the DO loop is always executed at least once. If this condition is fulfilled, the DO/UNTIL syntax is generally faster than WHILE/END due to the fact that exit is made from the bottom of the loop; that is, when a WHILE statement is found to be false all statements up to and including the END must be read by CHAINPLS but are effectively ignored.

For example, consider a KEYIN loop to enter a valid option:

```
// ASSIGN GOODIE=""
// DO
// KEYIN "Enter I,N,X,Z for run option: " GOODIE
// UNTIL (.GOODIE=1)&("INXZ"[GOODIE)
```

The UNTIL expression says "repeat the loop until the length of GOODIE is 1 byte and GOODIE is an "I", "N", "X", or "Z". If the key operator enters a null string, the test will fail and reentry will be requested; too long a string will cause the same action.

This is a simple yet powerful method for data entry control.

4.3 Data Item Value Alteration

4.3.1 The ASSIGN Statement

Since a data item in CHAINPLS has both a Boolean and an associated data value, two statements are used to control these values: a SET statement for the Boolean value and an ASSIGN statement for the data value. For example, consider the following command line:

```
CHAINPLS INFILE;DAILY
```

If the file INFILE were to contain:

```
// IF DAILY                (if DAILY is true)
// ASSIGN PRNTOPT=";L"     (assign a literal to PRNTOPT)
// ELSE                    (otherwise)
// ASSIGN PRNTOPT=";P"     (assign different value to PRNTOPT)
// XIF
```

Notice that any data item that is used as the receiving field of an ASSIGNment statement is automatically set to TRUE; its value, however, may be NULL.

The expressions used in an ASSIGN statement can be very complex. For example, consider these statements:

```
// ASSIGN NUM1=350
// ASSIGN NUM2=600
// ASSIGN RESULT= (NUM2-NUM1)*3
```

The result of these statements would be to assign the value of 750 to RESULT. Any type of operator can be used in an ASSIGNment statement, but care must be taken that the results of the operation are understood by the user.

4.3.2 The SET Statement

The SET statement is used to control the Boolean value of a data item. Any data item that is used as the receiving field in a set statement has its associated value set to null. It is important to note that if the Boolean value of a data item is FALSE, no replacement will be done on any occurrences of that data item in an input line.

Consider the following example:

```
// SET RESULT=DAILY ¦ WEEKLY
```

If either DAILY or WEEKLY were TRUE, RESULT would be set to TRUE; otherwise RESULT would be set to FALSE.

Very complex logical statements can be used in a SET statement, but, again, a user must be careful to understand the results of the use of a particular operator. For example, the statement:

```
// SET RESULT= (DAILY ¦ WEEKLY) & PRINT
```

RESULT would be SET to TRUE if either DAILY or WEEKLY are TRUE and PRINT is TRUE. RESULT would be SET to FALSE if PRINT were FALSE regardless of the values of DAILY and WEEKLY. Note again that any expression which can be used in a SET statement is valid in an IF or WHILE statement.

Note that the spaces shown in the example expressions are not required and are for readability only.

4.4 Basic Usage Example

A common problem with CHAIN files is that the user cannot guarantee that a certain data item was assigned a resonable value. Another problem is that often a user may want to default the value of a data item that is not entered in the command line because the item will nearly always have the same value. In the following example, the DRIVE number cannot be greater than four (4) and the LIST option will be defaulted to if PRINT or LIST is not specified.

```
// IF ~ DRIVE              (if DRIVE is FALSE)
.DRIVE MUST BE SPECIFIED (display a comment)
// ABORT                  (ABORT the run)
// XIF                    (end the IF)
```

```
// IF DRIVE>4              (if DRIVE was greater than 4)
.DRIVE MUST NOT BE > 4     (display a comment)
// ABORT                   (ABORT the run)
// XIF                     (end the IF)
// IF ~(PRINT ¦ LIST)      (if PRINT and LIST are FALSE)
// SET LIST=TRUE           (default to LIST option)
// XIF                     (end the IF)
....statements to be included in output file
```

CHAPTER 5. INTERACTIVE USAGE

5.1 The Use of KEYIN with WHILE and DO

A major problem assoicated with CHAIN files is the inability to correct simple errors in keying the instructions to the CHAIN utility. The only facility possessed by CHAIN is the ABORT statement which forces the user to restart all CHAIN activity. In CHAINPLS, the KEYIN statement can be used to interact with the user to correct errors. In fact, CHAINPLS files can be written in such a manner that they need no command line specifications at all; questions are asked by KEYIN statements, the answers are validated, and the execution continues.

A major tool in validation of KEYIN responses is the WHILE statement. The basic approach is to enclose a KEYIN statement in a WHILE block and not discontinue execution of the block (set the expression to FALSE) until a correct answer has been given to the question. A data item is declared and SET to FALSE; a correct entry to the KEYIN statement causes the data item to be SET to TRUE. For example:

```
// SET VALID=FALSE          (declare a FALSE data item)
// WHILE ~ VALID            (do until VALID is TRUE)
// KEYIN "ENTER PRINTOUT OPTIONS: " POPTION
                 (display message and ask for a response)
// IF (POPTION="LIST") | (POPTION="PRINT")
                 (if the keyin was "LIST" or "PRINT")
// SET VALID=TRUE           (set VALID to TRUE)
// XIF
// END                      (go back and see if VALID is TRUE)
```

In this example, the question "ENTER PRINTOUT OPTIONS:" would appear repeatedly until the console operator had entered either "LIST" or "PRINT"; at that time the execution would continue with the statement beyond the "// END".

5.2 Displaying Data Items With KEYIN

Any data item may be displayed with the KEYIN statement by including the item as a replacement item within a literal. For example:

```
// KEYIN "DATE IS #DATE#.  CORRECT? " YESNO
```

If DATE was entered on the command line as "03/17/79", the line would appear on the display screen as:

```
DATE IS 03/17/79.  CORRECT?
```

Note that the KEYIN statement may be composed of all literals; no responses are necessary to a KEYIN statement that does not contain data item names. Consider the following example:

```
// IF ~ DATE                (if DATE was not specified)
// KEYIN "DATE NOT SPECIFIED. ENTER DATE: "  DATE
                 (ask the operator for the date)
// XIF
// SET DATEOK=FALSE
// WHILE ~ DATEOK           (while DATEOK is FALSE)
// KEYIN "DATE IS #DATE#.  CORRECT? " YESNO
                 (ask the operator if DATE correct)
// IF (YESNO="Y")|(YESNO="YES")
                 (if the response was "Y" or "YES")
// SET DATEOK=TRUE          (date is correct)
// ELSE                     (they want to correct the date)
// KEYIN "ENTER CORRECT DATE: " DATE
                 (re-enter the date)
// XIF
// END                      (back to the WHILE if DATEOK FALSE)
// KEYIN "THE DATE IS #DATE#"
```

Note that in the final KEYIN statement, there are no data items specified. This KEYIN will merely display the entered and approved date on the console.

Many more sophisticated routines could be designed, including such things as table validation of options, "help" options to tell the operator of possible choices, conversion of dates and times from one format to another, and so on. A good general approach would be to design the CHAINPLS file to run without any operator intervention if the necessary options are specified on the command line and to ask for options if not. This allows the more sophisticated user to completely specify a CHAINPLS execution, while a novice can still correctly run the file and be informed of

all options and have all responses validated.

The DO statement can be used in place of the WHILE statement in most applications. Refer to the example in the chapter on BASIC USAGE.

CHAPTER 6. CHARACTER STRING MANIPULATION

6.1 The Basic Operations

CHAINPLS allows various types of expressions: logical, arithmetic, and character string. To most users, the character string operations will be the most unfamiliar, and yet, after exposure, could prove to be the most useful.

The basic character string operations are concatenation and sub-stringing. Concatenation merely means appending one character string onto the end of another to create a new character string. Sub-stringing is the extraction of part of an existing character string to create a new string.

6.1.1 Concatenation

The form of the concatenation operation is

<input string 1> \\ <input string 2>

The output of this operation is a string consisting of <input string 1> followed by <input string 2>. Consider the following example of concatenation:

```
// ASSIGN CHAR1="TODAY IS "
// ASSIGN CHAR2="FEBRUARY 28, 1979"
// ASSIGN NEWDATE= CHAR1 \\ CHAR2
```

The result string NEWDATE would be:

```
TODAY IS FEBRUARY 28, 1979
```

6.1.2 Sub-stringing

6.1.2.1 Definition of Sub-stringing

The sub-string operation is, in a way, the reverse of the concatenation operation. Instead of combining two strings, a piece of an existing string becomes a new string. The simple form of a sub-string operation is:

<old string> ^ <control expression>

6.1.2.2 Simple Sub-stringing

If the <control expression> is a single number, that number is used as an index into the <old string> and the pointed character is extracted from the old string. For example:

```
// ASSIGN OPTION="LBFTQP"
// ASSIGN CHOICE= OPTION ^ 3
```

The result string CHOICE would be the single letter "F".

6.1.2.3 Sub-string Control Expressions

The more general form of the sub-string operation allows for specification of the length of the result string. This form is:

<old string> ^ (<start expression> : <length expression>)

The result of this type of sub-string operation is the string of characters in the <old string> starting with the character pointed to by the <start expression> and continuing for <length expression> characters. For example:

```
// ASSIGN MONTHS="JANFEBMARAPRMAYJUNJULAUGSEP"
// ASSIGN THISMON = MONTHS ^ (7:3)
```

The value of THISMON after the operation would be "MAR"; that is, starting with the seventh character and taking three. Note that the items used in a sub-string operation can be expressions. For example, here is a conversion routine from standard "mm/dd/yy" format to the format required by the MOUT utility:

```
// ASSIGN TODAY="10/17/78"
// ASSIGN MONTHS="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
// ASSIGN MM=TODAY^(1:2)
// ASSIGN NUMMON=#MM#
// ASSIGN ALPHMON=MONTHS^(NUMMON*3-2 : 3)
```

```
// ASSIGN MOUTDATE=(TODAY^(4:2)) \\ ALPHMON \\ (TODAY^(7:2))
```

In the third line of the example above, the value "10" is assigned to the data item MM. In the fourth line, the value of MM is substituted into the line so that a numeric variable containing a value of 10 is created. In the fifth line, the value of NUMMON (now = 10) is used as part of an arithmetic expression which computes the starting address of the substring operation. The result of the sub-string operation is the value "OCT", which is the alphabetic equivalent of the 10th month. In the sixth line of the example the day of the month is extracted from TODAY, the ALPHMON is appended onto the back of the day of the month, and the year is extracted from TODAY and appended onto the previous result. The value assigned to MOUTDATE would be "17OCT78".

6.1.2.4 Circularization of Strings

Character strings in CHAINPLS are considered to be "circular"; that is, the last character of the string is considered to be logically adjacent to the first character of the string. If a "length expression" greater than the length of the string is specified in a control expression, the string characters are repeated until the desired length is satisfied. For example:

```
// ASSIGN ONETEN="12345267890"
// ASSIGN FIFTEEN=ONETEN^(1:15)
```

The resulting data item FIFTEEN would have a value of "123456789012345".

6.1.2.5 Negative Numbers in Sub-stringing

In the above description of a substring "control expression", only positive numbers are used as "start expressions" and "length expressions". CHAINPLS also allows these values to be negative (i.e., less than zero) and assigns a special meaning to such values.

In CHAINPLS, the last character of a string is considered to be position "-1". By this example, the next to the last position would be "-2". Using the data item ONETEN defined in the above example:

```
// ASSIGN ENDTEN=ONETEN^(-3:3)
```

The result field ENDTEN would have a value of "890", since "-3"

implies starting the operation with the third character from the end of the string.

If negative numbers are used in the length expression, they are interpreted as meaning a "backwards" direction for the sub-string operation. For example:

```
// ASSIGN ENDBACK=ONETEN^(-3:-3)
```

The result would be "876". This facility allows strings to be inverted easily.

Note that sub-stringing in either the forward (positive length) direction or backward (negative length) direction which passes the end or the beginning of the string merely causes the operation to continue at the opposite end of the string. For example:

```
// ASSIGN DUMMY=ONETEN^(-3:-15)
```

The result field DUMMY will have a value of "876543210987654". This result starts at the third character from the end of the field and continues backwards until 15 characters have be extracted.

6.2 The More Complex Operations

6.2.1 The "Length Of" Function

It is often important in character string manipulation to know the length of a string. This is especially useful in sub-stringing when the output string is to be the length of the input string starting at some intermediate position through the rest of the string. The "length of" function always returns a numeric value equal to the length of the string. For example:

```
// ASSIGN NUMBERS="1234567890"
// ASSIGN LENGTH= .NUMBERS
```

The value of LENGTH after the second ASSIGN statement is 10. The "length of" function can be used to obtain the length of character string expressions also. For example:

```
// ASSIGN LENGTH= .(NUMBERS \\ NUMBERS)
```

The value of LENGTH after the above statement would be 20, since it represents the length of NUMBERS concatenated with itself. Note that the value of NUMBERS is NOT changed by this statement.

6.2.2 The Pattern Match Operation

The "pattern match" operation finds the first location (if any) of a small string in a large one. The general form of the "pattern match" operation is:

<operand string> [<search string>

If the <search string> is found in the <operand string> the value of the pattern match expression is the location (relative to 1) of the search string in the operand string. If the search string is not found in the operand string the result of the expression is zero. Additionally, if the pattern match operation is used as a Boolean (only for TRUE/FALSE purposes), the expression will be TRUE if the search string is found, and FALSE if it is not. For example:

```
// ASSIGN TABLE="JANFEBMARAPRMAYJUNJULAUGSEPOCTNOVDEC"
// ASSIGN MONTH="AUG"
// SET FOUND= TABLE [ MONTH
// ASSIGN POSITION= TABLE [ MONTH
```

The data item FOUND will be SET to TRUE, and POSITION will be ASSIGNed the value 21. If MONTH had not been found in TABLE, FOUND would have been false and POSITION would have been zero.

