FASTBUS Diagnostic Language*
Users Manual

Version 3(74)

David Lesny
July, 1983

DO NOT MICROFILM
COVER

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

DEPARTMENT OF PHYSICS

LOOMIS LABORATORY OF PHYSICS

1110 W. GREEN STREET

URBANA, ILLINOIS 61801

MASTER

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency Thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.

DOE/ER/01195--478

DE83 017932

FASTBUS Diagnostic Language*
Users Manual,

Version 3(74)

David Lesny
July, 1983

University of Illinois at Urbana-Champaign
Department of Physics
High Energy Physics Group
Urbana, Illinois

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

* Work supported in part by the Department of Energy,
contract DE-AC02-76ER01195

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

EDB

Pacs Index: 06.50.Mk

Table of Contents

Preface

CHAPTER 1	INTRODUCTION	
1.1	FDL FEATURES	1-2
1.2	HOW TO USE THIS MANUAL	1-3
CHAPTER 2	USING FDL	
2.1	STATEMENTS	2-2
2.2	PARSING	2-3
2.3	INTERPRETER	2-5
CHAPTER 3	VARIABLES, CONSTANTS AND EXPRESSIONS	
3.1	ALLOCATE	3-2
3.2	DEALLOCATE	3-4
3.3	ASSIGNMENT STATEMENT	3-5
3.4	TYPE	3-20
3.5	RANDOM	3-24
CHAPTER 4	FASTBUS COMMAND STATEMENTS	
4.1	COMPLETE OPERATIONS	4-2
4.2	PRIMITIVE OPERATIONS	4-11
4.3	LIST OPERATIONS	4-19
4.4	SYSTEM VARIABLES	4-20
4.5	ERR\$STATUS SYSTEM VARIABLE	4-21
4.6	FASTBUS MESSAGES	4-22
4.7	HOLD BUS ON ERROR	4-25
4.8	SYSTEM PARAMETER BLOCK	4-26
4.9	RESET	4-28
CHAPTER 5	I/O STATEMENTS	
5.1	OPEN	5-2
5.2	CLOSE	5-3
5.3	WRITE AND WRITELN	5-4
5.4	READ	5-5
5.5	MODE	5-6
5.6	COMPARE	5-7
5.7	MAP	5-9

CHAPTER 6	PROCEDURES	
6.1	PROCEDURES	6-2
6.2	ALLOCATE	6-4
6.3	DEALLOCATE	6-5
6.4	PROCEDURE PARAMETERS	6-6
6.5	BRANCHING	6-8
6.6	RETURN	6-15
6.7	PROMPT	6-15
6.8	RECURSIVE CALLS	6-16
6.9	CONTINUE	6-17
CHAPTER 7	UTILITIES	
7.1	DIRECT	7-2
7.2	DELETE	7-3
7.3	INDIRECT FILES	7-4
7.4	SAVE	7-8
7.5	WAIT	7-9
7.6	EXIT	7-9
7.7	COMPRESS	7-10
7.8	CREATE	7-11
7.9	PURGE	7-12
7.10	SHOW	7-13
CHAPTER 8	EDITOR	
8.1	INVOKING THE EDITOR	8-2
8.2	EDITOR COMMANDS	8-3
8.3	EDITOR FILES	8-11
APPENDIX A	RESERVED NAMES	
APPENDIX B	COMMAND SYNTAX SUMMARY	
APPENDIX C	KNOWN PROBLEMS AND BUGS	
APPENDIX D	FUTURE ADDITIONS TO FDL	
APPENDIX E	EXAMPLES OF PROCEDURES	

APPENDIX F

ERROR MESSAGES

Index

Preface

FASTBUS Diagnostic Language (FDL) was conceived and designed by Keith Nater in the summer of 1980 while working as a research assistant for the High Energy Physics Group at the University of Illinois. FASTBUS was still in its infantile stages in 1980 and very little hardware and software existed. Mr. Nater hoped to develop a program which could be used to debug hardware in a FASTBUS system. A design goal was to produce a language which would be straight forward in use. It should not require a software engineer to write a special test program for each new hardware module. Also, hardware engineers and technicians should find it "user friendly".

The first description of FDL was written by Mr. Nater in 1980 and distributed to the members of the FASTBUS Software Working Group for comments. Copies were also given to the various engineers who would use such a program. Mr. Nater had previously used the CAMAC Diagnostic Language (CDL), developed at Fermi National Laboratory (FNAL), and the initial description was based roughly on CDL. After several revisions of the document, coding on FDL began early in 1981.

The first version of FDL was to be run on PDP-11 based computer systems under the RT11 operating system. These systems were in wide use at UIUC, FNAL and other national labs and universities. At the same time, the first version of the FASTBUS Standard Routines were being written at FNAL for use with the UNIBUS Processor Interface (UPI) under RT11. FDL was to be written in PASCAL. It was hoped that by writing in PASCAL, FDL might be easily transported to other computer systems.

By October of 1981, FDL was partially written. Mr. Nater, who was now using FDL as his masters thesis project, presented FDL as a poster session at the "Topical Conference on Computerized Data Acquisition in Particle and Nuclear Physics", Oak Ridge, Tennessee, May 28-30, 1981. The accompanying article on FDL was published in the "IEEE Transaction on Nuclear Science", October 1981, volume NS-28. Mr. Nater shortly thereafter received his masters degree in Computer Science and left the university for employment in private industry. Another student, Seth Abraham, took over the FDL project.

Mr. Abraham worked on FDL until his departure in the summer of 1982. During this time the first working version of FDL came to being and was used by the engineers at UIUC

for debugging FASTBUS modules for FNAL Experiment 401. FDL, though not complete in its programming, played a very useful role in debugging and diagnosing hardware problems in this first FASTBUS experiment. With the departure of Mr. Abraham in 1982, I took over the FDL project.

Since taking over the project, I have made many changes and additions to FDL. Working closely with the engineers at UIUC while debugging the first versions of Segment Interconnects (SI), I learned of various problems and deficiencies with the first release of FDL. Suggestions made by engineers at both UIUC and FNAL were incorporated into the language. Though still conceptually and functionally the same as the FDL created and designed by Mr. Nater in 1980, FDL now has a more complete command set. Extended memory on larger PDP-11 systems is utilized for variable space. Disk files are used to increase the storage capacity of various internal tables. This "Users Manual" is a complete rewrite of the original written by Mr. Nater. Examples are used extensively to help explain each command and feature. FDL is still not complete. Continued use of FDL will lead to more suggestions for improvements, bug fixes and additional functionality. However the current release is a product which is very usable and reliable.

I would like to thank everyone who was involved in the FDL project. A special thanks goes to Jerry Wray for his comments, criticisms and proof reading of this document. Thanks also goes to Ruth Pordes for her help in debugging my various releases of FDL and also acting as an intermediary between myself and the FNAL staff.

David Lesny
April, 1983

CHAPTER 1

INTRODUCTION

FASTBUS Diagnostic Language (FDL) is an interactive, interpretive language designed to aid the engineer or physicist/user in the testing and debugging of FASTBUS modules and systems. Since FASTBUS systems involve a variety of devices and data paths, it is frequently more efficient to utilize a high-level language system such as FDL for diagnostics, rather than to develop device-specific programs. FDL can also be used to a limited extent for both device control and data acquisition.

FDL statements and declarations, which are PASCAL-like in construct, can be entered from the terminal or stored in a disk file and submitted to FDL by a single user request. FDL is also conversational; hence, one can quickly develop and modify programs on-line.

FDL is designed to operate on any PDP or LSI-11 processor using the RT-11 Single Job operating system, Version 3B or later. FDL is written in PASCAL using the Oregon Software Pascal-2 compiler. A minimum of 28K words of memory is required. If more than 28K words are available, FDL can optionally use this extended memory space for variable storage. FDL will run from a floppy based disk system; however, since the program uses overlay techniques, a faster disk system is recommended. FDL interfaces to the FASTBUS system via the FASTBUS Standard Routine package. In principle any FASTBUS master interface which connects a PDP-11 system to FASTBUS can be used with FDL. To date only the UNIBUS Processor Interface (UPI) has been used with FDL. The I/O Register to FASTBUS Interface (IORFI) has also been used with an emulation package which makes the IORFI appear to be a UPI.

1.1 FDL FEATURES

Among the features available in FDL are the abilities to:

- o Perform any FASTBUS cycle either within a complete FASTBUS transaction or as a primitive operation. The cycles are arbitration, primary address, secondary address and single or block data transfers (read or write). Pipelined data transfers are not supported.
- o Allocate memory locations (variables) for data storage which can be referenced with symbolic names. These variables can be either single cells or single-dimensioned arrays. All variables are 32-bit signed, 2's complement integers.
- o Assign data values to variables and perform basic integer arithmetic operations - add, subtract, multiple and divide.
- o Display the contents of variables on the terminal in four different radices - decimal, hexadecimal, octal or binary.
- o Form named procedures consisting of FDL statements (FASTBUS cycles, variable assignment, etc). When invoked, FDL executes the statements contained within the procedure. It is possible to create loops within procedures for repetitive operations.
- o Perform simple disk input and output for storing and retrieving data in a disk file.

1.2 HOW TO USE THIS MANUAL

This manual consists of a table of contents, eight chapters, six appendices and an index. Each chapter begins with a brief description of the various sections contained in the chapter. Each section describes an FDL command or function in detail. The syntax of the command is given along with several examples. In the examples, FDL reserved words (commands, keys, etc.) are always printed in upper case letters. User defined names are always printed in lower case letters.

First time users should read chapter 2, "Using FDL", before any other chapters (except the introduction) so as to understand how FDL executes a command. Once the reader has become familiar with the various commands, the appendix "Command Summary" is useful as a quick reference for command syntax and functionality.

This manual was written assuming the reader was familiar with FASTBUS. The technical details on FASTBUS can be found in the document "FASTBUS, Modular High Speed Data Acquisition System". This document can be obtained from the National Bureau of Standards, Washington D.C.

CHAPTER 2

USING FDL

When the user initiates execution of the FDL program, FDL announces itself and then initializes itself and the FASTBUS interface. When initialization is complete, FDL issues the message "Complete" and a command prompt. For example:

```
FASTBUS Diagnostics Language    Version 3(74)
Initialization . . .
Complete
```

*

FDL is now ready for input from the terminal. The user can enter lines of up to 80 characters in length. FDL will accept input in either upper or lower case. Any mixture of upper and lower case characters is acceptable.

FDL operates in two different phases. Parsing, the first phase, begins when the user inputs a line of text to FDL and types a carriage return. The statement is checked for syntax errors, and if there are none, a series of instructions are generated and passed onto the second phase, interpretation. The interpretation phase actually performs the action specified by the user in the input line. Should any syntax errors be discovered during parsing, no interpretation takes place.

2.1 STATEMENTS

Statements may be entered on the terminal after the prompt is issued. Statements must be separated from each other by a semicolon. Multiple statements per input line are allowed. There are two statement types in FDL:

```
* verb/switch(s) arglist;
```

and

```
* variable := expression;
```

The first statement type, a command statement, performs the action specified by "verb". A "verb" is any legal FDL command, all of which are defined in later chapters. The action of the command may be modified by one or more "/switch(s)". The "arglist" is a group of one or more arguments, separated by commas, which provide data needed by the command. Not all commands require switches or arguments.

The second statement type, an assignment statement, allows the user to assign values to variables defined with the ALLOCATE command. This statement is discussed in chapter 3, "Variables, Constants and Expressions".

2.2 PARSING

Parsing of the input line begins when the user types a carriage return. Should FDL find an error while parsing a statement, parsing is stopped and a message is output to the terminal indicating the problem. An uparrow, "^", points to the location where the error was detected.

Example:

```
* ALLOCATE a[1..10,b;  
^  
>>>Expecting a ]  
*
```

In the above example, the user attempted to allocate memory space for an array called "a", with an index range from 1 to 10. However the range definition of the array was not completed with the right square bracket, "]". The uparrow points at the position where FDL expected the bracket. All input on the line following the location of the error is ignored.

Statements can be continued on multiple lines of input. If a carriage return is typed before a semicolon is entered, FDL will parse what has been entered. If no syntax errors are found, FDL outputs the continuation prompt, "-", and waits for more input.