The "pattern match" operation is so named because it can also be used to find a particular pattern of characters in an operand string. The special character "?" is used as a "don't care" character; that is, all characters will always compare as equal to "?" during a pattern match scan. For example:

```
// ASSIGN TABLE="A32ZB00MC256D77PE999"
// ASSIGN POSITION= TABLE [ "B??M"
```

The value of POSITION after the ASSIGN statement would be 5. The pattern match instruction can be used to search tables and extract values. For example:

```
// ASSIGN VALIDMON="01JAN02FEB03MAR04APR05MAY06JUN"
// ASSIGN DATE="04/17/78"
// ASSIGN MONTH= DATE ^ (1:2)
// ASSIGN POSITION= VALIDMON [ MONTH
```

```
// ASSIGN ALPHMON= VALIDMON^(POSITION+2:3)
```

The routine first extracts the sub-string MONTH from the string DATE. Then the position of MONTH in VALIDMON is found. Then that position is used to extract the sub-string from VALIDMON containing the alphabetic name of the current month. Note that since character string expressions are allowed, the above example could have been coded as:

```
// ASSIGN VALIDMON="01JAN02FEB03MAR04APR05MAY06JUN"
// ASSIGN DATE="04/17/78"
// ASSIGN ALPHMON= VALIDMON^((VALIDMON [ (DATE^(1:2)))+2:3)
```

It is easy to see that character string expressions can get very complex and unreadable. There is also the danger that, for example, the first two characters of the string DATE will not be found in VALIDMON and the resulting answer to the expression would be meaningless.

6.2.3 The Scan Operation

The scan operation is used to find the first occurrence in an operand string of ANY of the characters in a search string. The general form of the scan operation is:

```
<operand string> [[ <search string>
```

The result of the scan operation is similar to the pattern match operation: if the scan is successful the result is TRUE and equal to the position of the first matching character; if unsuccessful, it is FALSE and will be interpreted as zero. For example:

```
// ASSIGN FILESPEC="WORKFILE/TXT:DR5"
// ASSIGN POSITION= FILESPEC [[ "/: "
// ASSIGN FILENAME= FILESPEC ^ (1:(POSITION-1))
```

The second ASSIGN statement in the example will return the numeric position of the first slash (/), colon (:), or blank in the string FILESPEC. The third ASSIGN statement creates a new string containing all the characters in FILESPEC up to but not including the slash (/).

6.2.4 The Scan Unequal Operation

The scan unequal operation is very similar to the scan operation with the exception that the scan unequal is used to find the first occurrence in an operand string of any character that is not in the search string. The general form of the scan unequal operation is:

```
<operand string> ~[[ <search string>
```

The value returned by the scan unequal operation is the position in the operand string of the first character which does not match any of the characters in the search string.

6.2.5 The String Replace Operation

The string replace operation allows selected characters in a string to be replaced by other characters. The general format of the string replace operation is:

```
<operand string> \ <replacement string>
```

The value returned by the string replace operation is a character string whose length is the same as the operand string; its contents are the same as the operand string except for those characters whose replacements occur in the replacement string.

The replacement string is interpreted as a (possibly null) series of character pairs. The first character of each pair specifies the character to be replaced; the second specifies the character to be inserted into its place. For example:

```
// ASSIGN REPL="0A1B2C"

// ASSIGN SOURCE="0123456789"

// ASSIGN RESULT=SOURCE \ REPL
```

The string RESULT would contain "ABC3456789". This feature can be useful in eliminating such problem characters as double quotes (") from keyin input:

```
// ASSIGN FIXER=QQUOTE \\ "'"

// KEYIN "Enter error response:" RESPONSE

// ASSIGN RESPONSE= RESPONSE \ FIXER
```

The string FIXER is created containing a double quote and a single quote; this string is then used to replace any double quotes in the RESPONSE string with single quotes.

CHAPTER 7. COMPLEX EXPRESSIONS

In CHAINPLS, the basic operation types can be intermixed in almost any manner to produce results desired by the user. In some cases, certain defaults must be assumed and definitions used that may cause confusion if not clearly understood. There are many adavantages, however, in the use of mixed expressions.

7.1 Expression Evaluation

CHAINPLS does not use operator precedence; that is, no operation is always performed before any other. If the user desires that operations be performed in a particular manner, parentheses can be used to any depth to indicate the order desired. The expression evaluation takes place from left to right. For example:

```
// ASSIGN RESULT=ADD1+ADD2*MULT1
```

The expression evaluation would take place as follows: ADD1 would be added to ADD2 and the result would be multiplied by MULT1. Parentheses could be used to force normal FORTRAN-type evaluation:

```
// ASSIGN RESULT= ADD1+(ADD2*MULT1)
```

7.2 Intermediate Results of Expressions

In the evaluation of any expression more complex than "A=B" it becomes necessary to execute the requested operations in a step-by-step manner. This means that intermediate results are created at one level of evaluation, utilized at the next, and discarded. These intermediate results can safely be ignored if the expression being evaluated consists completely of the same class of operators; in hybrid expressions, however, it is important to note the class and use of these intermediate results.

7.2.1 Boolean Used as Arithmetic

The basic operation types are Boolean, arithmetic, and character string. All of the arithmetic and binary operators generate an arithmetic result. The logical operators generate Boolean results, that is, the result of the evaluation of a logical operation is always either TRUE or FALSE. The character string operations generate either character results, arithmetic results, or Boolean results, depending upon the particular operation. If an arithmetic operation is performed upon a Boolean value or result, the Boolean is interpreted as being zero if FALSE and one if TRUE. For example:

```
// ASSIGN FIVE=5
// ASSIGN THREE=3
// ASSIGN ANSWER1=(FIVE>THREE)*6+9
// ASSIGN ANSWER2=(FIVE<THREE)*6+9
```

The result of evaluating "FIVE>THREE" is a Boolean value of TRUE. In the subsequent multiplication, the value one (1) is used as the arithmetic equivalent of TRUE; the result of the expression is 15. The evaluation of "FIVE<THREE" results in FALSE, and the value zero (0) is used as its arithmetic equivalent. The value assigned to ANSWER2 is 9.

If a number or intermediate arithmetic result is used in a Boolean expression, the result is always TRUE. For example:

```
// WHILE 7
```

This expression is equivalent to "WHILE TRUE" and thus will be executed indefinitely.

7.2.2 Numeric Results from Character Operations

In character string manipulation, it is common for results of intermediate evaluations to vary between arithmetic and character type at different stages in the evaluation. Here is a list of the character string operators and the types of results they produce:

```
\\    concatenation; character string
^     sub-string; character string
:     sub-string control; numeric
.     string length function; numeric
[     pattern match; numeric
          (also FALSE/TRUE if used as Boolean)
[[    string scan; numeric
          (also FALSE/TRUE if used as Boolean)
~[[   string scan unequal; numeric
          (also FALSE/TRUE if used as Boolean)
\     string replacement; character string
```

It is common in table-type manipulation to match a small string to a large one and use the position of the small string in the large one as part of a sub-string control expression. Consider the example given in the chapter on character string manipulation:

```
// ASSIGN VALIDMON="01JAN02FEB03MAR04APR05MAY06JUN"
// ASSIGN DATE="04/17/78"
// ASSIGN ALPHMON= VALIDMON^((VALIDMON [ (DATE^(1:2)))+2:3)
```

The innermost expression in the above statement is the "(1:2)" sub-string control expression. The order of evaluation of this statement is:

```
(1:2)                creates arithmetic result N1
DATE^N1              creates character result C1
VALIDMON [ C1        creates arithmetic result N2
N2+2                 creates arithmetic result N3
(N3:3)               creates arithmetic result N4
VALIDMON^N4          creates character result C2
ASSIGN ALPHMON=C2    transfers C2 to ALPHMON
```

7.2.3 Character Results Used in Numeric Operations

If a character result is used in an arithmetic operation, the binary value of the first three characters (right-justified) of the character string are used as the arithmetic value. Obviously, this could lead to unpredictable results and is not advised.

7.2.4 Numeric Values Used in Character Operations

If a data item with a data type of NUMERIC or OCTAL is used in a character string operation, the value is first converted by type into a character string. For example:

```
// ASSIGN NUM1=230
// ASSIGN OCT1=0377
// ASSIGN CHARESLT=NUM1\\OCT1
```

The value assigned to CHARESLT would be "2300377", which is the result of concatenation of the character equivalents of NUM1 and OCT1.

Intermediate arithmetic results can also be used in character string operations. Since all intermediate arithmetic results are implicitly considered to be NUMERIC (i.e., decimal), the intermediate result is converted into a character string and utilized in the operation. Consider:

```
// ASSIGN CHARESLT= "VALUE:"\\(150/5)
```

The value assigned to CHARESLT would be "VALUE:30". Note that without the parentheses the result of the expression would be meaningless, since the character string result of concatenating "VALUE:" with "150" would then be divided by 5, resulting in garbage.

7.3 Comparison Evaluation

In any comparison, an examination of the operands in the comparison is made and adjustments are made for value type. If both of the operands in a comparison are of type NUMERIC or OCTAL, a comparison the binary equivalent of their values is made. If either of the operands are of type character, the NUMERIC or OCTAL operands are converted to character strings and the comparisons are made on a character-by-character basis. For example:

```
// ASSIGN NUM1=255
// ASSIGN OCT1=0377
// SET VALUE1= (NUM1=OCT1)
// SET VALUE2= (NUM1\\" ") = (OCT1\\" ")
```

In the first SET statement above, the binary equivalents of 255 and 0377 are equal and therefore VALUE1 would be SET to TRUE. In the second SET statement a space (" ") is concatenated on the end of the values NUM1 and OCT1. This forces them to be converted to

character strings and the comparison is made on the resulting intermediate character string values; the results of comparing "255 " to "0377 " is that they are not equal, thus VALUE2 is SET to FALSE.

In general, if a value is a pure Boolean it is of type CHARACTER with an associated value of NULL. This is not always true, consequently doing comparisons on pure Booleans or intermediate Boolean evaluations may generate unpredictable results.

In comparing strings, CHAINPLS does a left-to-right character match. If one string is shorter than another, the shorter is padded with trailing spaces. A null (zero length) string is considered to be "less" than anything but another null string; in such cases they are considered equal.

7.4 Array Simulation and Processing

One of the more esoteric features of CHAINPLS is the ability to create dynamic variable names. This is done by doing string replacement in the middle of a data item name. For example:

```
// ASSIGN INDEX=1
// ASSIGN LIMIT=10
// DO
// ASSIGN VAR#INDEX#=""
// ASSIGN INDEX=INDEX+1
// UNTIL INDEX>LIMIT
```

The above example creates a series of 10 variables named "VAR1" through "VAR10". The fourth line of the example forces the value for the symbol "INDEX" to be inserted into the line in such manner that the value of INDEX becomes part of the data item name. This causes no problem since CHAINPLS always does symbol replacement prior to line scanning and parsing.

Considering the above example, it is easy to see that INDEX is being used in the same manner as a subscript would be used in FORTRAN or COBOL. The fact that the data items thus created are not really part of the same data structure is not really a limitation, since this would only affect the ability to move the entire array by referencing a single name. Double subscripting could be done similarly. For example:

```
// ASSIGN SUB1=1
// DO
```

```
//  ASSIGN SUB2=1
//  DO
//    ASSIGN VAR#SUB1#X#SUB2#="data for variable"
//    ASSIGN SUB2=SUB2+1
//  UNTIL SUB2>10
// ASSIGN SUB1=SUB1+1
// UNTIL SUB1>10
```

This example creates a 10 by 10 array of items named "VAR2X4", "VAR10X5", etc.

7.5 Multiple Replacement

CHAINPLS allows the user to perform multiple "recursive" replacement into statements. This allows the user to fully utilize arrays and other associated data items. Consider the data items "VAR1" through "VAR10" created in the first array example above:

```
// ASSIGN INDEX=1
// ASSIGN LIM=10
// DO
KILL #VAR#INDEX##
Y
// ASSIGN INDEX=INDEX+1
// UNTIL INDEX>LIM
```

The "KILL" command issued above will be processed twice by the replacement routine. CHAINPLS will note the "#VAR#" name, but will be unable to find such an item in the symbol table. The next item found is "#INDEX#", which does exist. The line would then read, for example, "KILL #VAR3#". Another iteration of the replacement routine causes the value of "VAR3" to be inserted into the line.

CHAPTER 8. DISPLAY AND PRINT OPTIONS

8.1 The General Format

If the "D" option is specified on the command line, the CHAINPLS display driver is enabled; if the "P", "L", or "S" options are specified, the proper printer driver is enabled. All data lines displayed or printed by the display and print drivers of CHAINPLS are formatted in a similar manner. Each line begins with a single character identifying the line type followed by a colon (:) or an asterisk (*). The asterisk implies that the block in which the statement is contained was actually interpreted by CHAINPLS; if the asterisk is absent, the block was bypassed due to an evaluated expression on an IF or WHILE statement being FALSE. The line identifiers are:

I:	input line as read from input file
R:	input line after symbol replacement
O:	output line written to output file
K:	line displayed from KEYIN statement
D:	line displayed from DEBUG statement
E:	error message (always double spaced)

These same line identifiers appear on any requested printer output along with a run-time trace data.

If the display driver is not enabled ("D" option not set), any keyin lines will appear on the screen without the precedeing "K:".

8.2 The Print and Display Options.

If the "L", "S", or "P" options are given on the command line, the "K:" and "O:" lines are always printed. The "I:", "O:" and "R:" lines are printed or displayed if the "I", "O" or "R" options are specified on the command line along with the "D" option. The DEBUG output only appears if the "T" option is specified. Any errors are always displayed on the console. Error lines cause the processor console to emit a "beep" and the screen display is double-spaced. All error messages also appear on any selected print media.

At any time during the compilation, the console "display" key will suspend compilation temporarily and the "keyboard" key will terminate the compilation.

Note that the "P", "L", and "S" options require the presence of the "UTILITY/REL" file on a ready drive.

8.3 The Print Format

Along with the selected data lines, the CHAINPLS printer driver will also print the contents of the block "stack". The block stack is the list of currently active IFs, ELSEs, BEGINs and WHILEs. This data will appear at the far left of the printed page. Each level of the stack is represented by a single character: I=IF, B=BEGIN, E=ELSE, and W=WHILE. The oldest levels are the left-most levels on the listing. In addition, an asterisk will appear to the immediate right of the stack levels if the block was being interpreted; i.e., if the block was not being bypassed due to an earlier FALSE condition. For example:

```
*          // IF FALSE
I          // BEEP
I          // CLICK
I          // ELSE
E*         // KEYIN "THIS IS CORRECT"
E*         // WHILE 1<0
EW         // BEEP
EW         // END
E*         // IF TRUE
EI*        // KEYIN "THIS IS CORRECT"
EI*        // XIF
E*         // XIF
*          ..........etc......
```

CHAPTER 9. THE SPECIAL FUNCTION OPERATIONS

Certain special function operations exist in CHAINPLS to allow a greater variety of statement than would be possible with operator-format statements. The general form of the function operations is:

```
<function name> ( <operand list> )
```

The <operand list> always contains at least one operand. Operands may be data item names, alpha literals or numeric literals. They may not be expressions. Every function call returns a single answer which is of the same form as a normal data item.

9.1 The TTYPE Function

The TTYPE function is used to check the secondary data type of a variable. Problems which result from manipulation of character strings containing only numeric data or from command line entry of data items can be easily resolved with the TTYPE function. The operand of the TTYPE function is any legal data item name or literal; the result is a character string describing the data type of the operand. If the operand is a Boolean FALSE data item the result is Boolean FALSE. The general format is:

```
TTYPE ( <data item name or literal> )
```

If the <data item name> is a character string, the result will be the string "CHARACTER", a decimal operand will result in "DECIMAL", and octal operand will produce "OCTAL", and a user file variable will generate "FILE". For example:

```
// ASSIGN XX="SEPTEMBER 30, 1978"
// ASSIGN YY=1978
// ASSIGN TYPE1= TTYPE(XX)
// ASSIGN TYPE2= TTYPE(YY)
```

TYPE1 would be a character string of value "CHARACTER"; TYPE2 would be a character string of value "DECIMAL".

All functions, including the TTYPE function, may be imbedded into expressions. For example:

```
// IF (TTYPE(XX)~="CHARACTER")¦(TTYPE(YY)~="OCTAL")
```

```
// KEYIN "INVALID DATA ENTRY!"
// XIF
```

9.2 The FFILE Function

The FFILE function is used to determine if a file is available (on-line) and to return the drive number of the drive on which the file is located. The argument of the FFILE function is a character string data item or an alpha literal containing a DOS-standard file specification. The general form of the FFILE function is:

FFILE (<character string data item or literal>)

The argument of the FFILE function must have the general form:

<file name>/<extension>:D<drive number>

or: <file name>/<extension>:<volume name>

The FFILE function may be used in any expression. If the file is not found, the value of the function is FALSE if the expression is Boolean and zero if the expression is arithmetic. If the file is present, the value is TRUE if the expression is Boolean and equal to the drive number of the file if the expression is arithmetic. For example:

```
// ASSIGN FILESPEC="MYSPEC/TXT"
// SET GOTFILE=FFILE(FILESPEC)
```

The value of GOTFILE will be TRUE if the file is available and FALSE if the file is not available.

```
// ASSIGN DRIVENUM= FFILE("MYFILE/TXT")
```

The value of DRIVENUM will be zero if the file is not available and equal to the drive number of the file if available. Note that the FFILE function should always be used in a Boolean expression first to determine if it exists before determination of the drive number. This is because it is impossible to distinguish in an arithmetic usage between the file not being present and the file being on drive zero.

9.3 The MMEMBER Function

The MMEMBER function is used to determine if a certain member exists in a library file. The FFILE function should be used first to guarantee that the library file actually exists. The general form of the MMEMBER function is:

MMEMBER (<library file spec> , <member name>)

Note that the <member name> must be a character string data item or a literal and must have no special characters or imbedded blanks.

The result of the MMEMBER function is Boolean FALSE if the member does not exist, and is TRUE if it does. Additionally, if the result is TRUE and the library is a relocatable program library, the arithmetic value is the size of the relocatable member. For example:

```
// SET GOTMEM= MMEMBER("UTILITY/REL","SERVO")
```

The result of this operation would be TRUE if the member SERVO exists in the file UTILITY/REL, and FALSE if it does not.

9.4 The UNBIT Function

The UNBIT function is used to convert a numeric variable or literal into a character string of "0" and "1" characters according to its binary equivalent. The result of UNBIT will never been less than a single "0" nor greater than a string of 24 "1" characters. The operand of UNBIT should be Boolean TRUE. For example:

```
// ASSIGN NUMBER=255
// ASSIGN BITNUMR= UNBIT(NUMBER)
// KEYIN "bits are #BITNUMR#"
```

The line as displayed on the screen would be:

```
bits are 11111111
```

9.5 The BIT Function

The BIT function is used to convert a string of "0" and "1" characters into its binary equivalent. For example:

```
// ASSIGN CONVERT=BIT("101000000")
```

The value of CONVERT would be 640.

CHAPTER 10. CHAINPLS EXECUTION PHASE

10.1 General Description

At the start of the compilation phase, CHAINPLS attempts to open (or PREPare) the SYSJOB/SYS file on the booted drive. If this attempt fails (for example, due to FILE SPACE FULL), a global PREPare is executed.

After the compilation phase of CHAINPLS, the file SYSJOB/SYS has been created or extended by the inclusion of all statements selected during compilation. The module CHAINPLS/OV1 is then called to actually execute the commands and keyins from the SYSJOB/SYS file. The first record is read from the SYSJOB/SYS file and is placed into the Monitor Communiations Region of the DOS. The job file is then modified to indicate the last record delivered to the DOS as a command line for restart purposes. The resident DOS is modified to convert the KEYIN$ routine to read the proper responses from the job file.

If logging is active, the DSPLY$ routine is modified to call the CHAINPLS/OV2 module to log all console displays into the log file. Since all lines read from the job file are displayed, all command lines and keyin responses are also logged.

As each executed program returns control back to the DOS, the CHAINPLS/OV1 overlay is reloaded automatically and the above procedure is performed again. When the end of the current extent is reached, the job file is examined to see if any prior extents exist. That is, if the current chain was called from a chain, then execution will pick up from the statement following the statement which invoked the currently active CHAINPLS. There is a system of pointers inside the job file which indicates to CHAINPLS/OV1 the proper pickup point for each level of execution.

As each extent (file) within the job file is completed, a message is displayed on the system console:

CHAIN FILE COMPLETED: <file name>/<extension>:<volid>.<member>

When the entire job file has been executed, the message:

CHAINING COMPLETED

appears on the system console and the SYSJOB/SYS file is removed from disk. The DOS is then reloaded and the familiar "READY" message appears.

Note that as each extent (file) within the job file is completed, the disk space which that extent occupied is re-used by any subsequent nested chaining, thus reducing the size of the job file.

One additonal feature can provide increased compilation speed and greater restart capability: if the user BUILDs a SYSJOB/SYS file and delete-protectes it (via CHANGE), CHAINPLS will not KILL the file after the chaining is completed. This allows the user to restart even a completed file. In addition, the PREPare time for the SYSJOB file is saved during subsequent chaining.

10.2 Execution Restart

Serious errors can often cause chaining to be aborted during a program's execution. By use of the ABTIF statement, a user can test the system status after a job step and conditionally abort the chain. CHAINPLS possesses a variety of means to restart the system after a failure. These means should be clearly understood by the user.

10.2.1 Restart From Current Position

If the system operator keys in:

```
CHAINPLS/OV1 (enter)
```

execution will resume with the next sequential record following the last record which appeared on the screen. This method is generally used to skip over erroneous or unnecessary statements in the job file. If the next line in the job file is not a valid command line, CHAINPLS/OV1 will abort again. The restart message may then be reentered and the next record will be used to restart. This may be done as often as is necessary to find a proper restart point. Other techniques exist, however, which allow more versatile restart.

10.2.2 Restart From Last Command

If the system operator keys in:

CHAINPLS/OV1 * (enter)

execution will resume after repositioning the job file back to the last command line delivered to the DOS. This is probably the most commonly used restart. For example, an INDEX function may have failed because the file selected to be indexed was on a pack not currently mounted. The system operator can then mount the pack, wait for it to come ready, and issue the restart. Execution will resume at the INDEX command and system operation will be unimpaired.

10.2.3 Interactive Restart

If the system operator keys in:

CHAINPLS/OV1 ? (enter)

then CHAINPLS will enter a mode of question/answer interaction which allows the system operator to restart the job file from ANY POINT. The interactive restart displays a number of informative messages about the options available and the status of the SYSJOB/SYS file.

After the initial descriptive message, the user will be asked:

START AT THE CURRENT POSITION?

If "YES" (or "Y") is entered, the last command line delivered to DOS will be displayed. As this and each subsequent line are displayed, the user can choose to resume processing at the currently displayed line. The cursor will appear below the current line and the operator can enter a "YES" or just the "enter" key. During this procedure, the entire extent (file) will be displayed starting at the current position and continuing through to the end of the file. If the operator enters a "YES" to the cursor, execution will resume starting with the record just displayed. If a "NO" or just the "enter" key is entered, the next record in the file is displayed.

At the end of the current extent (file), the system console will display:

THIS FILE WAS <name>/<ext>
START AT THE BEGINNING OF THIS FILE?

If the operator answers "YES", the current extent will be redisplayed. If not, the job file will be repositioned to the prior extent. The message:

REPOSITIONING TO PRIOR FILE

will be displayed. If there is an earlier extent, the same procedure as above will be invoked; that is, the program will inquire if it should start at the beginning of the current extent, and a positive answer will cause the entire extent to be displayed one record at a time. If there is no earlier extent, the program will reposition the job file back to the end of the newest (current) extent and the entire procedure will start over.

By using the above procedure, the user can examine the entire job file as often as is desired and a restart point may be chosen from anywhere in the file. Any record in any extent may be chosen as a restart point by entering "YES" to the flashing cursor positioned under the line.

During examination of the contents of an extent, the keyboard and display keys can be used to scan the file quickly. If the display key is depressed records will be read in the forward direction from the SYSJOB/SYS file displayed rapidly on the screen. The message "==FORWARDS==" will appear on the bottom of the screen indicating the direction in which the file is being read. This rapid display will cease when the display key is released.

The keyboard key functions identically but causes the SYSJOB/SYS file to be read and displayed in the reverse direction. The message "==BACKWARDS==" is displayed to indicate the current direction. If the user attempts to backspace beyond the start of the file, the first record of the file will be re-displayed and the processor "beep" will sound.

Use of the keyboard and display keys allows quick but accurate scans of an extent without having to strike the enter key once for every record displayed.

10.2.4 Restart With Override Job File

Since the user can specify the name of the job file to be used instead of SYSJOB/SYS, a serious failure (e.g., one requiring system re-booting) will require reentry of the job file name. The generalized format for the restart command line is, therefore:

CHAINPLS/OV1 (<job file name>) (<*>/<?>)

Note that only the "last command" and "interactive" restarts are available after serious failure. The record of the exact position of the job file is kept in core during execution; the position of the last command is updated on disk between job executions.

10.3 Execution Logging

All system console activity which takes place while logging is active is written into the SYSLOG/SYS file. This file is in standard text file format and may be copied and saved using normal utilities.

It is important to note the effect of logging on nested chaining. If logging is specified at an outer (older) level of chaining, then logging is active and the OPTIONS specifications of any inner (newer) level of chaining are set to cause logging to occur. The only method for suspending logging once specified is through use of the "LOGOFF" and "LOGON" statements.

If an outer level of chaining does not specify logging but an inner level does, only the inner level chain activity is logged. In this case, it is important to note the use of the "Q" option for logging. If, for example, a user wished to build a CHAINPLS file to run three other CHAINPLS files, but only wanted to log the first and third of these into SYSLOG/SYS, the OPTIONS specifications on the first file should specify the "G" option. If, however, the third file also specified the "G" option the log file would be reinitialized and all prior log information would be lost. The function of the "Q" option is to notify CHAINPLS that new log information is to be stored at the end of any existing log file.

Use of the "Q" option when no previous log file exists causes the log file to be created and is identical to the "G" option.

It is also important to note that certain DOS utilities and other software will not function properly if the logging facility

is active. The reason for this is the dependence of these programs on the contents of the LFO disk buffer or the DOS overlay area. If problems are encountered with the use of logging for a particular program, the LOGOFF function can be used to temporarily disable logging and allow these programs to run normally. Logging could then be resumed immediately afterwards.