Example:

```
* ALLOCATE a[1..10],  
- b;  
*
```

After parsing the ALLOCATE command and the argument, FDL outputs the continuation prompt since no semicolon was seen. The user responded with a second argument and terminated the statement.

Comments can be entered at any point in the input stream. Comments are preceeded by a double minus, "--". FDL ignores all input between the double minus and the end of the line.

Example:

```
* ALLOCATE a[1..10];           -- allocate an array
* -- This is a comment line which is ignored by FDL
*
```

Statements must always begin with either a command name or a variable, which are called tokens. These tokens are used to determine which of the two statement types the user has entered. If the token is a command name, parsing assumes a command statement. If the token is not a command name, FDL assumes the statement to be an assignment statement. Should the token be neither a command nor a variable, FDL stops parsing and types an error message.

Example:

```
* 123456;
^
>>>Expecting a variable name
*
```

If the user mistypes a command name, FDL will assume an assignment statement is being entered and a syntax error will result.

Example:

```
* ALOCATE a,b,c;
^
>>>Expecting :=
*
```

In the above example the user meant to type the command "ALLOCATE" but FDL was processing an assignment statement using a variable called "ALOCATE".

If the user is entering an assignment statement and mistypes the variable name, FDL may parse the statement without detecting any errors. However FDL will issue an error message during the interpretation phase if the misspelled variable name has not been previously allocated.

Example:

```
* a := 1;
>>>Variable not defined
>>>Variable: A
*
```


2.3 INTERPRETER

Once FDL has parsed a statement and detected no errors, control passes to the interpretation phase. Instructions generated by the parser are to the interpreter. These instructions tell the interpreter what action is to be taken in response to the statement entered by the user. When the interpreter has completed execution, FDL returns to the parser phase and waits for more input.

CHAPTER 3

VARIABLES, CONSTANTS AND EXPRESSIONS

One of the basic features available to the user in FDL is the ability to dynamically allocate memory locations for data storage. These locations, or variables, are given symbolic names by the user for easy reference. Two types of variables are supported, single words and one dimensional arrays. The only data type supported by FDL is 32-bit, signed integers.

Variables can be assigned values with the assignment statement. Thirty different operators are available for use in expressions. Complex expressions can be formed using parentheses to indicate a priority of operation. In addition to the arithmetic operators add, subtract, multiply and divide, there are also operators for performing bit manipulations (rotation, shifting, masking) and logical comparisons (less than, greater than, equal to).

3.1 ALLOCATE

Variables are defined using the ALLOCATE command. ALLOCATE takes as its arguments a list of names which will be the symbolic names of the newly defined variables. Each name can be from 1 to 10 alphanumeric characters ("A" to "Z" and "0" to "9") in length. The name must begin with an alpha character. The characters "\$" and "" can also be used in names. The "\$" can be used anywhere an alpha character is legal; the "" can be used anywhere a numeric character is legal. Any name can be used except those in the reserved name list. Reserved names are FDL commands, system variables and key words. A list of reserved names is given in the appendix "Reserved Names".

Variable types, single or array, are implicitly defined by the syntax. If the variable name is followed by an index range, the variable will be an array. If not, the variable will be a single word. An index range defines the bounds of the array. It consists of two constants or variables enclosed in square brackets which are separated by a double period, "..". If only a single constant or variable is given in square brackets, the range is assumed to be from zero to the given number.

Syntax:

```
ALLOCATE variable;  
ALLOCATE array[first..last];  
ALLOCATE array[last];
```

Example:

```
* ALLOCATE a;  
* ALLOCATE b[1..100],c[30..10]  
* ALLOCATE d[20];  
*
```

The above example defines four different variables. Variable "a" is a single word while "b", "c" and "d" are arrays. Variable "b" has an index range from 1 to 100 and uses 100 memory locations. Variable "c" has an index range from 30 to 10 and uses 21 memory locations. Variable "d" has an index range from 0 to 20 and uses 21 memory locations.

NOTE

The initial contents of variables after allocation is unpredictable. The user should not rely on the contents of variable to be zero nor any other value. The variables should be manually initialized with the assignment statement.

There is limited memory which FDL can use for variables. The exact amount of memory available for variable allocation is an installation dependent parameter. The amount available can be display via the SHOW STATUS command which is described in chapter 7. If there is insufficient memory available for the allocation, FDL will issue a message indicating how many words are available. The specified variable is not allocated.

Example:

```
* ALLOCATE buffer[0..07FFFFFFh];
>>>Insufficient memory for allocation
>>>Variable: BUFFER
>>>Only 512 words available
*
```

Should this occur, the user can free memory by deleting variable allocations with the DEALLOCATE command.

3.2 DEALLOCATE

The memory locations used by variables can be returned to the free pool by using the DEALLOCATE command. DEALLOCATE can be used to remove definitions for all or only selected variables. The arguments for DEALLOCATE are a list of names of the variables to be deallocated. The ranges of arrays need not be given. The key word ALL can be used to deallocate all allocated variables.

If an attempt is made to deallocate a variable which has never been allocated, FDL outputs an error message and aborts processing. Likewise, if an attempt is made to deallocate a system variable, an error results.

Syntax:

```
DEALLOCATE <variable list>;
DEALLOCATE ALL;
```

Example:

```
* ALLOCATE a,b,c;           -- allocate some variables
* DEALLOCATE a,b;          -- deallocate only a and b
* DEALLOCATE ALL;          -- deallocate all variables
* DEALLOCATE a;
>>>Attempt to deallocate non-existent variable
* DEALLOCATE ERR$STATUS;
>>>Cannot allocate/deallocate system variables
*
```

3.3 ASSIGNMENT STATEMENT

Once a variable has been defined by the ALLOCATE command, the user can assign a value to the memory location with an assignment statement. Assignment statements are of the form,

```
* variable := expression;
```

where "variable" is the symbolic name of a previously allocated variable and "expression" any combination of constants, variables and operators.

The expression may involve only a single constant or variable or it may be a complex expression involving multiple operators, constants and/or variables. Each operator is assigned a priority which is used by FDL in evaluating a complex expression. If ambiguity results from two or more operators having the same priority, FDL will evaluate (that section) of the expression from left to right. Parentheses can be used within an expression to change the order of evaluation. Parentheses must be balanced. That is, for every left parenthesis, "(", there must also be a matching right parenthesis, ")".

Example:

```
* ALLOCATE a,b,c,d;           -- allocate variables
* a := 0;                     -- zero a
* b := 1 + 2 + 3;             -- assign b the value 6
* c := 100 * b + a;           -- assign c the value 600
* d := 10 * (c DIV b)         -- assign d the value 1000
*
```

Array assignments require that an index be given with the name of the variable. The index must be within the defined bounds or an error message is output.

Example:

```
* ALLOCATE a[1..10],b;        -- allocate some variables
* a[1] := 0;                  -- zero location 1
* b := 10;                    -- assign the value 10
* a[2] := a[1];               -- copy location 1 to 2
* a[b] := b;                  -- assign 10 to a[10]
* a[0] := -1;                 -- bounds error
>>>Subscript out of range
>>>Variable: A
*
```

Values may be assigned to multiple locations of an array in a single statement. This is done by using an index range on the array and a range expression. A range expression consists of 1 or more constants or variables separated by commas and contained between square brackets ([1,2,3,4,...,n]). The array assignment is done in the order (ascending or descending) specified by the index range. For each index a value is retrieved from the range expression. If there are not enough arguments in the range expression to fill all the array locations given by the index range, the range expression is repeated.

Example:

```
* ALLOCATE a[1..10],b;           -- allocate some variables
* a[1..4] := [1,2,3,4];         -- assign a[1..4], 1 to 4
* a[4..1] := [1,2,3,4];         -- assign a[4..1], 1 to 4
* a[1..10] := [0];              -- zero array a
* b := 10;                      -- assign b the value 10
* a[1..3] := [b];               -- assign b to a[1..3]
* a[1..b] := [1];               -- assign 1 to a[1..b]
*
```

3.3.1 Constants

FDL supports constants in four different radices - decimal, hexadecimal, binary and octal. Constants are assumed to be decimal unless the user indicates otherwise by adding a single letter suffix to the constant. The suffices are:

b	binary
o	octal
h	hexidecimal

The following constants are equal to ten decimal:

10	12o	0Ah	1010b
----	-----	-----	-------

NOTE

All constants must begin with a numeric character. Since the hexadecimal radix uses the characters A thru F, the user must take care to begin the constant with at least one zero (0) digit. If the constant does not begin with a numeric digit, FDL will interpret the constant as an undefined variable.

Example:

```
* ALLOCATE a;           -- allocate the variable a
* a := 10;              -- assign a 10 decimal
* a := 12o;             -- assign a 12 octal
* a := 0Ah;             -- assign a 0A hexadecimal
* a := 1010b            -- assign a 1010 binary
* a := Ah               -- bad constant
>>>Variable not allocated
>>>Variable: AH
* a:= 0Fb               -- bad character in constant
>>>Invalid character in constant
*
```


3.3.2 Operators

FDL supports thirty operators which can be combined in any order to form expressions. Each operator acts upon one or two operands and forms a result which can be assigned to a variable. When combined in an expression, operators are executed in an order based on a fixed priority. This priority scheme can be overridden by the use of parentheses. When two operators of the same priority occur in the same sub-expression, FDL evaluates that sub-expression from left to right.

FDL supports two types of operators, arithmetic and logical. Arithmetic operators return a 32 bit signed integer. The arguments for arithmetic operators are always treated as 32 bit signed integers. Logical operators always return a true or false result. A true result is a -1 value (all bits on). A false result is a 0 value (all bits off). The arguments for logical operators can be either 32 bit signed integers, 32 bit unsigned integers, or logical values. A value is considered logically true whenever it is a negative number (the sign bit is set). A value is considered logically false whenever it is a positive number (the sign bit is clear).

Single operand operators are of the form

<operator> <operand>

Example:

```
NOT 10
COM 15670o
SWAP OFFFFAAAAh
```

Double operand operators are of the form

<operand 1> <operator> <operand 2>

Example:

```
a + b
10 DIV 2
200h ROTATE 20
```

Parentheses can be used to enclose sub-expressions.

Example:

```
(a + b) * c;
((a + b) DIV (c + d))
NOT (a * b)
```

The supported operators, their priorities relative to each other, the number of operands required and the operator type are given in the following table.

operator	priority	operands	type
NOT	1	1	logical
SWAP	1	1	arithmetic
COM	1	1	arithmetic
Unary minus	1	1	arithmetic
Unary plus	1	1	arithmetic
*	2	2	arithmetic
DIV	2	2	arithmetic
MOD	2	2	arithmetic
+	3	2	arithmetic
-	3	2	arithmetic
SHIFT	3	2	arithmetic
ROTATE	3	2	arithmetic
LT	4	2	logical
LE	4	2	logical
GT	4	2	logical
GE	4	2	logical
EQ	4	2	logical
NE	4	2	logical
<	4	2	logical
<=	4	2	logical
>	4	2	logical
>=	4	2	logical
=	4	2	logical
<>	4	2	logical
AND	5	2	logical
OR	5	2	logical
XOR	5	2	logical
&	5	2	arithmetic
!	5	2	arithmetic
#	5	2	arithmetic

3.3.2.1 Logical Negation - (NOT operand)

The NOT operator returns the logical negation of its operand.

Example:

expression	result
-----	-----
NOT -1	0
NOT 0	-1
NOT 2	-1
NOT -2	0

3.3.2.2 SWAP - (SWAP operand)

The SWAP operator returns the swapped 16-bit half words of its operand.

Example:

expression	result
-----	-----
SWAP 0Fh	0F0000h
SWAP 0FFFF0000h	0FFFFh
SWAP 0FFFFFFFFh	0FFFFFFFFh

3.3.2.3 Ones Complement - (COM operand)

The COM operator returns the ones complement of its operand.

Example:

expression	result
-----	-----
COM 0Fh	0FFFFFF0h
COM 0FFFF0000h	0FFFFh
COM 0FFFFFFFFh	0
COM 25252525252o	12525252525o

3.3.2.4 Unary Minus - (- operand)

The Unary minus (-) is used to indicate a negative number. It returns the twos complement of its operand.