CHAPTER 11. USER-WRITTEN SUBROUTINES

11.1 General Concept

Within the CHAINPLS compilation phase assembly language code there exist many utility-type service routines. These service routines are used by CHAINPLS to perform the parsing of all directives read in from the input file. A facility has been included into CHAINPLS to allow assembly language level programmers to write their own CHAINPLS directives and avail themselves of these simple yet powerful internal routines.

When a statement is read in from the CHAINPLS input file, a number of actions occur almost automatically. First, the statement is scanned for any "replacement" characters such as "#", "$", or "%". If any are found, the proper replacement is made. Second, the statement is scanned again and if it begins with "//" (control statement) and the current block is active ("true") the statement is broken into "symbols", that is, individual items such as data names, operators, and numeric and alphabetic literals. The results of this scan are placed into an array called the "expression array". Each item in the array is coded to indicate what its original nature was. All data items are looked up in the symbol table. If they are found, the location of the proper entry is placed into the expression array; if they are not found, an entry is created for them and marked as "character string, false". The address of this new entry is then placed into the expression array.

This "expression array" is utilized by all parsing routines within CHAINPLS and greatly simplifies all further processing of the input line. The first symbol in the expression array is the pointer to the "keyword" or directive, such as "IF" or "ASSIGN". Within the symbol table entry for a directive is the address of its parsing routine. After all symbol scanning has been completed, control is passed to the parsing routine. When parsing is complete, the parsing routine simply does an assembly language "RETURN" back to the input read routine to continue execution.

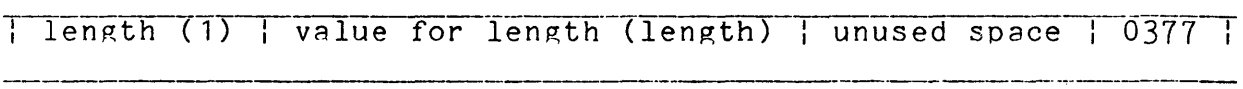
When a user declares a "USERPROG", a new directive is created within the symbol table. There is no difference whatever between the directives which are a permanent part of CHAINPLS and the directives which are created by the USERPROG statement. The newly

created directives can look into the symbol table, do disk I/O and keyin/display, and have access to all significant service routines. It is obvious that a user must be very careful in creating a "USERPROG", because there is no easy debugging tool available for assembly language modules. It is recommended that only those users with significant Datapoint assembly language experience attempt to utilize the "USERPROG" facility.

11.2 Symbol Table Value Structure

All data items used by CHAINPLS are contained in a symbol table. This table has the name of the item, is data type, current status and value, and links to other entries within the symbol table. Any user wishing to create his own directive should be familiar with the data item value formats within the symbol table. It is not necessary for a user to be concerned with the status byte formats and symbol linkage structures of the symbol table because adequate service routines exist to obtain all the information needed.

The basic format for a symbol table value is:



If the item is NUMERIC or OCTAL, the <length> will always be three (3) bytes; all numeric items are kept internally in binary form. If the item is a CHARACTER string, the length can vary from zero (0) to eighty (80). If the item is a FILE variable, the length will be zero (0) and the following 16 bytes will be the current Logical File Table entry for the file.

When a variable first appears in the input stream, it is inserted into the symbol table and given a value area of 14 bytes; the length indicator is set to zero. When a value is ASSIGNed to the variable, the length indicator is adjusted to the new length and the data is inserted into the value area. If the value area is not large enough, a new value area is allocated and linked into the symbol table entry for the variable. The trailer byte (0377) of the value area determines the maximum allowable size of the value. When inserting new values into variables, care must be taken not to overstore the 0377 byte or attempt to store data beyond it.

Care should be taken that all data items are checked for Boolean TRUE before their values are used. A service routine

exists for this purpose. Note that in many cases it is unnecessary to know the value type of a data item since a service routine exists which will move the value string to any location in memory and convert numeric items from binary to decimal or octal character strings as necessary.

It is not necessary for a user-written subroutine to process all items in a statement. No harm will come from failing to access all symbols in a statement. The final position of any of the internal pointers mentioned in the following routines is ignored and reset by the input read routines once control has passed back to the mainline logic of CHAINPLS.

11.3 Internal Service Routines

11.3.1 INCEXP -- Increment the Expression Array Pointer

This routine sets internal pointers to the next symbol which was scanned from the input line.

```
ENTER:    nothing
EXIT:     TRUE ZERO: end of expression
          FALSE ZERO: expression pointer updated
               HL==> array entry
```

When INCEXP returns TRUE ZERO the expression symbol pointer is pointing beyond the last symbol; that is, the statement is over.

11.3.2 DECEXP -- Decrement the Expression Array Pointer

This routine is identical to INCEXP except that is decrements the array pointer.

```
ENTER:    nothing
EXIT:     HL==> array entry
```

This routine is useful when a preliminary examination of the expression array is necessary to determine validity before execution. INCEXP can be used to examine the items in the statement for syntax or context and DECEXP can be used to reposition pointers for actual processing. Each call to DECEXP will lower the pointer by one symbol.

11.3.3 CLEAREXP -- Reset the Expression Array Pointer

This routine is used to reset the array pointer back to the first symbol in the statement (the keyword or directive).

```
ENTER:    nothing

EXIT:     nothing
```

This routine is useful if a syntax check of the symbols in the statement is necessary before processing.

11.3.4 CKSTAK1 -- Determine Current Symbol Type

This routine sets the condition codes to indicate the type of the currently pointed entry in the expression array. CKSTAK1 does not affect the expression array pointer.

```
ENTER:    nothing
EXIT:     TRUE CARRY: numeric literal;
               HL==> 3-byte (24 bit) value area
          TRUE ZERO: operator
          TRUE PARITY: reserved word
          TRUE SIGN: operand; item in symbol table
               HL==> value area length byte and value
               A = symbol table status byte
```

11.3.5 CKTYPE -- Determine Operand Type

This routine is used after a call to CKSTAK1 has returned TRUE SIGN indicating that the current symbol in the expression stack is an operand.

```
ENTER:    A = symbol table status byte (from CKSTAK1)
EXIT:     HL unchanged
          A = 1; operand Boolean TRUE
          A = 0; operand Boolean FALSE
          TRUE CARRY: operand is user file
          TRUE ZERO: operand is character string
          TRUE SIGN: operand is decimal number
          TRUE PARITY: operand is octal number
```

11.3.6 CKITEM -- Point and Type the Next Symbol in Array

CKITEM is a composite routine resulting from calls to INCEXP, CKSTAK1, and CKTYPE. It is used to avoid successive calls to each of the three routines. The results of the calls are stored by using the hardware Condition Code Save (CCS) instruction. The result of a CCS instruction is an 8-bit value which, when added to itself, will regenerate the current condition codes.

```
ENTER:    nothing
EXIT:     A, added to self, is result of INCEXP
          B, added to self, is result of CKSTAK1
          C, added to self, is result of CKTYPE
          note: if ADBB gives TRUE SIGN, HL==> value area
```

It should be noted that if adding A to A results in TRUE ZERO, then no other results are meaningful. Likewise, if adding B to B does not result in TRUE SIGN, then adding C to C is meaningless.

11.3.7 OPND1SET -- Set Up First Operand for COMPARE

OPND1SET is used to prepare the first operand for a comparison operation. The operand in a comparison can either be a symbol table operand or a numeric or character literal.

```
ENTER:    nothing; current array pointer used
EXIT:     nothing
```

11.3.8 OPND2SET -- Set Up Second Operand for COMPARE

OPND2SET is used to prepare the second operand for a comparison operation.

```
ENTER:    nothing; current array pointer used
EXIT:     nothing
```

11.3.9 COMPARE -- Compare Two Operands

COMPARE is used to compare to operands which have been set up with OPND1SET and OPND2SET.

```
ENTER:    nothing
EXIT:     TRUE CARRY: operand 1 > operand 2
          TRUE ZERO: operand 1 = operand 2
```

FALSE CARRY, FALSE ZERO: operand 1 < operand 2
FALSE ZERO: operand 1 ~= operand 2

11.3.10 ABORT -- Abort the CHAINPLS Compilation

ABORT is used when a terminal error condition has been detected.

```
ENTER:    HL==> error message
EXIT: does not return; exits to DOS
```

11.3.11 STLOOKUP -- Symbol Table Lookup by Name

STLOOKUP is used to set pointers to an entry in the symbol table when its ASCII character name is known.

```
ENTER:    HL==> 8 character name (blank filled)
EXIT:     TRUE CARRY: item not found
          FALSE CARRY: item found
               HL==> symbol table value string
               A = symbol table status byte
```

Note that because the symbol table status byte is returned in the A register the CKTYPE routine may be called to determine the operand type.

11.3.12 USRSPACE -- Obtain Work Space for User Routine

USRSPACE is used to get work space for use by a user routine. There is no limit to the number of times that USRSPACE may be called. CHAINPLS utilizes all available processor memory and will allocate it until it is expended.

```
ENTER:    BC = amount of space desired
EXIT:     TRUE CARRY: space unavailable
          FALSE CARRY: HL==> space obtained
```

11.3.13 CVBINDEC -- Convert Binary Value to ASCII Decimal

CVBINDEC is used to convert any numeric symbol table item value to its decimal equivalent. Note that CVBINDEC and CVBINOCT can convert the low-order 24-bits of any four byte field.

```
ENTER:    HL==> value area of operand (from CKSTAK1)
EXIT:     HL==> leftmost byte of result field
          C = length of result
```

The result is created in a separate memory area and must be moved from (HL) to the desired location. Note that any 24-bit (three byte) binary value may be converted if it has the general form of a symbol table value entry.

11.3.14 CVBINOCT -- Convert Binary Value to ASCII Octal

CVBINOCT is identical to CVBINDEC except that its result is in ASCII octal.

11.3.15 CVDECBIN -- Convert ASCII Decimal to Binary

CVDECBIN is used to convert standard ASCII decimal character strings into their binary representation.

```
ENTER:    HL==> character string delimited by space
EXIT:     CDE = value
          TRUE CARRY: invalid character in string
          FALSE CARRY: string was valid
```

Although CVDECBIN returns a 24-bit answer, it has an internal limit of 16 bits of accuracy (65K).

11.3.16 CVOCTBIN -- Convert ASCII Octal to Binary

CVOCTBIN is identical to CVDECBIN except that it converts a full 24-bit binary number to ASCII octal.

11.3.17 CHEKFILE -- Open a Disk File if Present

CHEKFILE utilizes a character string containing a file specification and attempts to open the file. If successful, the DOS LF3 Logical File Table will contain the opened LFT entry. If unsuccessful, LF3 will contain the result of SCANFS (Scan File Specification) and can be used for a PREP$ call if desired.

```
ENTER:    nothing; current array pointer used
EXIT:     TRUE CARRY: file not found; LF3 in SCANFS form
          FALSE CARRY: LF3 has opened LFT entry
```

It is important to remember that the contents of LF3 and its associated disk buffer are transient. The LFT entry resulting from a call to CHEKFILE must be saved into a private work area if the file is to be used in later calls. Also, when the file is used the sector described by the LFT entry must be re-read into the disk buffer.

11.3.18 ERRORDSP -- display non-fatal error message

ERRORDSP will display a non-fatal error message and return to the CHAINPLS parsing routines.

```
ENTER:    HL==>user's error message
EXIT:     Does not return
```

This routine is useful when a USERPROG detects an error condition and needs to inform the operator, but does not need to terminate compilation. The user's error message will be displayed in the following format:

```
E:  (nnnn) USER'S ERROR MESSAGE
```

The message should be terminated by an EOL (015) byte. After the message is displayed the rest of the CHAINPLS file will be compiled; however, the output file will not be executed. (See appendix B Terminal Errors "ERRORS FOUND; OUTPUT NOT EXECUTED".)

11.4 Program Assembly and Control

11.4.1 Assembly Time

If the SNAP/3 assembler is used, the command line option "T" must be used during assembly to guarantee that no forward references are left in the output relocatable library member. This is not necessary if the SNAP/2 assembler is used.

The file DOS/EPT should be included at assembly time if any of the DOS entry points are used by the module be assembled. Only the specified CHAINPLS service routines may be left undefined by the assembly.

11.4.2 Execution Time

When the "USERPROG" module is relocatably loaded, the entry point for the module is used as the main entry point each time the module is entered. The programmer should be aware that the module is only loaded once, at the execution of the "USERPROG" statement, regardless of the number of times it is used.

The user module returns to the mainline CHAINPLS compilation logic by executing a "RETURN" operation. Care should be taken that the stack is not disturbed in such a way as to accidentally discard this return address. All registers including the "X" register can be used if needed. The user may use as many as six stack levels.

If the user wishes to perform disk I/O operations, DOS Logical File Table number three (LF3) may be used for this purpose. Note that the contents of this LFT are transient and thus a user must save any opened LFT entry into a private area and must reread any disk sectors currently in use between invocations. LF3 is used by all CHAINPLS user file I/O (READ, WRITE, OPEN, CLOSE).

The same external reference and definition table is used throughout a CHAINPLS execution. Thus a module may be loaded via a USERPROG statement that contains external definitions which are necessary for a later module. This should be avoided whenever possible, since improper use can cause modules to fail to load for no apparent reason.

APPENDIX A. A CHAINPLS PROGRAMMING EXAMPLE

The following example is a text file of CHAINPLS statements which ask the console operator for numbers and determine the divisors of the entered number. The program ends when the number zero is entered.

```
// ASSIGN NUMBER=1
// WHILE NUMBER>0
// KEYIN "ENTER NUMBER TO BE TESTED: " NUMBER
// ASSIGN LOOP=2
// ASSIGN LIMIT=NUMBER/2
// SET FLAG=FALSE
// WHILE LIMIT>=LOOP
// CLICK
// ASSIGN REM=NUMBER/LOOP
// ASSIGN XREM=REM*LOOP
// DEBUG NUMBER LOOP REM XREM
// IF XREM=NUMBER
// KEYIN "THE NUMBER #NUMBER# IS DIVIDED BY #LOOP#"
// SET FLAG=TRUE
// XIF
// ASSIGN LOOP=LOOP+1
// END
// IF ~FLAG
// KEYIN "SORRY, THE NUMBER #NUMBER# HAS NO INTEGER DIVISORS"
// BEEP
// XIF
// END
```

Here is some sample output from the given example:

```
K:ENTER NUMBER TO BE TESTED: 123
K:THE NUMBER 123 IS DIVIDED BY 3
K:THE NUMBER 123 IS DIVIDED BY 41
K:ENTER NUMBER TO BE TESTED: 8
K:THE NUMBER 8 IS DIVIDED BY 2
K:THE NUMBER 8 IS DIVIDED BY 4
K:ENTER NUMBER TO BE TESTED: 60
K:THE NUMBER 60 IS DIVIDED BY 2
K:THE NUMBER 60 IS DIVIDED BY 3
K:THE NUMBER 60 IS DIVIDED BY 4
K:THE NUMBER 60 IS DIVIDED BY 5
K:THE NUMBER 60 IS DIVIDED BY 6
K:THE NUMBER 60 IS DIVIDED BY 10
K:THE NUMBER 60 IS DIVIDED BY 12
K:THE NUMBER 60 IS DIVIDED BY 15
K:THE NUMBER 60 IS DIVIDED BY 20
K:THE NUMBER 60 IS DIVIDED BY 30
K:ENTER NUMBER TO BE TESTED: 23
K:SORRY, THE NUMBER 23 HAS NO INTEGER DIVISORS
K:ENTER NUMBER TO BE TESTED: 101
K:SORRY, THE NUMBER 101 HAS NO INTEGER DIVISORS
K:ENTER NUMBER TO BE TESTED: 0
K:SORRY, THE NUMBER 0 HAS NO INTEGER DIVISORS
```

Note that the DEBUG statements in the file had no effect on the output since the "T" option was not specified on the command line.

APPENDIX B. ERROR MESSAGE SUMMARY

B.1 The Compilation Phase Errors

B.1.1 The Terminal Errors

Terminal compilation time errors cause program suspension and turn on the DOS "ABORT" flag. The messages all appear near the bottom of the screen and do not begin with a CHAINPLS display line indicator (i.e., "E:"). These are the terminal error messages possible:

INVALID COMMAND LINE SPEC
The command line entered contains a duplicate definition of a data item or attempts to define the value of a reserved word.

INVALID LITERAL ON COMMAND LINE
The command line contained an octal literal (beginning with a zero) which contained an 8 or 9.

INVALID OPTION SPEC
The "OPTIONS=xxxxx" statement from the command line contains an undefined option.

ERROR IN RELOCATABLE LOAD
An unrecoverable error was encountered while attempting to load a printer driver from the file UTILITY/REL. Primary reason for this would be the absence of UTILITY/REL or an obsolete version.

SYSTEM ERROR IN CHAINING
Chain file execution was requested while chaining was already active but control information within DOS has been destroyed.

DOS SYSTEM ERROR
The DOS has found an unrecoverable error in the DOS function loader mechanism.

PS ACTIVE; CHAINING ABORTED
CHAINPLS cannot be run from the remote or fixed partition under PS, PS66, or U.P.S.

SYSJOB/SYS MISSING OR INVALID
    CHAINPLS was invoked while chaining was active and the job file cannot be found or is invalidly structured.

SYSLOG/SYS MISSING OR INVALID
    Queued logging was requested and the log file is in an invalid format.

OV1 OR OV2 MISSING FROM COMMON DRIVE
    The execution control overlays must be present and on the same drive.

SYSTEM DATA AREA DESTROYED
    Control areas within DOS are destroyed and the operating system must be reloaded.

USERPROG LOAD ERROR
    A USERPROG directive specifed the loading of a member which was either not found in the library file, the library file itself was not found, or the module was too large for the remaining free space. In addition, this error could be caused by either forward references in the relocatable module or by unresolved external references.

COMPILATION ABORTED BY KEYBOARD KEY
    The user has voluntarily terminated execution of the compilation phase by pressing the keyboard key on the processor console.

EXECUTION ABORTED BY ABORT STMT
    The execution was terminated due to encountering an "ABORT" statement in an active block.

SYMBOL TABLE MEMORY OVERFLOW
    The processor/partition memory area available to CHAINPLS has been filled completely by symbol table values and definitions. DEBUG statements can be helpful in determining unused or re-usable data items to circumvent this condition.

INPUT FILE OVERWRITE REQUESTED
    The name of the output file is the same as the name of the input file; CHAINPLS prevents destruction of the input file.

ERRORS FOUND; OUTPUT NOT EXECUTED
    CHAINPLS is running in execute mode and the compilation phase detected errors in execution. No chaining takes place.

INPUT FILE PARITY/OFF-LINE ERROR

The input file specified contained a parity error or the drive containing it went off-line during processing.

INPUT FILE FORMAT/RANGE ERROR
The input file specified contained a format error or did not contain an EOF marker.

UNKNOWN RECORD IN INPUT FILE
The input file contained a record with unrecognizable binary data.

INPUT FILE IS LIBRARY. NO MEMBER SPECIFIED.
The input file is in library format, but no member name was on the command line. The member name is required.

B.1.2 The Syntax and Parsing Errors

These errors of CHAINPLS are errors in syntax or expressions that can be ignored to allow the execution to continue. The error messages will appear on any output print device selected and on the console display. Each message is preceded by an indicator ("E:") followed by the line number. The console display is double spaced after error messages.

As usual with language processors, the occurrence of certain errors can cause a "cascade" of other error conditions. For example, an erroneous "IF" statement will cause, due to its rejection, the associated "XIF" to result in additional error.

CHAINPLS sets an internal flag after detection of any run-time error. This flag will prevent execution of the output. Additionally, the DOS "ABORT" flag is set by the occurrence of any of these errors. The syntax errors are:

E: (line) INVALID SET STATEMENT
A SET statment was encountered which lacked an operand, an equal sign, or an expression; or the items were misplaced.

E: (line) INVALID EXPRESSION
An expression was found with mismatch parentheses, unidentified operators, or invalid operator usage.

E: (line) INVALID IF STATEMENT
An IF statement was encountered which did not contain an expression.

E: (line) INVALID ELSE STATEMENT

An ELSE statement was encountered and the last block control statement was not an IF.

E: (line) INVALID END STATEMENT
An END statement was encountered and the last block control statement was not a "BEGIN" or an "END".

E: (line) INVALID WHILE STATEMENT
A WHILE statement was encountered without an expression.

E: (line) INVALID ASSIGN STATEMENT
An ASSIGN statement was encountered which lacked an equal sign, an operand, or an expression.

E: (line) INVALID XIF STATEMENT
An XIF statement was encountered and the last block control statement was not an ELSE or IF.

E: (line) INVALID BEGIN STATEMENT
A BEGIN statement was encountered which contained something beyond the word BEGIN.

E: (line) INVALID LITERAL
A numeric literal was encountered which was either non-numeric or contained digits not allowable in the given base.

E: (line) INVALID USE OF RESERVED WORD
A statement identifier or some other reserved word was used in an expression.

E: (line) INVALID READ STATEMENT
A READ statement was encountered whose first operand was not an OPENed file, or which lacked a first operand.

E: (line) INVALID WRITE STATEMENT
A WRITE statement was encountered whose first operand was not an OPENed file, or which lacked a first operand.

E: (line) INVALID OPEN STATEMENT
An OPEN statement was encountered which was of improper form or whose file name specification was not found on a ready drive or was a numeric data item.

E: (line) INVALID CLOSE STATEMENT
A CLOSE statement was encountered which did not contain a first operand or whose first operand was not an OPENed file.

E: (line) INVALID STATEMENT

A control statement was encountered which was erroneous or unrecognizable.

E: (line) DIVIDE BY ZERO ATTEMPTED
A division operation during expression evaluation had a zero as a divisor.

E: (line) INPUT RECORD TOO LARGE
An input record whose length was 100 bytes or greater was either read from the input file or generated by symbol replacement.

E: (line) CIRCULAR REPLACEMENT IN INPUT RECORD
Replacement symbols in input record cause recursive replacement to be invoked more than 256 times.

E: (line) OUTPUT RECORD LENGTH > 80
An output record after symbol replacement was longer than 80 bytes; CHAIN will not accept such a record.

E: (line) INVALID OUTPUT STRING LENGTH/INDEX
A substring operation resulted in a zero index, or a length or index beyond the range of the operand string.

E: (line) INVALID KEYIN STATEMENT
A KEYIN statement was encountered which contained something beyond the word "KEYIN" that was not an operand or a literal.

E: (line) INVALID FUNCTION CALL
A FFILE or MMEMBER operation was lacking parentheses, operands, the proper number of operands, or the operands were not character strings.

E: (line) INVALID OPERAND FOR BIT FUNCTION
The operand for a BIT function must be a TRUE character string.

E: (line) INVALID OPERAND FOR UNBIT FUNCTION
The operand for a UNBIT function must be a TRUE numeric variable.

E: (line) INVALID OR UNKNOWN INCLUSION
An INCLUDE statement was processed whose argument could not be found or was not a character string.

E: (line) IF/BEGIN/WHILE STACK OVERFLOW
The number of active block control statements (IF, ELSE, BEGIN, WHILE) exceedes the number allowed.

E: (line) EXPRESSION STACK OVERFLOW
The number of operands, literals, and operators in an expression exceeds the number allowed.

E: (line) INVALID ITEM IN DISCARD STATEMENT STATEMENT
Literals are not allowed in DISCARD statements.

E: (line) INVALID OPTION MODIFICIATION
Only the R,I,D,T, and O options can be dynamically set and reset.

B.2 Execution Phase Errors

The execution phase errors are those that occur after the input text file has be processed (compiled) and the execution of the resulting commands has begun. These errors result in the CHAINPLS execution control overlay (CHAINPLS/OV1) returning control to the DOS.

ABORT BIT SET; CHAINING ABORTED
This message will appear upon the execution of an ABTIF statement if the "abort flag" is set on in the DOS control flag (DOSFLAG). The system utility ABTONOFF can be used to guarantee that this condition is clear prior to execution of utilities which can set it.

EXIT OR EOF ERROR; CHAINING ABORTED
This message can occur in one of three ways: 1) An executing program makes an "error" return to the DOS. This condition generally implies a serious failure in a program, e.g., a "format" trap. 2) A running program requests a keyin and the SYSJOB/SYS file contains no more records. 3) A keyin response within the SYSJOB/SYS file is longer (contains more bytes) than the program requesting the keyin allows.

EXECUTION ABORTED BY KEYBOARD KEY
CHAINPLS will terminate the execution of a job stream if the keyboard key is depressed **between** program executions. This message is merely documentation of an action taken at user request. Note that the pointers in SYSJOB/SYS are left indicating the last command executed.

SYSJOB/SYS FILE MISSING OR INVALID
The SYSJOB/SYS file was not found during when opened or the control information within the file is contaminated. This error commonly occurs when a restart is attempted on a job file that has been completed and deleted.

SYSTEM0/SYS MISSING OR UNLOADABLE
    The primary operating system resident program on the "booted" drive is unusable.

RESTART DISCONTINUED AT USER REQUEST
    At any time during "interactive restart", the user may respond to a keyin with an asterisk ("*"); this response causes immediate return to DOS.

CHAIN FILE COMPLETED: nnnnnnnn/ext
    Information only: the named chain file has successfully completed.

COMMAND LINE/NAME INVALID
    The "override job file name" is invalid or was not found.

CONTROL AREA DESTROYED; CHAINING TERMINATED
    The resident SYSJOB/SYS control area has been modified by another program and is thus unusable by CHAINPLS/OV1. The chain file may be restart from last command or interactively.

CONTROL AREA DESTROYED; CHAINING RESUMED AT LAST COMMAND
    A "next line" restart request cannot be fulfilled because the resident file control area is not intact; chaining is resumed at the last executed (but not completed) program.