Example:

expression	result
-----	-----
-1	-1
-0	0
-0FFFFFFFh	1h
-0FFFF0000h	10000h
-25252525252o	12525252526o

3.3.2.5 Unary Plus - (+ operand)

The Unary plus (+) is used to indicate a positive number. By default, all constants (in the decimal radix) will be treated as a positive number. This operator is provided for completeness.

3.3.2.6 Multiply - (operand 1 * operand 2)

The * operator returns the result of multiplying operand 1 by operand 2.

Example:

expression	result
-----	-----
100 * 2	200
0FFh * 3h	2FDh
2525o * 2525o	7067071o
10010b * 1011b	11000110b

3.3.2.7 Divide - (operand 1 DIV operand 2)

The DIV operator returns the quotient of operand 1 divided by operand 2.

Example:

expression	result
-----	-----
200 DIV 2	100
2FDh DIV 0FFh	3h
7067071o DIV 2525o	2525o
11000110b DIV 1011b	10010b

3.3.2.8 Modulo - (operand 1 MOD operand 2)

The MOD operator returns the remainder of operand 1 divided by operand 2, the modulo. The result always takes the sign (positive or negative) of operand 1.

Example:

expression	result
-----	-----
1234 MOD 10	4
2525o MOD 45o	41o
0FFFFh MOD 2h	1h
-0FFFFh MOD 2h	0FFFFFFFh
-0FFFFh MOD -2h	0FFFFFFFh

3.3.2.9 Add - (operand 1 + operand 2)

The + operator returns the result of adding operand 1 to operand 2.

Example:

expression	result
-----	-----
1 + 2	3
0CD21Bh + 0F2A0h	0DC4BBh
034525o + 66523o	123250o
110110b + 101110b	1101100b

3.3.2.10 Subtract - (operand 1 - operand 2)

The - operator returns the result of subtracting operand 2 from operand 1. It may also be used to indicate a negative number or the two's complement of a number in a radix other than decimal.

Example:

expression	result
-----	-----
3 - 1	2
0DC4BBh - 0F2A0h	0CD21Bh
123250o - 034525o	123250o
1101100b - 101110b	110110b
0FFh - -3h	102h

3.3.2.11 Bit Shift - (operand 1 SHIFT operand 2)

The SHIFT operator returns the result of binary shifting operand 1 by the number of bits specified by operand 2. If operand 2 is positive, the shift is to the right (the most significant bit moves towards the least significant position). If operand 2 is negative, the shift is to the left (the least significant bit moves towards the most significant position). As the shift takes place, bits which move beyond the least or most significant positions in the 32 bit word are discarded. A 0 bit is always carried into the end bit position vacated by the shift.

Example:

expression	result
-----	-----
100110b SHIFT 1	10011b
100110b SHIFT 3	100b
100110b SHIFT -3	100110000b
0FFFFFFh SHIFT -10h	0FFFF0000h

3.3.2.12 Bit Rotation - (operand 1 ROTATE operand 2)

The ROTATE operator returns the result of binary rotating operand 1 by the number of bits specified by operand 2. If operand 2 is positive, the shift is to the right (the most significant bit moves towards the least significant position). If operand 2 is negative, the shift is to the left (the least significant bit moves towards the most significant position). As the rotation takes place, bits which move beyond the least or most significant positions in the 32-bit word are carried into the bit position at the opposite end of the word.

Example:

expression	result
-----	-----
100110b ROTATE 1	10011b
100110b ROTATE -3	100110000b
0FFFFFFh ROTATE -10h	0FFFF000Fh

3.3.2.13 Unsigned Less Than - (operand 1 LT operand 2)

The LT operator returns the logical comparison of two 32-bit unsigned integers. If operand 1 is less than operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 LT 2	-1
1 LT 1	0
-1 LT 0	0
-1 LT -2	0
10 LT 10o	0

3.3.2.14 Unsigned Less Than Or Equal - (op 1 LE op 2)

The LE operator returns the logical comparison of two 32-bit unsigned integers. If operand 1 is less than or equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 LE 2	-1
1 LE 1	-1
-1 LE 0	0
-2 LE -1	-1
10 LE 10o	0

3.3.2.15 Unsigned Greater Than - (operand 1 GT operand 2)

The GT operator returns the logical comparison of two 32-bit unsigned integers. If operand 1 is greater than operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 GT 2	0
1 GT 1	0
-1 GT 0	0
-2 GT -1	0
10 GT 10o	-1

3.3.2.16 Unsigned Greater Than Or Equal - (op 1 GE op 2)

The GE operator returns the logical comparison of two 32-bit unsigned integers. If operand 1 is greater than or equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 GE 2	0
1 GE 1	-1
-1 GE 0	0
-2 GE -1	-1
10 GE 10o	-1

3.3.2.17 Logical Equal - (operand 1 EQ operand 2)

The EQ operator returns the logical comparison of logical values. If operand 1 and operand 2 are both true or if operand 1 and operand 2 are both false, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
0 EQ 0	-1
-1 EQ -1	-1
0 EQ -1	0
1 EQ 2	-1

3.3.2.18 Logical Not Equal - (operand 1 NE operand 2)

The NE operator returns the logical comparison of two logical values. If operand 1 is true and operand 2 is false or if operand 1 is false and operand 2 is true, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
0 NE 0	0
-1 NE -1	0
0 NE -1	-1
1 NE 2	0

3.3.2.19 Signed Less Than - (operand 1 < operand 2)

The < operator returns the logical comparison of two 32-bit signed integers. If operand 1 is less than operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
1 < 2	-1
1 < 1	0
-1 < 0	-1
10 < 10o	0

3.3.2.20 Signed Less Than Or Equal - (op 1 <= op 2)

The <= operator returns the logical comparison of two 32-bit signed integers. If operand 1 is less than or equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
1 <= 2	-1
1 <= 1	-1
-1 <= 0	-1
10 <= 10o	0

3.3.2.21 Signed Greater Than - (operand 1 > operand 2)

The > operator returns the logical comparison of two 32-bit signed integers. If operand 1 is greater than operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
1 > 2	0
1 > 1	0
-1 > 0	0
10 > 10o	-1

3.3.2.22 Signed Greater Than Or Equal - (op 1 >= op 2)

The >= operator returns the logical comparison of two operands. If operand 1 is greater than or equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 >= 2	0
1 >= 1	-1
-1 >= 0	0
10 >= 10o	-1

3.3.2.23 Arithmetic Equal - (operand 1 = operand 2)

The = operator returns the logical comparison of two arithmetic values. If operand 1 is equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 = 2	0
1 = 1	-1
-1 = 0	0
10 = 10o	0

3.3.2.24 Arithmetic Not Equal - (operand 1 <> operand 2)

The <> operator returns the logical comparison of two arithmetic values. If operand 1 is not equal to operand 2, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
1 <> 2	-1
1 <> 1	0
-1 <> 0	-1
10 <> 10o	-1

3.3.2.25 Logical AND - (operand 1 AND operand 2)

The AND operator returns the result of the logical AND of two logical values. If operand 1 and operand 2 are both true, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
-1 AND -1	-1
0 AND 0	0
-1 AND 0	0
0 AND -1	0
-2 AND 0FF0F23B0h	-1

3.3.2.26 Logical OR - (operand 1 OR operand 2)

The OR operator returns the result of logical OR of two logical values. If either operand 1 or operand 2 is true, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
-1 OR -1	-1
0 OR 0	0
-1 OR 0	-1
0 OR -1	-1
-2 AND 0FF0F23B0h	-1

3.3.2.27 Logical Exclusive OR - (operand 1 XOR operand 2)

The XOR operator returns the result of the logical exclusive OR between two logical values. If either operand 1 or operand 2 is true, but not both, the result is -1; otherwise the result is 0.

Example:

expression	result
-----	-----
-1 AND -1	0
0 AND 0	0
-1 AND 0	-1
0 AND -1	-1
-2 AND 0FF0F23B0h	0

3.3.2.28 Arithmetic And - (operand 1 & operand 2)

The & operator returns the result of the bitwise AND (or mask) of operand 1 and 2 where operand 2 is the mask.

Example:

expression	result
-----	-----
FFFFFFFFh & 0FFh	0FFh
252525252o & 707070o	505050o
110100101b & 1011001b	1b

3.3.2.29 Arithmetic Or - (operand 1 ! operand 2)

The ! operator returns the result of the bitwise OR of operand 1 and operand 2.

Example:

expression	result
-----	-----
0F0F0F0h ! 0F0F0Fh	FFFFFFFh
2134671o ! 123456o	2137777o
1101101b ! 111001b	1111101b

3.3.2.30 Arithmetic Exclusive OR - (operand 1 # operand 2)

The # operator returns the result of the bitwise exclusive OR between operand 1 and operand 2.

Example:

expression	result
-----	-----
1234 # 5678	4860
0ABCDh # 1234h	0B9F9h
0FFFFh # 0FFFFh	0h
0FF00h # 0FFh	0FFFFh
10101b # 11111b	1010b

3.4 TYPE

The user can display the contents of variables or arrays on the terminal with the TYPE command. The arguments to the TYPE command are the symbolic names of the variables or arrays whose contents are to be displayed. Arrays can be output in two different methods, by individual element or by a range. If only one value is given in square brackets, only that element will be output. If a range is given (value .. value), then all elements which fall within the range will be output.

There are two optional tags for each argument. The first indicates the radix in which the contents are to be displayed. The second tag specifies the minimum number of columns which FDL is to use in the display.

FDL supports another type of argument on the TYPE command called strings. A string is any combination of ASCII characters enclosed in double quotes ("). The TYPE command will display the string (except for the double quotes) on the terminal. If the user wishes to include a double quote in the string, two double quotes (") must be used.

There is no limit as to the number of arguments one can use with the TYPE command. Also the types of arguments (strings, variable, arrays) can be mixed in any order.

Syntax:

```
TYPE variable:radix:width, ... ;  
TYPE array[element]:radix:width, ... ;  
TYPE array[first..last]:radix:width, ... ;  
TYPE "string", ... ;
```


3.4.2 Output Width

The TYPE command by default will use 15 columns to display the contents of a variable. This field width can be modified by using a second tag on each argument.

NOTE

FDL will always use as a minimum the number of columns to correctly display the contents of the given variable in the specified radix. FDL will do this even if the specified field width is smaller than the required minimum. The letter which proceeds the output value is included in the output width.

Example:

```
* ALLOCATE a,b[0..2];           -- allocate some variables
* a:= 255;                      -- assign 255 decimal
* TYPE a:d:3;                   -- output width of 3
255
* TYPE a:d:5;                   -- output width of 5
255
* TYPE a::5;                    -- use default radix
FFh
* TYPE a:d:1;                   -- output width of 1
255
* b[0..2] := [0,1,2];           -- assign b
* TYPE b[0..2]:h:3;             -- output array b
0h 1h 2h
* TYPE b[0]::3,b[1],b[2]::5;    -- output by element
0h          1h 2h
* TYPE a:d:5,b[0]:b:3,b[2]:o:8
255 0b      2o
*
```

3.4.3 Output Strings

A string is any combination of ASCII characters enclosed in double quotes ("). The string (except for the double quotes) is output to the terminal. To output a double quote, two double quotes in a row must be used (").

Strings are limited to a maximum length of 85 characters. If a string is longer than 85 characters FDL will truncate the string. An error message will be output to the terminal indicating the string had been truncated.

>>>String truncated after 85 characters

Example:

```
* TYPE "This is an example of a string";
This is an example of a string
* TYPE;

* TYPE "";

* TYPE """";
"
* TYPE "This line includes a double quote (""");
This line includes a double quote (")
*
```


3.5 RANDOM

The RANDOM command generates random data values and stores them in variables previously defined by the user with the ALLOCATE statement. RANDOM takes as its arguments a list of variables or arrays with given ranges. If an array is given, random numbers are placed in the array within the range given. The range of values generated by RANDOM are between -2147483648 and 24783647 (0h to 0FFFFFFFh).

Syntax:

```
RANDOM var,... ;
```

Example:

```
* ALLOCATE a,b[0..3];           -- allocate some variables
* RANDOM a;                     -- get a random number
* TYPE a:h;                     -- write out the value
    986E3619h
* RANDOM b[0..3];               -- fill b with random values
* TYPE b[0..3]:h:10;            -- output b to the tty
    EE9FBE66h 4EE479F7h AFCD431Ch F2A3665h
* b[0..3] := [0];               -- zero b
* RANDOM b[1..2];               -- random b[1] and b[2]
* TYPE b[0..3]:h:10;            -- output b to the tty
    0h 530BEFA2h FEC0CAE3h      0h
* RANDOM a,b[0];                -- RANDOM accepts lists
* TYPE a:h:10,b[0]:h:10;        -- type them out
    4025DF14h 4BD97978h
*
```

CHAPTER 4

FASTBUS COMMAND STATEMENTS

There are two distinct classes of FASTBUS command statements in FDL. The first class involves complete operations. In this case a single FDL statement performs the entire sequence of arbitration, primary address cycle, secondary address cycle and data cycle. These statements can also hold or release mastership of FASTBUS at the discretion of the user.

The second class of FASTBUS command statements are the FASTBUS primitives. These correspond to single FASTBUS cycles such as arbitration and primary addressing. Primitives can be combined to build complex FASTBUS transactions which cannot be handled by the first class of commands.

It is possible to take advantage of the list driving features of some FASTBUS master interfaces such as the UNIBUS Processor Interface (UPI). Groups of FASTBUS commands can be clustered into what are called FBLOCKS. From these FBLOCKS, FDL builds the data structures which a list driven interface, like the UPI, can execute faster than if each command were done individually.

Status about the FASTBUS operation is relayed to the user via system defined variables or optionally by messages printed on the terminal.

4.1 COMPLETE OPERATIONS

Most data transfers on FASTBUS follow the same sequence of operations (arbitration, primary address, secondary address, data, termination). FASTBUS master interfaces such as the UPI have command structures optimized accordingly. With one request (for the UPI this is a list element), the interface can perform all FASTBUS cycles needed to transfer data. This type of data transfer is referred to as a complete operation.

There are eight different commands which perform complete operations. Each command executes all the FASTBUS cycles needed to do a single or block data transfer. These commands fall into two groups - IN and OUT. IN commands perform FASTBUS read data cycles. OUT commands perform FASTBUS write data cycles. The secondary address cycle, which is optional, and the type of data transfer (single or block) are also implied by the name. Switches are used to determine the mode select lines value during the primary address cycle (control vs data space, single device addressing vs broadcast). Switches are also used to determine the termination sequence of the operation; whether to hold or not to hold FASTBUS mastership.

4.1.1 Command Syntax

The syntax for each command statement is as follows:

IN	/switch(s) pad,var;
INNTA	/switch(s) pad,sad,var;
INBLK	/switch(s) pad,blk[range],cnt;
INBLKNTA	/switch(s) pad,sad,blk[range],cnt;
OUT	/switch(s) pad,var;
OUTNTA	/switch(s) pad,sad,var;
OUTBLK	/switch(s) pad,blk[range],cnt;
OUTBLKNTA	/switch(s) pad,sad,blk[range],cnt;

4.1.1.1 Pad - primary address

Constant or variable containing the primary address.

4.1.1.2 Sad - secondary address

Constant or variable containing the secondary address.

4.1.1.3 Var - variable

Variable in which to store data on reads or variable which contains the data for writes. This argument cannot be a constant.

4.1.1.4 Blk[range] - block transfer array

Array in which to store data on a block transfer read or an array which contains the data for a block transfer write. The [range] argument is of the form [first .. last]; first is the starting element of the array to be referenced and last is the last element of reference. First must be less than or equal to last. The range argument implies the length of the block transfer. For example, the range [0..5] would imply a six word block transfer.

4.1.1.5 Cnt - counter

Variable in which to store the number of data words transferred during a block transfer read or write.

NOTE

This variable does NOT contain the number of data words the user WISHES to transfer. The number of words to transfer is implied by the range given in the blk argument.

4.1.1.6 /switch(s) - action modifiers

Each complete operation command can accept five different switches which modify their actions. The five switches are

1. /CS - Control space mode select

The primary address cycle addresses a device in control space. If this switch is not given, data space will be addressed.

2. /BR - Broadcast mode select

The primary address cycle broadcasts the address to a group of devices. If this switch is not given, only a single device will be addressed.

3. /HOLD - Hold FASTBUS mastership

When the FASTBUS operation completes, the interface will retain FASTBUS mastership. Normally mastership is released between FASTBUS operations.

4. /FIFO - First In, First Out device

The internal address of the device addressed is a FIFO. This information is needed for automatic retry on error during block transfer operations.

5. /PARITY - Ignore parity errors

Parity errors normally halt the operation with a fatal error. If this switch is used, parity errors will be ignored. This switch only has meaning for the IN group since parity errors cannot be detected by the master on OUTs.

4.1.2 Order Of Operations

Complete operation commands perform all needed FASTBUS cycles to complete a data read or write to a FASTBUS module. Some cycles, such as secondary address cycles, are optional. Others, such as the primary address cycle, are always done. The following are the cycles which are performed in order of execution.

1. Arbitration

The FASTBUS interface arbitrates for FASTBUS mastership. If the interface is already master, no arbitration cycle is performed. The arbitration vector to be used during the arbitration cycles is taken from the System Parameter Block. See the SET command for details on the System Parameter Block.

2. Primary Address Cycle

If the FASTBUS interface is currently address locked to a FASTBUS device, the address connection is broken prior to performing the primary address cycle. The primary address, which can be a logical, geographical, or broadcast address, is taken from the pad argument of the command. The mode select (MS) value to be used during the primary address cycle is determined by the use or non-use of the /CS and /BR switches. The following are the values of MS based upon the setting of the switches.

switches	MS	space	type
-----	--	-----	----
none	0	data	single
/CS	1	control	single
/BR	2	data	broadcast
/CS/BR	3	control	broadcast

3. Secondary Address Cycle

A secondary address cycle is only done by four complete operation commands (INNTA, OUTNTA, INBLKNTA and OUTBLKNTA). A secondary address cycle is a write data cycle whose MS=2. The secondary address to be used is taken from the sad argument.

4. Data cycle

The data cycle can be either a read or a write, and can transfer a single word or a block of words. Those commands which contain BLK in their name (INBLK, OUTBLK, INBLKNTA and OUTBLKNTA) do block transfers (MS=1). All other complete operation commands do single data transfers (MS=0). IN and OUT indicate if the cycle is to be a read or a write. For single data transfers, the data is taken or stored in the variable given by the var argument. For block data transfers, the data is taken or stored in the array given by blk[range] and the total number of words transfers is returned in the cnt argument. The switches /FIFO and /PARITY are used to control the FASTBUS interface if a FASTBUS error occurs. If the register within the device is a first in-first out device, /FIFO must be given for proper error handling. Parity errors which occur on read data cycles can be ignored by the use of the /PARITY switch.

5. Termination

Normally the FASTBUS interface will release mastership after the data cycle has been completed (GK=0, AS=0). However in some cases the user may want to maintain mastership of the bus after the operation is complete. If the /HOLD switch is used on the command, mastership will be maintained. Also any connection to a device which was addressed during the primary address cycle is maintained.

4.1.3 IN

IN reads a data word using single transfer mode (MS=0) from the device whose address is given by the primary address, and stores it in the given variable. This operation does not do a secondary address cycle.

Syntax:

```
IN/switch(s) pad,var;
```

Example:

```
* ALLOCATE pad,var;           -- allocate some variables
* pad := 1000000h;           -- assign a primary address
* IN pad,var;                 -- read from data space
* IN/CS pad,var;              -- read from CSR space
* IN/HOLD 11h,var;           -- read and hold the bus
*
```

4.1.4 INNTA

INNTA reads a data word using single transfer mode (MS=0) from the device whose address is given by the primary address, and stores it in the given variable. A secondary address cycle is performed immediately following the primary address cycle.

Syntax:

```
INNTA/switch(s) pad,var;
```

Example:

```
* ALLOCATE pad,sad,var;       -- allocate some variables
* pad := 1000000h;           -- assign a primary address
* sad := 0;                   -- and a secondary address
* INNTA pad,sad,var;          -- read from data space
* INNTA/CS pad,0,var;         -- read CSR 0
* INNTA/HOLD 11h,sad,var;     -- read and hold the bus
*
```


4.1.5 INBLK

INBLK reads a block of data words using block transfer mode (MS=1) from the device whose address is given by the primary address, and stores the data in the given array. This operation does not do a secondary address cycle.

Syntax:

```
INBLK/switch(s) pad,blk[range],cnt;
```

Example:

```
* ALLOCATE pad,blk[0..9],cnt;    -- allocate some variables
* pad := 1000000h;              -- assign a primary address
* INBLK pad,blk[0..9],cnt;      -- read 10 words
* INBLK 20h,blk[1..6],cnt;      -- read 6 words
* INBLK pad,blk[1..1],cnt;      -- read only 1 word
*
```

4.1.6 INBLKNTA

INBLKNTA reads a block of data words using block transfer mode (MS=1) from the device whose address is given by the primary address, and stores the data in the given array. A secondary address cycle is performed immediately following the primary address cycle.

Syntax:

```
INBLKNTA/switch(s) pad,sad,blk[range],cnt;
```

Example:

```
* ALLOCATE pad,sad,blk[0..9],cnt;
* pad := 1000000h;              -- assign a primary address
* sad := 0;                     -- and a secondary address
* INBLKNTA pad,sad,blk[0..9],cnt;
* INBLKNTA 20h,100h,,blk[1..6],cnt;
* INBLKNTA pad,sad,blk[1..1],cnt;
*
```

4.1.7 OUT

OUT writes a data word using single transfer mode (MS=0) to the device whose address is given by the primary address. The data is taken from the given variable. This operation does not do a secondary address cycle.

Syntax:

OUT/switch(s) pad,var;

Example:

```
* ALLOCATE pad,var;           -- allocate some variables
* pad := 1000000h;           -- assign a primary address
* var := 0F0F0F0F0h;         -- assign some data
* OUT pad,var;               -- write to data space
* OUT/CS pad,var;            -- write to CSR space
* OUT/HOLD 1lh,var;          -- write and hold the bus
*
```

4.1.8 OUTNTA

OUTNTA writes a data word using single transfer mode (MS=0) to the device whose address is given by the primary address. The data is taken from the given variable. A secondary address cycle is performed immediately following the primary address cycle.

Syntax:

OUTNTA/switch(s) pad,var;

Example:

```
* ALLOCATE pad,sad,var;      -- allocate some variables
* pad := 1000000h;           -- assign a primary address
* sad := 0;                  -- and a secondary address
* var := 0F0F0F0F0h;         -- assign some data
* OUTNTA pad,sad,var;        -- write to data space
* OUTNTA/CS pad,0,var;       -- write CSR 0
* OUTNTA/HOLD 1lh,sad,var;   -- write and hold the bus
*
```

4.1.9 OUTBLK

OUTBLK writes a block of data words using block transfer mode (MS=1) to the device whose address is given by the primary address. The data is taken from the given array. This operation does not do a secondary address cycle.

Syntax:

```
OUTBLK/switch(s) pad,blk[range],cnt;
```

Example:

```
* ALLOCATE pad,blk[0..9],cnt;    -- allocate some variables
* pad := 1000000h;              -- assign a primary address
* blk[0..9] := [0];            -- zero the array
* OUTBLK pad,blk[0..9],cnt;     -- write 10 words
* OUTBLK 20h,blk[1..6],cnt;     -- write 6 words
* OUTBLK pad,blk[1..1],cnt;     -- write only 1 word
*
```

4.1.10 OUTBLKNTA

OUTBLKNTA writes a block of data words using block transfer mode (MS=1) to the device whose address is given by the primary address. The data is taken from the given array. A secondary address cycle is performed immediately following the primary address cycle.

Syntax:

```
OUTBLKNTA/switch(s) pad,sad,blk[range],cnt;
```

Example:

```
* ALLOCATE pad,sad,blk[0..9],cnt;
* pad := 1000000h;              -- assign a primary address
* sad := 0;                    -- and a secondary address
* blk[0..9] := [0];            -- zero the array
* OUTBLKNTA pad,sad,blk[0..9],cnt;
* OUTBLKNTA 20h,100h,,blk[1..6],cnt;
* OUTBLKNTA pad,sad,blk[1..1],cnt;
*
```

4.2 PRIMITIVE OPERATIONS

Complete operation commands serve the needs of most types of FASTBUS data transfers. Since they do multiple FASTBUS cycles, complete operation commands are generally fast and efficient. However in some circumstances the user needs to do individual FASTBUS cycles in an arbitrary order. Primitive operation commands have been provided for this purpose.