APPENDIX C. USER FILE I/O PROGRAMMING EXAMPLE

The following example is a text file of CHAINPLS statements which processes two input files previously created by the FILES utility. The purpose of the program is to match the two files and write an output file which specifies which file names are missing from which files. Note that both of the input files are sorted into ascending sequence by file name (whole record).

```
&...
&... First get the names of the input
&... files from the operator
&...
// KEYIN "ENTER FIRST 'FILES' FILE: " FILE1NAM
// KEYIN "ENTER SECOND 'FILES' FILE: " FILE2NAM
&...
&... Now get the name of the output file and
&... verify the existence of the input files.
&...
// KEYIN "ENTER OUTPUT FILE: " OUTNAME
// IF ~((FFILE(FILE1NAM))&(FFILE(FILE2NAM)))
// BEEP
// KEYIN "EITHER #FILE1NAM# OR #FILE2NAM# DOESN'T EXIST!"
// ABORT
// XIF
&...
&... Now open the files and read a record from each
&... (standard match/merge technique)
&...
// OPEN FILE1(FILE1NAM)
// OPEN FILE2(FILE2NAM)
// OPEN OUTFILE(OUTNAME)
// READ FILE1,FILE1REC
// READ FILE2,FILE2REC
&...
&... Now create a main block of statement to be
&... executed as long as either file is open.
&... Note the use of the file names as Boolean
&... values.
&...
// WHILE FILE1|FILE2
&...
```

```
&... Now a nested block to be processed if either
&... FILE1 or FILE2 is no longer open.
&...
// IF ~(FILE1&FILE2)
&...
&... Now a nested block to be processed if
&... FILE1 is no longer open.
&...
// IF ~FILE1
// ASSIGN MISSFILE=FILE2REC^(1:12)
// KEYIN "FILE #MISSFILE# MISSING FROM FILE1"
// WRITE OUTFILE,MISSFILE,"<== FROM ",FILE1NAME
// READ FILE2,FILE2REC
// ELSE
&...
&... Now a nested block to be processed if
&... FILE2 is no longer open.
&...
// ASSIGN MISSFILE=FILE1REC^(1:12)
// KEYIN "FILE #MISSFILE# MISSING FROM FILE2"
// WRITE OUTFILE,MISSFILE,"<== FROM ",FILE2NAME
// READ FILE1,FILE1REC
// XIF
&...
&... Now a block if both files are still open.
&...
// ELSE
&...
&... If the records match, read a record
&... from both files, otherwise read the file
&... whose record is lower in alphanumeric value.
&...
// IF (FILE1REC^(1:12))=(FILE2REC^(1:12))
// READ FILE1,FILE1REC
// READ FILE2,FILE2REC
// ELSE
// IF (FILE1REC^(1:12))<(FILE2REC^(1:12))
// ASSIGN MISSFILE=FILE1REC^(1:12)
// KEYIN "FILE #MISSFILE# IS MISSING FROM FILE2"
// WRITE OUTFILE,MISSFILE,"<== FROM ",FILE2NAM
// READ FILE1,FILE1REC
// ELSE
// ASSIGN MISSFILE=FILE2REC^(1:12)
// KEYIN "FILE #MISSFILE# IS MISSING FROM FILE1"
// WRITE OUTFILE,MISSFILE,"<== FROM ",FILE1NAM
// READ FILE2,FILE2REC
// XIF
// XIF
```

```
// XIF
&...
&... Now check if an end-of-file was detected
&... during any recent reading
&...
// IF (.FILE1REC=0)&(FILE1)
// CLOSE FILE1
// XIF
// IF (.FILE2REC=0)&(FILE2)
// CLOSE FILE2
// XIF
// END
&...
&... Since the input files are closed
&... automatically by the main read
&... loop, all that remains is to
&... close the output file.
&...
// CLOSE OUTFILE
&...
&... Now cause CHAINPLS to exit to the LIST program
&... to list the resulting output file.
&...
// EXIT "LIST #OUTNAME#"
```

APPENDIX D. USER-WRITTEN SUBROUTINE PROGRAMMING EXAMPLE

The following example consists of three parts: the assembly language program, the CHAINPLS input file which executes it, and the log of keyin/display activity associated with running the input file.

D.1 Assembly Language Subroutine

The following program existed in a text file called "COMPARE/TXT". It was assembled with the command line:

```
SNAP3 COMPARE,CHAINPLS/REL;T
```

This command line put the relocatable output file into a library called "CHAINPLS/REL"; the "T" option forces the assembly to perform two passes to eliminate forward references.

```
*
.   THIS PROGRAM PARSES A STATEMENT OF THE FORM:
.
.      // COMPARE <operand 1>,<operand 2>
.
.    AND DISPLAYS THE RESULTS ON THE SCREEN
.
COMPARE   PROG                          THE MEMBER NAME WILL BE "COMPARE"
          INC       DOS/EPT             INCLUDE THE DOS. ENTRY POINTS
COMPARE   ORG       0                   THE PAB MUST BE RELOCATABLE
          USE       COMPARE
PREP      DC        H,LC,V,BL,ECL,EOS   MESSAGE POSITIONING STRING
GTR       DC        'GREATER',EOL
LSS       DC        'LESS',EOL
EQL       DC        'EQUAL',EOL
.
.
START     CALL      CKITEM              GET THE FIRST SYMBOL
          ADA                           CHECK FOR OVER ALREADY
          RTZ                           BACK IF NOTHING
          ADBB                          CHECK FOR OPERAND OR DATA
          RTZ                           BACK IF OPERATOR
          RTP                            OR RESERVED WORD
          CALL      OPND1SET            SET UP THE FIRST OPERAND
```

```
        CALL    CKITEM          GET THE NEXT ITEM
        ADA                     CHECK FOR OVER ALREADY
        RTZ                     BACK IF NOTHING
        ADBB                    CHECK FOR OPERAND OR DATA
        RTZ                     BACK IF OPERATOR
        RTP                      OR RESERVED WORD
        CALL    OPND2SET        SET UP THE SECOND OPERAND
        HL      PREP            PREPARE THE SCREEN
        CALL    DSPLY$
        PUSH    DE              SAVE THE SCREEN POSITION
        CALL    COMPARE         COMPARE THE TWO
        POP     DE              RESTORE THE SCREEN POSITION
        HL      GTR             ASSUME GREATER
        JTC     SHOW            YES, GREATER
        HL      EQL             TRY EQUAL
        JTZ     SHOW            YES,EQUAL
        HL      LSS             ALL THAT'S LEFT IS LESS
SHOW    EX      CLICK
        JMP     DSPLY$          RETURN TO MAINLINE AFTER DISPLAY
        END     START
```

D.2 CHAINPLS Input File

The following records were contained in a text file called "TEST/TXT". Its only function is to exercise assembly language "COMPARE" program.

```
&...
&... this statement will load member "COMPARE"
&... from library "CHAINPLS/REL"
&...
// USERPROG COMPARE("CHAINPLS/REL","COMPARE")
&...
&... now declare some numeric and alpha fields
&...
// ASSIGN NUMR1=1
// ASSIGN NUMR2=2
// ASSIGN ALPHA1="A"
// ASSIGN ALPHA2="B"
&...
&... prepare a loop which will execute
&... until an "*" is entered
&...
// SET READY=TRUE
// WHILE READY
// KEYIN "FIRST OPERAND NUMERIC OR ALPHA? A/N " AN
// IF AN="*"
// SET READY=FALSE
// ELSE
// IF AN="N"
// KEYIN "ENTER NUMERIC: " NUMR1
// ASSIGN OP1="NUMR1"
// ELSE
// KEYIN "ENTER ALPHA: " ALPHA1
// ASSIGN OP1="ALPHA1"
// XIF
// KEYIN "SECOND OPERAND NUMERIC OR ALPHA? A/N " AN
// IF AN="N"
// KEYIN "ENTER NUMERIC: " NUMR2
// ASSIGN OP2="NUMR2"
// ELSE
// KEYIN "ENTER ALPHA: " ALPHA2
// ASSIGN OP2="ALPHA2"
// XIF
&...
```

```
&... note that by assigning the names of
&... of the fields to other string variables,
&... the need for a series of four IF
&... statements and COMPAREs is eliminated.
&...
// DEBUG #OP1# #OP2#
// COMPARE #OP1#,#OP2#
// XIF
// END
```

D.3 Execution Results

The lines below are the results of a short execution of the above CHAINPLS input file.

```
CHAINPLS TEST;OP=C
CHAINPLS  1.1.B   CHAIN FILE COMPILATION   JUNE 15, 1978

FIRST OPERAND NUMERIC OR ALPHA? A/N A
ENTER ALPHA: AABB
SECOND OPERAND NUMERIC OR ALPHA? A/N A
ENTER ALPHA: AACC
LESS
FIRST OPERAND NUMERIC OR ALPHA? A/N N
ENTER NUMERIC: 123
SECOND OPERAND NUMERIC OR ALPHA? A/N N
ENTER NUMERIC: 123
EQUAL
FIRST OPERAND NUMERIC OR ALPHA? A/N A
ENTER ALPHA: 1234
SECOND OPERAND NUMERIC OR ALPHA? A/N N
ENTER NUMERIC: 1324
LESS
FIRST OPERAND NUMERIC OR ALPHA? A/N A
ENTER ALPHA: GOGO
SECOND OPERAND NUMERIC OR ALPHA? A/N A
ENTER ALPHA: GOGOA
LESS
FIRST OPERAND NUMERIC OR ALPHA? A/N *

CHAIN FILE COMPLETED: TESTX/TXT
CHAINING COMPLETED
```

APPENDIX E. CHAINPLS RELOCATABLE SUBROUTINE LIBRARY

Supplied with the release tapes of CHAINPLS is a file called CHAINPLS/REL. This file is a DOS-standard relocatable library file created by the SNAP3 assembler. It contains various utility parsing routines designed to be used by programmers with unusual requirements for system control.

Routines from the library are loaded into memory by use of the USERPROG statement. It is the user's responsibility to load the routines which are to be used. Since memory is allocated dynamically, there is no certainty that all routines specified for loading can actually be loaded.

It is important to note that the use of the USERPROG facility and the subroutine library should be limited to those with an extensive assembly language background and an in-depth understanding of Datapoint processor architecture and disk structures. No support will be given to use of the USERPROG facility or the use of the relocatable subroutine library.

Each subroutine in the library, when loaded, becomes a new statement type. It may, after loading, be invoked as needed and no additional load will take place. The following descriptions are brief and contain only syntactic definitions. It is recommended that a user avoid the use of any subroutine whose definition or function is not fully understood.

E.1 SAVEPTR -- Save the Position of a User File

Format:

```
// SAVEPTR <file var>,<octal var>
```

SAVEPTR will save the position (LRN, BUFADR) of a user file into an octal variable. It must be called PRIOR to reading the record whose position is to be saved.

E.2 RESTPTR -- Restore the Position of a User File

Format:

```
// RESTPTR <file var>,<octal var>
```

RESTPTR will reposition a user file back to the same position it possessed when the SAVEPTR statement was executed.

E.3 FREE -- Determine Free Memory Available

Format:

```
// FREE <numeric var>
```

FREE will return into the numeric variable the amount of free memory, in bytes, remaining in the processor in use.

E.4 ROLLOUT -- Save Execution State and Return to DOS

Format:

```
// ROLLOUT <char var 1>,<char var 2>
```

ROLLOUT will create a command file in standard object file format containing all memory in use by the execution of CHAINPLS. The name of this command file will be <char var 1>. Optionally, a second parameter may be specified (<char var 2>) which contains a command line to be inserted into MCR$ preceded by ">>" for use by NXTCMD. Note: the CHAINPLS execution may be restarted by keying in the name of the command file created to DOS.

E.5 STARTIME -- Start a Timing Function

Format:

```
// STARTIME <numeric var>
```

STARTIME uses a DOS four millisecond interrupt vector to count milliseconds into the <numeric var>. The variable is automatically zeroed by the operation. Note: only ONE timer operation may be done at any time.

E.6 STOPTIME -- Stop a Timing Function

Format:

```
// STOPTIME
```

STOPTIME discontinues the timing process started by the STARTIME statement.

E.7 POSIT -- Position a User File to a Logical Record

Format:

```
// POSIT <file var>,<numeric var or literal>
```

The POSIT subroutine will position a user file to the LRN specified in the <numeric var or literal>. It is the user's responsibility to validate the LRN; this subroutine will cause space to be allocated to the file to allow positioning to the LRN given.

E.8 FILENAME -- Obtain DOS Filename

Format:

```
// FILENAME <PDN num var>,<PFN num var>,<char var>,
                        (<subdirectory num var>)
```

FILENAME will return the 11-character DOS file name of the file whose PDN is given in the first <numeric var> and whose PFN is given in the second <numeric var>. If the file does not exist, a NULL string will be returned. In addition, if a final numeric variable is given, the subdirectory number will be delivered into it.

E.9 SURNAME -- Obtain DOS Subdirectory Name

Format:

```
// SURNAME <PDN numeric var>,
          <subdir numeric var>,<subdirectory char var>
```

SURNAME will utilize the numeric variable containing the PDN and the numeric variable containing the subdirectory number and return the subdirectory name into the <character variable>. Note

that the SYSTEM subdirectory is 0377.

SURNAME is retained for compatability only. Pre-defined data items SUR0-SUR31 contain the names of the current subdirectories of all on line volumes.

E.10 KILL -- KILL an Open User File

Format:

```
// KILL <file var>
```

KILL will remove all record of an open file from the disk directory via the standard DOS mechanism of CHOP/CLOSE.

E.11 CHOP -- CHOP an Open User File

Format:

```
// CHOP <file var>,<LRN numeric var>
```

CHOP will cause all space beyond the logical record number given in the numeric variable to be deallocated from the file and returned to the free space on the disk. If the LRN specified is not in allocated space, space is allocated for it and the CHOP occurs afterwards. In accordance with the standard DOS interface, deallocation only occurs at CLOSE time; thus for CHOP to take effect, the file must eventually be explicitly CLOSEd.

E.12 PROTECT -- Change Protection on a User File

Format:

```
// PROTECT <file var>,<char var or lit>
```

The <char var> can contain only the following characters:

D	Delete-protect the file
W	Write-protect the file
X	Remove all protection from the file

In accordance with the standard DOS interface, protection changes only occur at CLOSE time; thus for PROTECT to take effect, the file must eventually be explicitly CLOSEd.