There are ten different primitive operation commands. Each performs only a single FASTBUS cycle. Certain restrictions apply to the primitives and caution must be taken when using them. Primitive commands perform only single FASTBUS cycles, i.e. bus mastership and any address connection are maintained until a RELEASE command is executed. Primitives chained together form complex operations. If an attempt is made to perform a data transfer without an address connection, FDL aborts the operation and prints the an error message.

>>>Illegal sequence of operations

Since primitives maintain bus mastership between cycles, FASTBUS cannot be used by other masters until the bus is released. Caution should be exercised in these situations.

4.2.1 Command Syntax

The syntax for each primitive command is as follows:

```
ARBITRATE;  
ADDRESS      /switch(s) pad;  
PUTNTA       /switch(s) sad;  
GETNTA       /switch(s) var;  
PUTBLK       /switch(s) blk[range],cnt;  
GETBLK       /switch(s) blk[range],cnt;  
PUT          /switch(s) var;  
GET          /switch(s) var;  
RELEASE ADDRESS;  
RELEASE BUS;
```

4.2.1.1 Pad - primary address

Constant or variable containing the primary address.

4.2.1.2 Sad - secondary address

Constant or variable containing the secondary address.

4.2.1.3 Var - variable

Variable in which to store data on reads or variable which contains the data for writes. This argument cannot be a constant.

4.2.1.4 Blk[range] - block transfer array

Array in which to store data on a block transfer read or an array which contains the data for a block transfer write. The [range] argument is of the form [first .. last]; first is the starting element of the array to be referenced and last is the last element of reference. First must be less than or equal to last. The range argument implies the length of the block transfer. For example, the range [0..5] would imply a six word block transfer.

4.2.1.5 Cnt - counter

Variable in which to store the number of data words transferred during a block transfer read or write.

NOTE

This variable does NOT contain the number of data words the user WISHES to transfer. The number of words to transfer is implied by the range given in the blk argument.

4.2.1.6 /switch(s) - action modifiers

Some primitive operation commands can accept five different switches which modify their actions. The five switches are

1. /CS - Control space mode select

The primary address cycle addresses a device in control space. If this switch is not given, data space will be addressed.

2. /BR - Broadcast mode select

The primary address cycle broadcasts the address to a group of devices. If this switch is not given, only a single device will be addressed.

3. /HOLD - Hold FASTBUS mastership

When the FASTBUS operation completes, the interface will retain FASTBUS mastership. Normally mastership is released between FASTBUS operations.

4. /FIFO - First In, First Out device

The internal address of the device addressed is a FIFO. This information is needed for automatic retry on error during block transfer operations.

5. /PARITY - Ignore parity errors

Parity errors normally halt the operation with a fatal error. If this switch is used parity errors will be ignored. This switch only has meaning for the IN group since parity errors cannot be detected by the master on OUTs.

4.2.2 ARBITRATE

The FASTBUS interface arbitrates for the FASTBUS mastership. If the FASTBUS interface is currently master, mastership is first released and an arbitration is performed.

Syntax:

```
ARBITRATE;
```

Example:

```
* ARBITRATE;                -- arbitrate for mastership
*
```

4.2.3 ADDRESS

The ADDRESS primitive performs a FASTBUS primary address cycle. The value of the mode select lines is determined by the use of the /CS and /BR switches. If the FASTBUS interface is not bus master, the ADDRESS primitive will automatically perform an arbitration cycle. If a device is currently addressed, the address connection is released prior to the new primary address cycle.

Syntax:

```
ADDRESS/switch(s) pad;
```

Example:

```
* ALLOCATE pad;              -- allocate a variable
* pad := 1000000h;           -- assign a primary address
* ADDRESS pad;               -- address data space
* ADDRESS/CS/BR 3h;          -- broadcast control space
* ADDRESS/CS 11h;            -- address control space
*
```

4.2.4 PUTNTA

The PUTNTA performs a secondary address write cycle. This cycle loads the NTA register of the connected slave with a new value. A slave must be addressed before issuing this command.

Syntax:

```
PUTNTA/switch(s) sad;
```

Example:

```
* ADDRESS/CS 11h;          -- address a device
* PUTNTA 0;                -- load NTA with a 0
* PUTNTA 100h;             -- load NTA with 100h
*
```

4.2.5 GETNTA

The GETNTA performs a secondary address read cycle. This cycle reads the NTA register of the connected slave and stores it in a variable. A slave must be addressed before issuing this command.

Syntax:

```
GETNTA/switch(s) var;
```

Example:

```
* ALLOCATE var;            -- allocate a variable
* ADDRESS/CS 11h;          -- address a device
* PUTNTA 0h;               -- load the NTA with a 0
* GETNTA var;              -- read the NTA
* TYPE var:h;              -- type out the variable
                           0h
* PUTNTA 100h;             -- load the NTA with 100h
* GETNTA var;              -- read the NTA
* TYPE var:h;              -- type out the variable
                           100h
*
```


4.2.6 PUTBLK

The PUTBLK primitive performs a block data write cycle. Data is taken from the blk[range] argument. The total number of words transferred are stored in the variable given in the cnt argument. A slave must be addressed before this primitive can be used.

Syntax:

```
PUTBLK/switch(s) blk[range],cnt;
```

Example:

```
* ALLOCATE blk[0..9],cnt;      -- allocate some variables
* blk[0..9] := [0];           -- zero the array
* ADDRESS 10h;                 -- address a device
* PUTNTA 0h;                   -- start at zero
* PUTBLK blk[0..9],cnt;        -- write 10 words
* TYPE cnt:h;                  -- write what is in cnt
                                Ah
*
```

4.2.7 GETBLK

The GETBLK primitive performs a block data read cycle. Data read from the slave is stored in the blk[range] argument. The total number of words transferred is stored in the variable given in the cnt argument. A slave must be addressed before this primitive can be used.

Syntax:

```
GETBLK/switch(s) blk[range],cnt;
```

Example:

```
* ALLOCATE blk[0..9],cnt;      -- allocate some variables
* ADDRESS 10h;                 -- address a device
* PUTNTA 0h;                   -- start at zero
* GETBLK blk[0..9],cnt;        -- read 10 words
* TYPE cnt:h;                  -- write what is in cnt
                                Ah
*
```

4.2.8 PUT

The PUT primitive performs a single data write cycle. The data to write is taken from the var argument. This argument must be a variable. A slave must be addressed before this primitive can be used.

Syntax:

```
PUT/switch(s) var;
```

Example:

```
* ALLOCATE var;           -- allocate a variable
* ADDRESS/CS OFFh;        -- address a device
* PUTNTA 3h;              -- point at CSR 3
* var := OFF000000h;       -- load var
* PUT var;                -- load CSR 3 with var
*
```

4.2.9 GET

The GET primitive performs a single data read cycle. The data read from the slave is stored in the var argument. A slave must be addressed before this primitive can be used.

Syntax:

```
GET/switch(s) var;
```

Example:

```
* ALLOCATE var;           -- allocate a variable
* ADDRESS/CS OFFh;        -- address the EG board
* PUTNTA 3h;              -- point at CSR 3
* GET var;                -- read the register
* TYPE var:h;             -- write out the variable
  FF000000h
*
```

4.2.10 RELEASE ADDRESS

The RELEASE ADDRESS primitive releases any address connection to a FASTBUS slave. If no slave is currently addressed, this command performs no action. Mastership of FASTBUS is maintained.

Syntax:

```
RELEASE ADDRESS;
```

Example:

```
* ADDRESS/CS OFFH;           -- connect to the EG board
* RELEASE ADDRESS;          -- release the connection
*
```

4.2.11 RELEASE BUS

The RELEASE BUS primitive releases mastership of FASTBUS. Any address connection is released prior to releasing mastership. If the FASTBUS interface is not master, this primitive performs no action.

Syntax:

```
RELEASE BUS;
```

Example:

```
* ARBITRATE;                 -- get control of the bus
* RELEASE BUS;               -- release mastership
*
```

4.3 LIST OPERATIONS

Some FASTBUS master interfaces use lists for their command structure. This type of interface can execute many commands without the aid of the host CPU. The UPI is such an interface. FDL supports lists using two command statements, FBLOCK and FEND. When the user enters the FBLOCK statement, FDL parses the successive statements and builds a list. Interpretation of the statements does not take place until the user terminates the block with the FEND statement.

Only FASTBUS command statements, both complete operations and primitives, can be used within FBLOCKS. If a non-FASTBUS statement is issued within an FBLOCK, FDL outputs the error message.

>>>Illegal statement in FBLOCK

The illegal statement is ignored and FDL remains in the FBLOCK parsing state. To help the user understand what parsing state FDL is in, FDL changes its prompt from a * to a > whenever it is in the FBLOCK parsing state.

Syntax:

```
FBLOCK;  
  <FASTBUS statements>  
FEND;
```

Example:

```
* ALLOCATE id,log;          -- allocate some variables  
* FBLOCK;                  -- start an FBLOCK  
> ADDRESS/CS 13h;          -- address in CSR space  
> PUTNTA 0;                -- point at CSR 0  
> GET id;                  -- read the id  
> PUTNTA 3;                -- point at CSR 3  
> GET log;                 -- get its logical address  
> RELEASE ADDRESS;         -- release the address  
> RELEASE BUS;             -- release the bus  
> FEND;                    -- execute the list  
*
```

4.4 SYSTEM VARIABLES

Whenever a the FASTBUS interface performs a FASTBUS operation, a variety of errors can result. To aid in reporting the status of FASTBUS operations, FDL maintains a set of 30 system variables. These variables always contain a value of either 0 or -1. A value of 0 indicates a false condition; a value of -1 indicates a true condition. The contents of these system variables are updated after each FASTBUS command statement or list. These variables can be used anywhere an arithmetic expression is legal. They cannot be used in WRITE statements.

The following is a list of FASTBUS system variables:

FBERR	FASTBUS fatal error
FBWRN	FASTBUS non-fatal error (warning)
FBPAR	FASTBUS parity error
FBBUS	FASTBUS interface is bus master
FBADR	FASTBUS interface has an address connection
RTOBUS	Arbitration timeout
RTOADR	Primary address cycle timeout
SS1ADR	Slave status = 1 on primary address cycle
SS2ADR	Slave status = 2 on primary address cycle
SS3ADR	Slave status = 3 on primary address cycle
SS4ADR	Slave status = 4 on primary address cycle
SS5ADR	Slave status = 5 on primary address cycle
SS6ADR	Slave status = 6 on primary address cycle
SS7ADR	Slave status = 7 on primary address cycle
RTONTA	Secondary address cycle timeout
SS1NTA	Slave status = 1 on secondary address cycle
SS2NTA	Slave status = 2 on secondary address cycle
SS3NTA	Slave status = 3 on secondary address cycle
SS4NTA	Slave status = 4 on secondary address cycle
SS5NTA	Slave status = 5 on secondary address cycle
SS6NTA	Slave status = 6 on secondary address cycle
SS7NTA	Slave status = 7 on secondary address cycle
RTODAT	Data cycle timeout
SS1DAT	Slave status = 1 on data cycle
SS2DAT	Slave status = 2 on data cycle
SS3DAT	Slave status = 3 on data cycle
SS4DAT	Slave status = 4 on data cycle
SS5DAT	Slave status = 5 on data cycle
SS6DAT	Slave status = 6 on data cycle
SS7DAT	Slave status = 7 on data cycle

Example:

```
* ALLOCATE var,id;           -- allocate a variable
* INNTA/CS 20h,0,id;         -- read the id
* var := RTOADR;             -- get the RTO flag
* TYPE id:h,var:d;           -- output the id and flag
    110000h                  0
*
```

4.5 ERR\$STATUS SYSTEM VARIABLE

In addition to the previously defined system variables, FDL maintains a FASTBUS interface-dependent system variable for status information. This variable has the symbolic name ERR\$STATUS. ERR\$STATUS is an array with a range of [0..22] which contains information about the previous FASTBUS command statement or list. It can be used anywhere an ALLOCATEable variable is legal. The format of ERR\$STATUS will differ between the various types of FASTBUS interfaces used. The following is a description of the status returned by the UPI. The user should refer to the document "FASTBUS Segment Driver Microcode Description" for a detailed description about the UPI and the status it returns.

4.5.1 ERR\$STATUS[0] - Header Status Word

The header status word contains flags which indicate what type of errors (or warnings) occurred during the execution of the last FASTBUS statement or list. This is the STLESW entry of the UPI status block.

4.5.2 ERR\$STATUS[1] - Primary Address

If the FASTBUS statement or list terminates with a FASTBUS slave still connected, this entry in ERR\$STATUS contains the primary address used to connect to that slave. This is the STLPAD entry of the UPI status block.

4.5.3 ERR\$STATUS[2] - Secondary Address

If the FASTBUS statement or list terminates with a FASTBUS slave still connected, this entry in ERR\$STATUS contains the secondary address used to address into the slave module. This is the STLSAD entry of the UPI status block.

4.5.4 ERR\$STATUS[3..22] - Status Elements

These entries contain the status elements (STEESW) for each FASTBUS command statement in a list. If the user issued only a single FASTBUS command statement, only ERR\$STATUS[3] will contain any information. If the user used the FBLOCK list structure, ERR\$STATUS[3] contains information on the first FASTBUS command within the list, ERR\$STATUS[4] contains information on the second FASTBUS command, etc.

4.6 FASTBUS MESSAGES

In some cases, system variables are a slow and tedious way of finding the status of a FASTBUS operation. To alleviate this situation, FDL supports FASTBUS messages. When the user has enabled the FASTBUS messages, FDL will output a one line message to the terminal indicating the status of the last executed FASTBUS command statement. These messages fall into three classes, status, error and warning.

1. Status messages report the state of the FASTBUS interface after the requested operation completes. Currently only the state of the FASTBUS interface with regards to FASTBUS mastership and slave address connection are reported.
2. Error messages report any fatal errors which caused the FASTBUS interface to abort the requested operation prior to its completion. The type error (slave status number, response timeout, etc) is reported along with the class.
3. Warning messages report any non-fatal errors or warnings which occurred during the execution of the requested operation. FASTBUS errors which fall into the warning class are those which were successfully retried or ignored. The type of warning (slave status number, response timeout, etc) is reported along with the class.

FASTBUS messages are enabled and disabled via the SET command statement. Initially all FASTBUS messages are disabled.

Syntax:

```
SET class ON;           -- turn messages on
SET class OFF;          -- turn messages off
```

Example:

```
* SET STATUS ON;        -- turn status messages on
* SET STATUS OFF;       -- turn status messages off
* SET ERROR ON;         -- turn error messages on
* SET ERROR OFF;        -- turn error messages off
* SET WARN ON;          -- turn warning messages on
* SET WARN OFF;         -- turn warning messages off
*
```

There are thirty different key words associated with FASTBUS messages. They will appear on the terminal in one line immediately following the FASTBUS command statement. The following is a list of the key words.

1. fbbus - FASTBUS mastership

The FASTBUS interface maintained bus mastership after completing the FASTBUS operation. This is a status class message.

2. fbadr - FASTBUS address connection

The FASTBUS interface maintained an address connection to a FASTBUS slave module. This is a status class message.

3. fberr - Fatal error

A fatal FASTBUS error has occurred. The FASTBUS interface did not complete execution of the requested operation. This is an error class message.

4. fbwrn - Warning

A non-fatal FASTBUS error (warning) has occurred. The requested operation has completed the requested operation however an error which was ignored or retried successfully occurred during execution. This is a warning class message.

5. fbpar - FASTBUS parity error

The FASTBUS interface detected a parity error while reading the FASTBUS address/data lines. This may be an error or warning class message.

6. rtobus - Arbitration timeout

The FASTBUS interface was unable to gain bus mastership. This may be an error or warning class message.

7. rtoadr - Primary address cycle timeout

The FASTBUS interface was unable to address the a FASTBUS device with the given primary address. This may be an error or warning class message.

8. rtonta - Secondary address cycle timeout

The FASTBUS interface timed out when attempting to perform a secondary address cycle to the connected device. This may be an error or warning class message.

9. rtodat - Data cycle timeout

The FASTBUS interface timed out when attempting to perform a data cycle to the connected device. This may be an error or warning class message.

10. ssNadr - Slave status error on primary address cycle

The FASTBUS interface detected a slave status error (SS=N) after performing a primary address cycle. This may be an error or warning class message.

11. ssNnta - Slave status error on secondary address cycle

The FASTBUS interface detected a slave status error (SS=N) after performing a secondary address cycle. This may be an error or warning class message.

12. ssNdat - Slave status error on data cycle

The FASTBUS interface detected a slave status error (SS=n) after performing a data cycle. This may be an error or warning class message.

Example:

```
* SET ERROR ON;           -- turn errors on
* SET WARN ON;            -- turn warnings on
* SET STATUS ON;          -- turn status on
* ALLOCATE id;             -- allocate some variables
* INNNTA/CS 20h,0,id;      -- read the id of a device
fberr  rtodr
* INNNTA/CS/HOLD 21h,0,id; -- try at 21h
fbbus  fbadr
* PUTNTA 8h;              -- try CSR 8h
fberr  ss7nta
*
```

4.7 HOLD BUS ON ERROR

Normally the FASTBUS master interface will release any addressed device and bus mastership whenever a fatal error occurs. In some circumstances, particularly in diagnostics, the user wants to freeze the state of the bus when such an error occurs. An example would be a slave status value of 1, 2 or 3 during a primary address cycle. When such an error occurs, the interface has actually not addressed the desired slave module. The address connection is to a Segment Interconnect module somewhere in the logical address path. The user may wish to read the appropriate CSR registers of the connected SI to determine where the address failed. If the interface released the bus after such an error the user would not be able to interrogate the SI. FDL supports a "hold on error" (HOLDERR) option which maintains the current state of FASTBUS whenever there is a fatal error. The HOLDERR option is controlled via the SET command statement. The initial state of HOLDERR is off.

Syntax:

SET HOLDERR ON;	-- hold on error
SET HOLDERR OFF;	-- do not hold on error

Example:

* ALLOCATE id;	-- allocate a variable
* SET HOLDERR OFF;	-- dont hold the bus
* SET ERROR ON;	-- turn messages on
* SET STATUS ON;	-- for error and status
* INNNTA/CS 1000020h,0,id;	-- read a module in group 1
tberr ss2adr	
* SET HOLDERR ON;	-- hold the bus
* INNNTA/CS 1000020h,0,id;	-- try again with holderr
fberr ss2adr fbbus fbadr	
* PUTNTA 0h;	-- point at CSR 0 of the SI
fbbus fbadr	
* GET id;	-- read the id
fbbus fbadr	
* TYPE id:h;	-- output the SI id
800002h	
* PUTNTA 42h;	-- read the near side
fbbus fbadr	
* GET id;	-- geographical address
fbbus fbadr	
* TYPE id:h;	-- output it
200001Fh	
* RELEASE ADDRESS;	-- release the SI
fbbus	
* RELEASE BUS;	-- release the FASTBUS
*	

4.8 SYSTEM PARAMETER BLOCK

To perform a FASTBUS operation, the FASTBUS master interface may need a large amount of information. The FASTBUS complete operations and primitive commands require the user to supply with each command information which is changed frequently, i.e. the primary address. There are other parameters which are not frequently changed which are contained in the System Parameter Block. The arbitration vector, error control words, retry counter and others are contained within the parameter block. Each entry in the parameter block is a 32 bit word. The UPI requires seven different entries in its System Parameter Block. For a complete description of each entry in the parameter block, the user should read the "FASTBUS Segment Driver Microcode Description".

The entries in the System Parameter Block can be changed with the SET command. The SET command takes two arguments. The first is a key word associated with a particular entry in the parameter block. The second argument is a variable or constant which is the new value for the given entry. There are seven key words associated with the SET command.

4.8.1 BURST - Block Transfer Burst Size

The UPI automatically breaks block transfers into small packets called bursts. Between bursts the UPI releases both the slave device and bus mastership. This allows other masters access to FASTBUS during a large block transfer. The UPI allows bursts of from 1 to 256 words. The default burst size is 256.

4.8.2 CLOCK - Clock Cycle

This parameter would contain the cycle frequency to be used during pipeline block transfers. The UPI does not support pipeline block transfers at this time. This parameter is provided for compatibility with other types of FASTBUS masters.

4.8.3 VECTOR - Arbitration Vector

This entry contains the 6 bit arbitration vector and arbitration inhibit flag (AI) for the interface. The vector resides in bits 0 thru 5. The AI flag resides in bit 6. All other bits are ignored.

4.8 SYSTEM PARAMETER BLOCK

To perform a FASTBUS operation, the FASTBUS master interface may need a large amount of information. The FASTBUS complete operations and primitive commands require the user to supply with each command information which is changed frequently, i.e. the primary address. There are other parameters which are not frequently changed which are contained in the System Parameter Block. The arbitration vector, error control words, retry counter and others are contained within the parameter block. Each entry in the parameter block is a 32 bit word. The UPI requires seven different entries in its System Parameter Block. For a complete description of each entry in the parameter block, the user should read the "FASTBUS Segment Driver Microcode Description".

The entries in the System Parameter Block can be changed with the SET command. The SET command takes two arguments. The first is a key word associated with a particular entry in the parameter block. The second argument is a variable or constant which is the new value for the given entry. There are seven key words associated with the SET command.

4.8.1 BURST - Block Transfer Burst Size

The UPI automatically breaks block transfers into small packets called bursts. Between bursts the UPI releases both the slave device and bus mastership. This allows other masters access to FASTBUS during a large block transfer. The UPI allows bursts of from 1 to 256 words. The default burst size is 256.

4.8.2 CLOCK - Clock Cycle

This parameter would contain the cycle frequency to be used during pipeline block transfers. The UPI does not support pipeline block transfers at this time. This parameter is provided for compatibility with other types of FASTBUS masters.

4.8.3 VECTOR - Arbitration Vector

This entry contains the 6 bit arbitration vector and arbitration inhibit flag (AI) for the interface. The vector resides in bits 0 thru 5. The AI flag resides in bit 6. All other bits are ignored.

4.8.4 ERWADR - Primary Address Cycle Error Response Word

The primary address error response word contains the error response codes to be used for primary address cycle errors.

4.8.5 ERWNTA - Secondary Address Cycle Error Response Word

The secondary address error response word contains the error response codes to be used for secondary address cycle errors.

4.8.6 ERWDAT - Data Cycle Error Response Word

The data cycle error response word contains the error response codes to be used for data cycle errors.

4.8.7 RETRY - Retry Counter

This entry contains the maximum number of times an error should be retried before deciding the error is fatal.

Syntax:

```
SET BURST value;
SET CLOCK value;
SET VECTOR value;
SET ERWADR value;
SET ERWNTA value;
SET ERWDAT value;
SET RETRY value;
```

Example:

```
* ALLOCATE var;           -- allocate a variable
* var := 01111111h;       -- retry all errors
* SET ERWADR var;         -- primary address word
* SET ERWNTA var;         -- secondary address word
* SET ERWDAT var;         -- data word
* SET RETRY 1;            -- retry once
* SET BURST 128;          -- burst size of 128
*
```

4.9 RESET

The RESET command allows the user to reset either the local FASTBUS segment or the FASTBUS master interface.

4.9.1 RESET BUS

When RESET BUS is executed, the FASTBUS RB line is pulsed on the local FASTBUS segment for a minimum time period as specified in the "FASTBUS Specification" document. RESET BUS should be used with extreme caution since it will disable all devices on the local segment.

4.9.2 RESET INTERFACE

RESET INTERFACE will reset the FASTBUS master interface to its powerup state. All parameters in the System Parameter Block will be reset to their initial values. This command statement does not affect those flags which are FDL specific (HOLDERR, ERROR, WARN, STATUS, etc).