E.13 NEXTMEM -- Obtain First/Next Member Names From Library

Format:

```
// NEXTMEM <file var>,<type var>,<name var>,<LRN var>
```

Where:

<file var> is an OPENed library file
<type var> is a NUMERIC variable
<name var> is a CHARACTER variable
<LRN var> is a NUMERIC variable

This routine will deliver all member names from a library. No I/O operations can be executed on the library (READ, WRITE, etc.) during the course of the NEXTMEM processing. The file must be a library file which has been opened but has had no other I/O operations performed upon it. When NEXTMEM returns a null (zero) length string as the name, there are no more member names in the library. The type variable returns the library types as documented in the DOS User's Guide. The LRN returned is the logical record number of the start of the member.

E.14 NEXTSYM -- Obtain Next Symbol Name from Symbol Table

Format:

```
// NEXTSYM <char var>
```

This routine allows a CHAINPLS programmer to obtain the names of all user symbols which have been declared and stored into the symbol table. This can be very useful for extensive debugging or in determination of memory usage. After the execution of NEXTSYM, the character string variable contains the name of the next physical item in the symbol table. The items are not returned in any particular order, but iterated calls to NEXTSYM will deliver all user symbol names once and only once. When no more user symbols exist, the character variable will be given a length of zero (null). Note that variables declared between calls to NEXTSYM are not guaranteed to be returned.

E.15 PAUSE -- Timed Suspension of Processing

Format:

```
// PAUSE <numeric var or literal>
```

This routine is used when a suspension of all CHAINPLS processing is desired for a time. The numeric variable or literal must contain the number of milliseconds to wait; this value must be greater than 4. The display key will be checked every 5 seconds, and, if depressed, will cause the PAUSE to be exited. This routine utilizes, through the DOS Function interface, a 4 millisecond DOS F/G process.

E.16 PRINT -- User print interface

```
// PRINT <variable or literal>,<variable or literal>,...
```

This routine allows user printing during the compile phase of CHAINPLS. Before PRINT can be USERPROG'ed one of the drivers from UTILITY/REL (LOCAL, SCREEN, SERVO or FILE) must be USERPROG'ed. If one of the drivers is not USERPROG'ed first a load error will occur.

The first character of the first variable or literal is used for format and operation control. If it is one of the standard ASCII control characters the indicated function will be performed:

```
1 =          Skip to top of page
0 =          Double space.
- =          Triple space.
<blank> =Single space.
```

In addition the characters 'O', 'Q' and 'C' are defined as follows:

```
O   Open a new file.
Q   Queue the print information to the end of an existing file.
C   Close the print file.
```

If the FILE driver is used then the "O" (open) or "Q" (queue) control character must be in the first line printed. The rest of the print line containing the open or queue control character is used as a file name to open. The "C" (close) control character

should be the last line printed. This writes the end of file and updates the file control sector. If a file is started with the queue command and is not closed the data printed to the file will be lost.

If the LOCAL, or SERVO driver is used the open or queue command check that a printer is on line and acquires the printer for use by CHAINPLS. The close command releases the printer and assures even page parity.

If the SCREEN driver is used the open, queue, and close command characters have no effect and the line is ignored.

It is recommended, but not necessary, that the first variable of the line be reserved for command control and be 1 character in length.

A print line is built consisting of all specified variable and literals up to a maximum of 134 bytes. Zero length varables are handled properly. Only character and numeric variables can be printed. Octal and decimal data is converted to ASCII for output, however the length of the resulting line is indeterminate.

An example of useing the file print driver follows.

```
// USERPROG DRIVER ("UTILITY/REL","FILE")
// USERPROG PRINT ("CHAINPLS/REL","PRINT")
// PRINT "O PRTFILE/PRT"
// PRINT "1","TOP OF FORM"
// PRINT "0","DOUBLE SPACE"
// PRINT " ","SINGLE SPACE"
// PRINT "C"
```

No reference should be made to the directive DRIVER. By changing the member in the first USERPROG directive (to "LOCAL" or "SCREEN") the print could be sent to the display screen of local printer.

The PRINT userprog should not be used if the command line options L or P are used. If used in this manner the lines from the CHAINPLS compilation will be intermixed with the user output on one of the selected output devices.

Manual Name_____

Manual Number_____

READER'S COMMENTS

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Name_____ Date_____

Organization_____

Street_____

City_____ State_____ Zip Code_____

All comments and suggestions become the property of Datapoint.

Fold Here

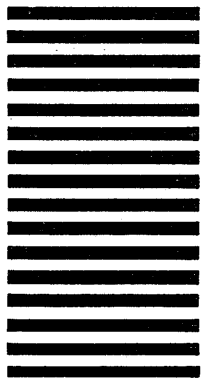
Fold Here and Staple

FIRST CLASS
Permit
5774
San Antonio
Texas

BUSINESS REPLY MAIL
No Postage Necessary if mailed in the United States

Postage will be paid by:

DATAPOINT CORPORATION
DIRECTOR OF SOFTWARE SUPPORT
8550 Datapoint Drive, Mail Station# N60
San Antonio, Texas 78284



CHAPTER 7. ABTONOFF COMMAND

7.1 Purpose

The ABTONOFF command is used to manually modify the ABTIF bit in DOSFLAG (see the description of //ABTIF in the chapter on the CHAIN command.)

7.2 Use

The command line for ABTONOFF is:

ABTONOFF [<condition>]

Where <condition> is one of "ON" or "OFF", specifying the desired condition of the bit. The command will display the prior condition of the bit before modifying its status. If it is desired to just manually inspect the bit without modifying it, specify no <condition>.

CHAPTER 16. CHAIN COMMAND

16.1 Purpose

The CHAIN command executes a series of programs as defined by a procedure file created by the user. The procedure file contains the commands to invoke all required programs, and all inputs for those programs. Basically, CHAIN replaces the DOS keyboard entry routine with a routine that reads lines from a work file when the keyboard entry routine is called. Each time any program would normally request a line to be entered from the keyboard, it will read from the work file instead. When the last line of the work file has been read, DOS is reloaded and commands are again accepted from the keyboard.

CHAIN features several directives to control the procedure executed. Tags defined on the CHAIN command line can be specified to modify lines of the procedure file. CHAIN provides procedure restart capabilities via "CHAIN *" and "CHAIN/OV1". When used with AUTO and AUTOKEY, CHAIN provides an extensive automatic procedure facility, as described in the AUTOKEY chapter.

The procedure file is a normal DOS text format file. Procedure files are generally created using the DOS editor or the BUILD command, but may also be created by any means producing a suitable text file (a DATABUS program, for example).

16.2 Use

The command line to invoke a CHAIN procedure is of the form:

CHAIN <procedure>[;<tag_1>[=<val_1>][,<tag_2>[=<val_2>]. . .][-]]

<procedure> is the user-defined chain procedure file. This file must already exist and must be specified on the command line. The default extension is /TXT. The <tag_n> and <val_n> entries in the option field are chain tags and their substitution values, described fully below. The substitution value for a tag may be specified in the form <tag_n>#<val_n># as well as in the form <tag_n>=<val_n>.

The CHAIN command line can be extended to more than one line

by placing a hyphen (-) at the end of the option field. After scanning the current line of the command, CHAIN will display a colon as a prompt for the operator and wait for entry of another line of tags and substitution values. The command can be continued for several lines by repeated use of the hyphen.

16.2.1 CHAIN Compilation

CHAIN executes two phases, the first of which is compilation. During compilation the specified procedure file is read and compiled into a chain work file. Compilation consists of evaluating and executing CHAIN directives and performing tag substitution. The output of compilation is placed in a file called CHAINP/SYS, which directs the operation of the program chain during execution phase.

The chain work file is always placed on the same logical drive as CHAIN/CMD and CHAIN/OV1, the CHAIN program files. When operating under PS (Datapoint Partition Supervisor) the partition ID is used in the work file name instead of "P" to assure unique identification of the chain work file for each partition. The work file is placed in subdirectory SYSTEM no matter what the current subdirectory is, so the current subdirectory can be changed during the chain and the work file will still be accessible. If the work file is created on an ARC (Attached Resource Computer) remote volume it is placed in the current subdirectory (rather than SYSTEM) to avoid work file usage conflicts among different applications processors.

When CHAIN is used recursively (that is, when CHAIN is invoked from within a chain procedure) the same work file is re-used, the additional compiled information being added to the end of the file. The extent of recursive nesting of chain procedures is limited only by the amount of space available for the work file.

16.2.2 CHAIN Execution

Execution begins following compilation, when the first line of the chain work file is read and given to DOS as a command line input. Execution continues until the work file is exhausted or a fatal error occurs. During CHAIN execution the DOS keyboard entry routine is replaced by a disk read routine so that any entry normally read from the keyboard will be read instead from the chain work file. For details on this execution interface see the section on "CHAIN Programming Considerations".

CHAIN execution is aborted when:

1. A line from the chain work file is longer than allowed. DOS command lines within the chain procedure can be 80 characters long. The allowable length of lines for input to different programs depends on the programs used. For example, when a program requests a file name it generally allows about 20 characters to be entered. If a chain procedure gave a line of 30 characters in response to such a request the chain would abort.

2. The end of the work file is reached while a program is requesting input. The work file must provide all responses needed for execution of the programs used; it cannot invoke a program then end without supplying all required inputs.

3. An //ABTIF directive is executed when the ABTIF bit is set. See the section on "ABORT Directives".

4. A program executing during the chain procedure terminates in a fatal error. Each program can control whether it aborts or continues a chain upon termination. For details see the section on "CHAIN Programming Considerations".

16.3 Tag Definition

The CHAIN command line can contain both tag names and substitution values for the tags. The tag names can be from one to eight characters in length and may have values from one to seventy characters in length. A tag must contain only letters or digits. The value of a tag may contain any valid character except comma (,), equals (=) or pound sign (#). The character restriction depends on the syntax being used.

A tag is defined by just its presence on the CHAIN command line. Tags may have a value given to them by one of the following syntaxes:

```
CHAIN DOIT;LIST,DATE=30NOV76,TIME=1500hr      (New Syntax)

CHAIN DOIT;LIST,DATE#30NOV76#,TIME#1500hr#    (Old Syntax)
```

Both syntax structures are supported and the result of the two CHAIN commands is identical. The tag LIST has been defined but has a null value; DATE has the value of 30NOV76 and TIME has the value of 1500hr.

CHAIN allows two uses to be made of tags:

1. A tag can be tested to determine whether it was defined on the CHAIN command line.

2. The value of the tag can be substituted on CHAIN input statements before the line is written to the work file.

16.4 CHAIN Directives

All CHAIN directives are denoted by the characters "//" at the beginning of a line. Any number of spaces (including zero) are scanned until the CHAIN directive is reached. The first thing after the "//" must be a valid CHAIN directive else an error message is issued and CHAIN is aborted. The following is a list of these statements.

```
//IFS           IF SET (TAG DEFINED)
//IFC           IF CLEAR (TAG NOT DEFINED)
//XIF           END OF IF
//ELSE          REVERSE EFFECT OF IF
//BEGIN         BRACKETS A GROUP OF
//END             IF/ELSE/XIF STATEMENTS
//.             EXECUTION TIME COMMENT
//*             EXECUTION TIME BREAKPOINT
//ABORT         ABORT CHAIN COMPILATION
//ABTIF         CONDITIONALLY ABORT CHAIN EXECUTION
.               COMPILATION TIME COMMENT.   (Note that the //'s
                  are not present)
```

16.4.1 IF Directive

The IF directive has two variations, IFS and IFC, which are IF SET and IF CLEAR. The IFS directive proves positive if the tag named appeared on the CHAIN command line, and negative if the tag was omitted.

For example:

```
//IFS LIST
```

will prove positive if LIST was mentioned in the CHAIN command line, and negative if the tag does not exist, and

```
//IFC LIST
```

will prove positive if LIST was omitted and negative if it appeared on the CHAIN command line.

When an IF directive tests negative, it causes the chain compilation to skip all following lines of the procedure file until a directive is reached which clears the effect of the IF (an ELSE or XIF). When an IF directive tests positive it has no effect on the chain compilation. Normally the chain compilation is said to "include" lines from the procedure file; inclusion is inhibited by a negative evaluation of an IF directive.

Simple logical operations can be performed by IF directives. The tags to be used are separated by logical operators. The logical OR is indicated by '¦' (vertical bar) or ',' (comma). The logical AND is indicated by '&' (ampersand) or '.' (period). For example the following lines are in the file DOIT:

```
//IFS DATE&TIME¦QUICK     or     //IFS DATE.TIME,QUICK
DBCMP TEST;L                     DBCMP TEST;L
SAMPLE COMPILE                   SAMPLE COMPILE
```

If DATE and TIME or QUICK are defined on the CHAIN command line the DBCMP lines will be included in the work file.

```
     CHAIN DOIT;DATE=30NOV76,TIME=1500hr
or
     CHAIN DOIT;QUICK
or
     CHAIN DOIT;DATE,TIME
```

will all result in a true logical condition and the DBCMP lines will be included.

IF directives are only evaluated if lines are being included. If one IF directive has proven negative and has inhibited the inclusion of lines, all following IF directives will be ignored until either an ELSE or XIF statement is found. For example:

```
//IFS DATE
//IFS TIME
DBCMP TEST;L
SAMPLE COMPILATION
//XIF
```

If DATE was not defined, all lines until the //XIF will be ignored. In this example, if DATE were not defined the //IFS TIME statement would not be evaluated and the DBCMP TEST;L would not be included even if TIME was defined.

16.4.2 ELSE/XIF Directives

CHAIN has two directives that will alter the inclusion of lines from an IF directive. The first is the XIF directive. It will unconditionally terminate the range of the last IF directive. The second is the ELSE directive; it will reverse the results of the last IF directive; that is to say, if lines were being skipped because the last IF proved negative, an ELSE would cause lines to be included.