Syntax:

```
RESET BUS;  
RESET INTERFACE;
```

Example:

```
* RESET BUS;                -- reset the local segment  
* RESET INTERFACE;          -- reset the interface  
*
```

CHAPTER 5

I/O STATEMENTS

As with most programming languages, FDL has command statements which transfer data between the executing program and external devices. For most purposes this device will be the terminal, however data may be stored on or retrieved from any supported RT11 device such as a disk or line printer.

FDL supports two general purpose I/O statements, READ and WRITE. There are also two statements which open and close RT11 files, OPEN and CLOSE. Input can be from either of two logical devices; the /TT device is the terminal; the /IN device is any RT11 readable file opened by the OPEN command. Output can be to either of two logical devices; the /TT device is the terminal; the /OUT device is any RT11 writeable file opened by the OPEN command.

Besides the WRITE command, there are two other command statements which can output data to the /OUT device. The MAP command performs a mapping operation on a given FASTBUS group. The COMPARE command is used to compare the contents of two arrays. Both of these statements can output their information to either the terminal or the output device opened by the OPEN command.

5.1 OPEN

Before the user can use the /IN or /OUT devices with any I/O statement, the OPEN must be used to associate an RT11 file with the FDL logical device. Only two logical devices are supported by the OPEN statement /IN and /OUT. The device /IN is used to open a file for input. The device /OUT is used to open a file for output.

OPEN takes as its single argument an RT11 file specification enclosed in double quotes ("). A logical device of either /IN or /OUT must be used as a switch to indicate whether the file is to be opened for input or output. The default file specification is "DK:FDL.FDL".

If the user attempts to open an FDL device which has previously been opened, FDL first closes the current file before attempting to open the new file. Should FDL be unable to open the requested file, FDL aborts execution and outputs an error message.

```
>>>Unable to open output file
```

or

```
>>>Input file not found
```

Syntax:

```
OPEN/IN "<RT11 file specification>";  
OPEN/OUT "<RT11 file specification>";
```

Example:

```
* OPEN/IN "DK:INPUT.DAT";      -- open a file for input  
* OPEN/IN "LP:";               -- open lpt for input?  
>>>Input file not found  
* OPEN/OUT "OUTPUT";           -- open an output file  
* OPEN/OUT "LP:RESULT.DAT";    -- send data to the lpt  
* OPEN/OUT "CR:";              -- card reader for output?  
>>>Unable to open output file  
*
```


5.2 CLOSE

The CLOSE command closes an FDL logical device previously opened via the OPEN command. CLOSE does not take any arguments but the logical device to close must be given as a switch. If the device was not open, the statement is ignored.

NOTE

Any output file must be closed to flush all data from internal buffers and to make it a permanent part of the file directory.

Syntax:

CLOSE/IN;	-- close the input device
CLOSE/OUT;	-- close the output device

Example:

* OPEN/IN "DLO:DATA.DAT";	-- open an input file
* CLOSE/IN;	-- close the input file
* CLOSE/IN;	-- none open but thats ok
* OPEN/OUT "LP:";	-- open the line printer
* CLOSE/OUT;	-- close the output file
*	

5.3 WRITE AND WRITELN

The WRITE and WRITELN commands are similiar to the TYPE command in their actions. Both output the contents of variables, arrays and/or strings. However the WRITE and WRITELN commands can write data to two different devices, the terminal (/TT) and the output device (/OUT). The device to be used for output is indicated by using a logical device switch. If no switch is used, the terminal is assumed. If an attempt is made to write to an /OUT device which has not been opened, execution is aborted and FDL outputs an error message.

>>>Output file not open

WRITE and WRITELN differ from each other only in how the output line is terminated. WRITELN always terminates the line with a carriage return, line feed. However WRITE leaves the carriage where the last data was output. This allows multiple WRITE commands to write data on the same output line.

Syntax:

```
WRITE      <argument list>;
WRITE/TT   <argument list>;
WRITE/OUT  <argument list>;
WRITELN    <argument list>;
WRITELN/TT <argument list>;
WRITELN/OUT <argument list>;
```

Example:

```
* ALLOCATE a,b[0..3];           -- allocate some variables
* a := 1;                       -- assign a a value
* b[0..3] := [0,1,2,3];         -- assign b some values
* OPEN/OUT "LP:TEST";           -- open /OUT
* WRITELN a;                    -- write a to default
    lh
* WRITELN/TT a;                 -- write a to /TT
    lh
* WRITELN/OUT a;                -- send it to /OUT
* WRITE "The value of a is ";   -- output a string
* WRITELN a;
The value of a is               lh
* WRITE "The value of b is [";
* WRITELN b[0..3]:h:3,"";
The value of b is [ 0h 1h 2h 3h]
* WRITELN/OUT "The value of a is ",a;
* CLOSE/OUT;
* WRITE/OUT "This should make an error";
>>>Output file not open
*
```

5.4 READ

The READ command takes data from either the terminal or an input device. The data is stored in a list of variables which are the arguments of the READ command statement. Only constants in the four supported radices (decimal, hexadecimal, binary and octal) can be read. If the data input to the READ command is not a numeric constant, FDL aborts execution and outputs one of two different error messages.

```
>>>Invalid constant (read)
>>>Invalid character in constant
```

FDL supports a system defined variable, EOF\$INPUT, to indicate when all the data has been read from the input file. When the user attempts to read beyond the end of file, FDL sets EOF\$INPUT to true (-1). At all other times EOF\$INPUT is set to false (0). When the user attempts to read beyond the end of file, the variables in the argument list are set to zero.

Syntax:

```
READ      <variable list>;
READ/TT   <variable list>;
READ/IN   <variable list>;
```

Example:

```
* ALLOCATE a,b,c;           -- allocate some variables
* a := 1;                   -- assign a a value
* b := 2;                   -- assign b a value
* OPEN/OUT "TEST";          -- open "TEST" on "DK:"
* WRITE/OUT a,b;            -- write out a and b
* CLOSE/OUT;                -- close the output file
* OPEN/IN "TEST";           -- open same file for input
* c := EOF$INPUT;           -- get the eof flag
* TYPE c:d;                 -- type it out
                             0
* READ/IN b,a;              -- read back in b, a order
* c := EOF$INPUT;           -- get the end of file flag
* TYPE a,b,c:d;             -- what do we have
                             2h          lh          -1
* CLOSE/IN;                 -- close the input file
*
```

5.5 MODE

The TYPE, WRITE and WRITELN commands allow the user to output the contents of a variable in any of the four supported radices by applying a tag on the variable. This tag however is optional. When it is not given a default radix is assumed. Initially the default is hexadecimal. However the user may change the default at any time with the MODE command.

The MODE command takes only one argument - a key word which describes one of the four different radices which FDL supports.

NOTE

The MODE command only controls the default radix for output with the WRITE command. The READ command expects the user to suffix all constants with the radix tag (h, b or o). Decimal constants do not have any tags.

Syntax:

```
MODE DECIMAL;  
MODE HEX;  
MODE OCTAL;  
MODE BINARY;
```

Example:

```
* ALLOCATE a;           -- allocate a variable  
* a := 10;              -- assign 10 decimal  
* MODE DECIMAL;         -- default to decimal  
* TYPE a;               -- output a  
                        10  
* MODE HEX;             -- default to hexadecimal  
* TYPE a;               -- output a  
                        Ah  
* MODE OCTAL;           -- default to octal  
* TYPE a;               -- output a  
                        12o  
* MODE BINARY;          -- default to binary  
* TYPE a;               -- output a  
                        1010b  
*
```

5.6 COMPARE

In many diagnostic situations sets of data must be compared, i.e. a memory test. To facilitate this, the COMPARE command was implemented. COMPARE takes two arguments - two arrays with ranges of equal number of components. The contents of the two arrays are compared on an element by element basis. If any differences are found, the two values may be written to the terminal, an output device or not written at all. In all cases a system variable called "DIFFER" is set to true (-1) if there are differences and false (0) if there are none.

The only restriction on ranges is that both be of the same length. The ranges may start at different elements within the arrays and may even be given in inverse order. If the lengths of the ranges differ, FDL uses the smaller range and outputs the message

```
>>>Array range discrepancy, smaller range used
```

The output device is specified by a logical device switch given after the COMPARE statement. If no switch is given, no output is generated. Only the /TT and /OUT logical devices can be used.

Syntax:

```
COMPARE      array1[range],array2[range];  
COMPARE/TT   array1[range],array2[range];  
COMPARE/OUT  array1[range],array2[range];
```

Example:

```

* ALLOCATE a[0..4],b[0..4],c;      -- allocate some arrays
* a[0..4] := [0];                 -- zero a
* b[0..4] := [0];                 -- zero b
* COMPARE/TT a[0..4],b[0..4];      -- compare, no difference
* c := DIFFER;                    -- get the DIFFER flag
* TYPE c:d;                       -- what was it
                                0
* RANDOM a[0..4],b[0..4];          -- fill with random data
* COMPARE/TT a[0..4],b[0..4];      -- compare, results to tty
A      [ 0h .. 4h]                B      [ 0h .. 4h]
-----
530BEFA2h [ 0h]                   986E3619h [ 0h]
FEC0CAE3h [ 1h]                   EE9FBE66h [ 1h]
4BD97978h [ 2h]                   4EE479F7h [ 2h]
77267AF1h [ 3h]                   AFCD431Ch [ 3h]
51B74A1Eh [ 4h]                   F2A3665h [ 4h]
* COMPARE/TT a[0..4],b[1..0];      -- compare different ranges
>>>Array range discrepancy, smaller range used
A      [ 0h .. 4h]                B      [ 1h .. 0h]
-----
530BEFA2h [ 0h]                   EE9FBE66h [ 1h]
FEC0CAE3h [ 1h]                   986E3619h [ 0h]
* COMPARE a[0..4],b[0..4];        -- compare, no output
* c := DIFFER;                    -- get the DIFFER flag
* TYPE c:d;                       -- output the flag
                                -1
*

```

5.7 MAP

The MAP command is used to "map" a list of FASTBUS groups. The arguments can be either constants or variables. In mapping, FDL performs a primary address cycle to each of the 256 possible geographical addresses which can reside within a group. If a slave module responds at any address, FDL reads CSR registers CSR0, CSR3 and CSR8 and outputs the information to requested output device.

Since the group field of the primary address can vary in width between FASTBUS systems, the argument(s) given to the MAP command should be the geographical address of slot 0 of the given group. For example, on a system with an 8 bit group field, the group number resides in bits 31 through 24 of the primary address. If group 1 is to be mapped, the value 1000000h would be the argument to the MAP command. All bits between the least significant bit of the group field and the most significant bit of the geographical address field must be zero. In the above example, bits 23 through bit 8 must be zero.

The information which the MAP command collects can be output to either of two devices, the terminal or the logical output device. The device is specified by using either the /TT switch for terminal or /OUT switch for the logical output device. If no output device is specified, the terminal is assumed.

Syntax:

```
MAP      <group list>;  
MAP/TT   <group list>;  
MAP/OUT  <group list>;
```

Example:

```
* ALLOCATE a;           -- allocate a variable
* a := 1000000h;        -- map group 1
* MAP a;                -- map whats in a
```

Identity	Geographical	Logical	Vector
A00800h	1000003h	None	10h
110440h	100000Ch	None	1h
800002h	1000013h	None	2h
1040000h	10000FFh	1000000h	None

```
* MAP 0;                -- map the local segment
```

Identity	Geographical	Logical	Vector
A00800h	3h	None	10h
110440h	Ch	None	1h
800002h	13h	None	2h
1040000h	FFh	1000000h	None

```
* MAP 1000000h,2000000h,3000000h;
```

Identity	Geographical	Logical	Vector
A00800h	1000003h	None	10h
110440h	100000Ch	None	1h
800002h	1000013h	None	2h
1040000h	10000FFh	1000000h	None
800002h	200001Eh	None	2h
800002h	200001Fh	None	1h
1040000h	20000FFh	2000000h	None
810001h	3000001h	None	None
810001h	300000Ah	None	None
110000h	300000Dh	None	1h
800002h	3000013h	None	2h
810001h	3000016h	None	None
1040000h	30000FFh	3000000h	None

*

CHAPTER 6

PROCEDURES

Up to this point the features of FDL have only been discussed in terms of the interactive nature of FDL. It is also possible to create subroutines, called PROCEDURES, which are named blocks of FDL statements. When the user invokes the procedure with the CALL statement, FDL executes every statement within the procedure they were being typed to FDL one at a time. This allows the user to do repetitive operations or operations involving many statements more easily and with fewer mistakes.