For example, the DOIT file contains the following lines:

```
//IFS LIST
DBCMP TEST;L
SAMPLE COMPILATION
//ELSE
DBCMP TEST
//XIF
//IFS TAPE
MOUT;D,30NOV76,V
TEST/DBC
*
//XIF
```

If CHAIN is invoked by 'CHAIN DOIT;LIST' the work file will contain

```
DBCMP TEST;L
SAMPLE COMPILATION
```

If invoked by 'CHAIN DOIT;TAPE', the work file will contain

```
DBCMP TEST
MOUT;D,30NOV76,V
TEST/DBC
*
```

16.5 Tag Value Substitution

A tag value is substituted whenever a pair of '#' symbols are found with a syntactically valid tag name between them. The value substituted is the tag value given in the CHAIN command line.

For example, contents of a file called DOIT:

```
DBCMP TEST;XL
TEST PROGRAM COMPILED ON #DATE# -- #TIME#
DBCMP #NAME#;XL
#NAME# PROGRAM COMPILED ON #DATE# -- #TIME#
```

If CHAIN is invoked by

```
CHAIN DOIT;TIME=2400hr,DATE=29NOV76,NAME=TEST2
```

the work file will contain

```
DBCMP TEST;XL
TEST PROGRAM COMPILED ON 29NOV76 -- 2400hr
DBCMP TEST2;XL
TEST2 PROGRAM COMPILED ON 29NOV76 -- 2400hr
```

If a tag is mentioned in the CHAIN command line but given no value and if the value is to be used for substitution, a null value is substituted for the #tag# within the line. The effect is that the #tag# characters disappear from the line. Continuing the above example, if CHAIN was invoked by

```
CHAIN DOIT;DATE=29NOV76,NAME=TEST2
```

the work file will contain

```
DBCMP TEST;XL
TEST PROGRAM COMPILED ON 29NOV76 --
DBCMP TEST2;XL
TEST2 PROGRAM COMPILED ON 29NOV76 --
```

16.6 BEGIN/END Directives

The BEGIN and END statements allow groups of IF/ELSE/XIF statements to be parenthesized. A counter called the BEGIN/END counter is initialized to zero when compilation of a procedure begins. If the use of procedural lines is turned off and a BEGIN operator is encountered, then the BEGIN/END counter is incremented. If an END operator is encountered, then the BEGIN/END counter is decremented unless it is already zero. The ELSE and XIF operators have no effect if the BEGIN/END counter is not equal to zero. For example:

```
//IFS FLAG1
DBCMP TEST1;XL
TEST PROGRAM ONE
//ELSE
//BEGIN
//IFS FLAG2
DBCMP TEST2;XL
TEST PROGRAM TWO
//ELSE
DBCMP TESTTEST;XL
TEST TESTER
//XIF
//END
//XIF
//IFS FLAG3.FLAG27
LIST SCRATCH;L
THE SCRATCH FILE AT FLAG 27
//XIF
```

The 6th through the 12th lines will not be used if FLAG1 exists, notwithstanding the fact that there is an ELSE and XIF operator within those lines, because the BEGIN/END pair prevented these statements from having any effect.

16.7 ABORT Directives

The //ABORT statement will cause CHAIN to return to DOS if it is processed. For example:

```
//IFC TIME¦DATE
.**** TIME AND DATE ARE BOTH REQUIRED
//ABORT
//XIF
.
.
.
```

If the procedure file is invoked with TIME or DATE missing, the error message comment line would be displayed, and the compilation of the input file would ABORT.

The //ABTIF statement will conditionally cause the execution phase of CHAIN to ABORT. This statement causes DOSFLAG to be examined and if bit 7 (ABTIF) is on, the chaining will abort. Bit 7 of DOSFLAG is the abnormal program completion bit. If non-fatal errors have been found during the execution of the last program

the ABTIF bit should be set. For example, the procedure file contains:

```
ABTONOFF OFF
KILL TESTFILE/CMD
Y
//ABTIF
KILL OUTPUT/TXT
Y
.
.
```

If the file TESTFILE/CMD is not found by KILL, it will set the ABTIF bit. When the //ABTIF statement is processed the abnormal program completion bit will be checked, and in this case it will be on, so the CHAIN will be aborted.

The ABTONOFF command should always be used to turn the ABTIF bit off prior to execution of a program which will be tested using //ABTIF. Once ABTIF is set on by some error, it is not cleared except by ABTONOFF or by an abort caused by an //ABTIF directive.

16.8 Comments

CHAIN allows for two types of comment lines within the procedural file. One type is the execution time comment. This type may appear only before a DOS command entry and will not appear until just before that command is to be executed. An execution time comment can appear only just before a command because at any other place in a procedure file, the comment would be presented as keyboard response to an executing program. Comments can be placed at the end of a procedure, since this location is equivalent to immediately prior to a command. For example, the procedure file containing:

```
//. COMPILATION OF THE TEST PROGRAM
DBCMP TEST;XL
TEST PROGRAM
```

would cause the first line to be displayed before the assembly was executed. A variation on the execution time comment is the operator break point. For example, the procedure file containing:

```
//* INSERT TAPE Z12548 INTO THE FRONT CASSETTE DECK
MOUT ;LV
TEST/TXT
DATA/TXT
*
```

would cause a BEEP and the first line to be displayed. At this point the machine would wait for the operator to depress either the KEYBOARD or DISPLAY key and then continue with the MOUT process.

The second type of comment line is a compilation time comment. This line is not included in the work file but is displayed on the screen immediately after it is read from the procedural file. This is useful in communicating to the operator what procedure is about to be followed by CHAIN.

Both types of comment lines will be ignored (not displayed or written) just as other procedure lines if a test has proven negative and an ELSE or XIF operator has not been reached. For example, if the following procedure file MAKETEST was created:

```
. COMPILATION OF TEST PROGRAM
//IFS LIST
. YOU ARE GOING TO GET A LISTING
DBCMP TEST;XL
TEST PROGRAM
//ELSE
. YOU AREN'T GOING TO GET A LISTING
DBCMP TEST
```

and the CHAIN command:

```
CHAIN MAKETEST;LIST
```

was given, then only the lines:

```
. COMPILATION OF TEST PROGRAM
. YOU ARE GOING TO GET A LISTING
```

will appear on the screen before the procedure is executed. If, however, the CHAIN command:

```
CHAIN MAKETEST
```

was given, then only the lines:

. COMPILATION OF TEST PROGRAM
. YOU AREN'T GOING TO GET A LISTING

will appear on the screen before the procedure is executed.

16.9 Complex CHAIN Examples

The chapter on the AUTOKEY command contains an example of the use of AUTO and AUTOKEY combined with the use of CHAIN directives using tag existence testing to set checkpoints for automatic restart of a lengthy automated procedure. The example below uses BUILD within a chain procedure to create a procedure file for later execution by another chain. It uses several tags for both existence testing and value substitution.

The procedure file below, "RUNTEST", is part of a series of CHAIN procedures for program generation and testing. RUNTEST builds a procedure file for program compilation; the resulting procedure file would be run by a later CHAIN.

RUNTEST recognizes several tags:

PLUS - mention of this tag indicates the compilation should use the DBCMPLUS compiler instead of the older DBCMP compiler.

XTR - mention of this tag causes use of the additional list output commands (C and R) available in DBCMPLUS.

FLAG - the substitution value for this tag will be tag existence tested for list control on the output procedure file.

PROG - the substitution value for this tag will be a tag to provide program name in the output procedure file.

DATE - the substitution value for this tag will provide the compilation date in the output procedure file.

RUNTEST contents:

```
.
. TEST FOR DBCMPLUS COMPILER FLAG
.
//IFC PLUS
//BEGIN
.
. BEGIN PROCEDURE FOR DBCMP COMPILATION
.
BUILD COMPIT;!
.
. NOTE HOW BEGINNING INPUT LINE TO BUILD/CMD WITH THE TERMINATION CHARACTER
. ALLOWS ENTERING CHAIN COMMANDS TO THE OUTPUT FILE.  THE LINE IMMEDIATELY
. BELOW IS WRITTEN OUT AS "//IFS #FLAG#"; IF IT HAD NOT BEGUN WITH "!", IT
. WOULD HAVE BEEN INTERPRETED AS A CHAIN DIRECTIVE FOR THE CURRENT CHAIN.
.
!//IFS #FLAG#
!//* COMPILATION LISTING  -  BE SURE PRINTER IS READY
DBCMP ##PROG##;LX
##PROG## COMPILATION      #DATE#
!//ELSE
DBCMP ##PROG##
!//XIF
!
//END
.
. THIS "//ELSE" INSTRUCTION REVERSES THE EFFECT OF THE "//IFC PLUS" ABOVE
.
//ELSE
//BEGIN
.
. BEGIN PROCEDURE FOR DBCMPLUS COMPILATION USING OPTIONS OF DBCMPLUS
. BASED ON "XTR" FLAG.
. THE "BEGIN" ABOVE CAUSES THE "XIF"S AND "ELSE"S IN THE FOLLOWING SECTION
. TO AFFECT ONLY DIRECTIVES AT THE SAME BEGIN/END LEVEL, AND NOT THE
. "//ELSE" DIRECTIVE ABOVE, WHICH CONTROLS THE ENTIRE "PLUS" CONDITIONAL
. SECTION.
.
BUILD CMPLIT;!
!//IFS #FLAG#
!//* COMPILATION LISTING  -  BE SURE PRINTER IS READY
.
. THE FOLLOWING DIRECTIVES ARE RECOGNIZED DURING CHAIN COMPILATION AND
. CONTROL SELECTION OF LINES TO FOLLOW THE BUILD COMMAND ABOVE.
.
//IFS XTR
```

```
DBCMPLUS ##PROG##;LXCR
//ELSE
DBCMPLUS ##PROG##;LX
//XIF
##PROG## COMPILATION #DATE#
!//ELSE
DBCMPLUS ##PROG##
!//XIF
!
.
. PROCEDURE IS EFFECTIVELY FINISHED AT THIS POINT, BUT IT IS ESSENTIAL
. PROVIDE AN "END" DIRECTIVE TO MATCH THE UNMATCHED "BEGIN" ABOVE, AND
. AN "XIF" TO TERMINATE THE "ELSE" IMMEDIATELY PRIOR TO THE "BEGIN".
.
//END
//XIF . . END OF RUNTEST SAMPLE FILE .
.....................................................................
.
```

Entering the command

```
CHAIN RUNTEST;PLUS,XTR,FLAG=LIST,PROG=NAME,DATE=21OCT78
```

produces a procedure file CMPLIT/TXT with the following contents:

```
//IFS LIST
//* COMPILATION LISTING  -  BE SURE PRINTER IS READY
DBCMPLUS #NAME#;LXCR
#NAME# COMPILATION 21OCT78
//ELSE
DBCMPLUS #NAME#
//XIF
```

Entering the command

```
CHAIN RUNTEST;FLAG=PRINT,PROG=PROG,DATE
```

produces a procedure file COMPIT/TXT with the following contents:

```
//IFS PRINT
//* COMPILATION LISTING  -  BE SURE PRINTER IS READY
DBCMP #PROG#;LX
#PROG# COMPILATION
//ELSE
DBCMP #PROG#
//XIF
```

16.10 Resuming An Aborted CHAIN

Before the CHAIN overlay fetches the next DOS command it stores in the CHAINP/SYS file pointers for the line to be used. If something goes wrong during the DOS command which follows and the procedure is aborted, CHAIN still knows where it was in the CHAINP/SYS file when the problem occurred. Since CHAIN does not delete the CHAINP/SYS file unless the procedure completes successfully, it can pick up where it stopped in the CHAINP/SYS file if the operator can correct the condition which caused the procedure to abort in the first place. Often, the reason for the abort is something correctable like the disk running out of files. In this case, the operator need only correct the condition and then enter:

```
CHAIN *
```

and the procedure will pick up with the command which failed before. This action can generally be applied even if the RESTART key has been depressed. Thus, one can recover from jammed paper in a printer half way through a listing by simply depressing RESTART, fixing the printer, and then entering the CHAIN * command.

If the failing command cannot ever succeed, it may be bypassed by entering the command:

```
CHAIN/OV1
```

This simply restarts the chain with the next available line in the procedure. If the next line had been intended as a keyin line for the failed program (as opposed to a DOS command line) the chain will generally immediately abort again. However, by restarting the chain in this manner, repeatedly if necessary, the invalid step can usually be bypassed and chaining resumed. Use of CHAIN/OV1 will not always work, since it depends on information in processor memory to function. If the area from MCR$+80 to MCR$+100 is disturbed, CHAIN/OV1 will fail, usually causing a range error or perhaps a system data failure.

16.11 CHAIN Programming Considerations

CHAIN only replaces the DOS keyboard entry routine (KEYIN$). Therefore, only programs that use this routine for input will receive their input from the chain file. Programs which have their own input routines, like the DOS editor, can be invoked from a chain file but editing must be done manually by the operator. Sometimes programs will use a different keyin routine based on DOS Function 6 to request operator action for special circumstances when it is desired to avoid using lines from the chain procedure.

When a program exits via EXIT$ or NXTCMD the chain continues normally. If a program exits via ERROR$ or CMDAGN the chain is aborted. Generally the terminating error message displayed by an aborting program will remain visible on the screen following the CHAIN abort.

Some programs can go through a rather complex set of requests for input, which can make them difficult to use with the CHAIN program. For this reason, most DOS programs allow almost all options to be specified on the command line and keep the variation in the number of keyin requests to a minimum. It is good practice for all programs to be written with this concern in mind to facilitate their use with CHAIN.