Procedures may call other procedures. This allows complex programs to be written in a modular form. Procedure calling may even be recursive, i.e. a procedure may call itself. To allow such recursiveness, FDL supports two levels of variable allocation, GLOBAL vs LOCAL. Arguments may be passed between procedures when they are invoked with the CALL statement.

All FDL statements described through chapter 5 may be used within procedures or separately. The branching, flow and loop control statements, GOTO, CASE-GOTO, IF-GOTO and ON-GOTO can only be used in procedures. They are described later in this chapter.

6.1 PROCEDURES

A PROCEDURE is a named block of individual statements which can be invoked with a single request. Before a procedure can be used it must be defined. To define a procedure the user enters the PROCEDURE command and the symbolic name of the procedure to be defined. The name may be from 1 to 15 alphanumeric characters long and must begin with an alpha character (A through Z). The name cannot be one of FDL's reserved names. See the appendix "Reserved Names" for a list of FDL reserved names.

After the user issues the PROCEDURE statement, FDL changes its command prompt from a * to a > (similar to FBLOCK-FEND in section 4.3). When FDL is accepting statements for a procedure, all statements typed at the terminal are parsed for syntax but no interpretation takes place. FDL stays in this mode until the user types the END statement. The END statement terminates the definition of the procedure and enters the procedure name into the FDLs directory. The command prompt is changed back to a * indicating that FDL is in the interactive state.

Having defined a procedure, the user may now cause FDL to interpret the statements within the procedure by using the CALL statement. CALL takes as its only argument the name of the procedure to be invoked. When the procedure has been invoked, all the statements contained within the procedure are executed in sequential order except as modified by flow control statements. When the END statement is reached, FDL issues the interactive command prompt (*) and is again ready for interactive use.

It is possible to CALL procedures from within other procedures. Recursive calling is also allowed, i.e. a procedure may call itself. Recursive calling is described in section 6.8.

Syntax:

```
PROCEDURE <procedure name>;
    <FDL statements>;
END;

CALL <procedure name>;
```

Example:

```
* PROCEDURE example;           -- define example
> ALLOCATE a;                  -- allocate a variable
> a := 10;                     -- assign a value
> TYPE "The following are all equal to 10 decimal.";
> TYPE a:d:8;a:h:8,a:o:8,a:b:8;
> END;                         -- end the procedure
* CALL example;                -- invoke example
The following are all equal to 10 decimal
    10      Ah      12o    1010b
* PROCEDURE example2;          -- define example2
> TYPE "Example2 is called";
> CALL example;                -- call first example
> END;
* CALL example2;               -- call example2
Example2 is called
The following are all equal to 10 decimal
    10      Ah      12o    1010b
*
```

6.2 ALLOCATE

The ALLOCATE command was discussed in chapter 3. There are, however, two different forms of this command. Storage can be broken into two classes, global and local. If a variable is declared global, its contents can be referenced by any and all procedures and at the interactive level. However, local variables may only be accessed from within the procedure in which they were allocated. To define a global variable in a procedure, the /GLOBAL switch must be used when the variable is allocated. If the /GLOBAL is not used, FDL assumes the variable is to be local. Interactive ALLOCATE commands are always considered global.

It is possible and perfectly legal to have a global and a local variable of the same name. When an ambiguous situation arises, FDL will always use the local variable.

Syntax:

```
ALLOCATE      <variable list>;
ALLOCATE/GLOBAL <variable list>;
```

Example:

```
* ALLOCATE a,b;           -- a and b are globals
* a := 0;                 -- zero a
* b := 0;                 -- zero b
* PROCEDURE example;      -- define a procedure
> ALLOCATE a;             -- allocate a as a local
> ALLOCATE/GLOBAL c;      -- allocate c as a global
> a := 1;                 -- assign local a
> b := 2;                 -- assign global b
> c := 3;                 -- assign global c
> END;
* TYPE a:h:5,b:h:5;       -- output a and b
    0h    0h
* TYPE c:h:5;             -- c is not defined
>>>Variable not defined
>>>Variable: C
* CALL example;           -- do example
* TYPE a:h:5,b:h:5;       -- only b should change
    0h    2h
* TYPE c:h:5;             -- c is now defined
    3h
*
```

6.3 DEALLOCATE

The DEALLOCATE was discussed in chapter 3. However, there are also two different forms of the DEALLOCATE command. As with ALLOCATE, DEALLOCATE can take a /GLOBAL switch. If the user wishes to deallocate a variable which was defined as global, the /GLOBAL switch must be used. If DEALLOCATE is used at the interactive level, /GLOBAL is assumed.

It is not necessary to deallocate local variables which have been declared within a procedure. When FDL exits a procedure (by reaching the END statement) all variables which were defined as local are automatically deallocated.

NOTE

Global variables defined within a procedure remain defined when the procedure is exited.

Syntax:

```
DEALLOCATE      <variable list>;
DEALLOCATE/GLOBAL <variable list>;
```

Example:

```
* ALLOCATE a,b;           -- a and b are globals
* a := 0;                 -- zero global a
* b := 0;                 -- zero global b
* PROCEDURE example;     -- define example
> ALLOCATE c;             -- c is a local
> c := 0;                 -- zero c
> DEALLOCATE/GLOBAL b;    -- deallocate global b
> END;                   -- end example
* TYPE a:h:5;            -- output a
  0h
* TYPE b:h:5             -- output b
  0h
* TYPE c:h:5;            -- c does not exist
>>>Variable not defined
>>>Variable: C
* CALL example;          -- invoke example
* TYPE a:h:5;            -- output a
  0h
* TYPE b:h:5;            -- b does not exist
>>>Variable not defined
>>>Variable: B
* TYPE c:h:5;            -- c does not exist
>>>Variable not defined
>>>Variable: C
```

6.4 PROCEDURE PARAMETERS

In many cases it is desirable to pass data to or from procedures. This can be done in FDL by using parameter lists. When a procedure is defined, the procedure name may be followed by a list of symbolic names enclosed in parentheses. The parameter names can be from 1 to 15 alphanumeric characters in length, but must begin with an alpha character. Within the procedure these names are treated as local variables. These names may be used in expressions, TYPE statements, etc.

The initial value of these variables is determined by the CALL statement used to invoke the procedure. When the user calls the procedure, a list of variables or constants, the replacement list, must also be given. The replacement list must have the same number of entries as there are parameters defined for the procedure. When the procedure is entered, FDL creates local variables with the names given by the parameter list and the values of the replacement list are copied into these local variables.

Parameter passing is, by default, only one way. The original replacement variables are not affected when the parameter variable is modified within the procedure. If values are also to be passed back to the caller through the replacement variables, the parameter(s) used to export values must be prefixed with the key word OUT. Such parameters must have variables for their replacements in the CALL statement. If a constant is passed as the replacement for an export parameter, FDL aborts execution of the procedure and outputs an error message.

>>>Illegal replacement variable for OUT parameter

When the procedure is entered, FDL does not create local variables for export parameters; it uses the variable(s) passed in the CALL statement.

Parameter variables cannot be deallocated with the DEALLOCATE command. If the user attempts to deallocate a parameter variable, FDL aborts execution of the procedure and outputs an error message.

>>>Cannot DEALLOCATE procedure parameter variables

Syntax:

```

PROCEDURE <procedure name> (<parameter list>);
    <FDL statements>
END;

CALL <procedure name> (<replacement list>);

```

Example:

```

* PROCEDURE example(a,OUT b);    -- declare a procedure
> TYPE "The value of passed A =",a;
> TYPE "The value of passed B =",b;
> a := 0;                        -- zero a
> b := 0;                        -- zero b
> END;
* ALLOCATE c,d;                  -- allocate global c,d
* c := 100h;                     -- make c 100h
* d := 200h;                     -- make d 200h
* CALL example(c,d);             -- pass c and d
The value of passed A =         100h
The value of passed B =         200h
* TYPE c,d;                      -- only d should change
    100h                        0h
* CALL example(1,d);             -- try with a constant
The value of passed A =         1h
The value of passed B =         0h
* CALL example(d,1);             -- should be an error
>>>Illegal replacement variable for OUT parameter
*

```

6.5 BRANCHING

The normal flow of execution in FDL procedures is in sequential order. However in many situations it is desirable to repeat a given set of statements or to skip a set of statements entirely. FDL allows the user to change the flow of execution within a procedure with branching statements. The four different branching statements supported are GOTO, CASE-GOTO, IF-GOTO and ON-GOTO. All three change the flow of execution to a marker called a label.

A label marks symbolically a location to which execution may branch. A label is a name, from 1 to 15 alphanumeric characters in length, enclosed in double angle brackets (<<label-name>>). The name must begin with an alpha character (A through Z) and cannot be an FDL reserved name. (See the appendix "Reserved names" for a list of FDL reserved names) Labels can only be used within procedures. If an attempt is made to use a label at interactive level, FDL outputs an error message.

>>>Labels must be within procedures

Labels must be unique within a procedure but the same name may be used in different procedures. If the user issues the same label name within a procedure, FDL ignores the second label and outputs an error message.

>>>Duplicate label in procedure
>>>Label: label-name

Labels are not statements and therefore do not need to be separated from statements by a semicolon. They however must always come before any other text in a statement.

Syntax:

<<label-name>>

Example:

```
* PROCEDURE example;           -- define a procedure
> ALLOCATE a;                  -- allocate a variable
> <<label>>                     -- define a label
- TYPE "This line is marked by a label";
> <<another>> a := 1;           -- labels are not statements
> TYPE "A =",a;                -- output a
> END;
* CALL example;                -- invoke the example
This line is marked by a label
A =          1h
*
```


6.5.1 GOTO

The GOTO command is an unconditional branching statement and can only be used within procedures. GOTO takes as its only argument the name of a label which has been or will be defined within the current procedure. Only the label name must be given, not the double angle brackets. When FDL executes a GOTO, execution is transferred to the location within the procedure where the label is defined.

Forward referencing of labels is legal. Forward reference is made when a label is used in a GOTO before it has been defined. Because of forward referencing FDL will accept any legal name as the label on a GOTO command. However when the END command is issued, FDL verifies that all referenced labels have been defined. If a label has not been defined, FDL issues an error message.

```
>>>Unresolved label (GOTO, IF-GOTO, ON-GOTO)
>>>Label: label-name
```

Syntax:

```
GOTO label-name;
```

Example:

```
* PROCEDURE example;           -- define a procedure
> GOTO abc;                     -- goto label abc
> TYPE "This should not appear";
> <<abc>> TYPE "Only this will appear";
> END;
* CALL example;                 -- execute
Only this will appear
* procedure igoofed             -- another procedure
> GOTO nolabel;                 -- reference a label
> END;                          -- but dont define it
>>>Unresolved label (GOTO, IF-GOTO, ON-GOTO)
>>>Label: nolabel
*
```

6.5.2 IF-GOTO

The IF-GOTO command allows conditional branching. IF-GOTO takes two arguments; a logical expression on which the condition is based and the label to branch to if the condition is true. The logical expression can be any legal expression as described in chapter 3. If the result of the expression is true, FDL will branch to the label on the GOTO. If the result is false, FDL will go to the next statement in sequence. A true condition is any value which is negative (less than or equal to -1 decimal). A false condition is any value which is positive (greater than or equal to 0 decimal).

Forward referencing and undefined labels are handled the same as for the GOTO statement.

Syntax:

IF expression GOTO label;

Example:

```
* PROCEDURE example(end);          -- define a procedure
> ALLOCATE a;                      -- define a variable
> a := 0;                          -- zero a
> <<loop>> TYPE "A =",a;            -- output the value of a;
> a := a + 1;                      -- increment a
> IF (a LE end) GOTO loop;         -- loop till a equals end
> TYPE "end of example";          -- lets us know it is done
> END;
* CALL example(2);                 -- loop while a LE 2
A =                                0h
A =                                1h
A =                                2h
end of example
* CALL example(-1);                -- never loops
A =                                0h
end of example
*
```

6.5.3 CASE-GOTO

The CASE-GOTO command allows conditional branching to multiple labels. CASE-GOTO takes two arguments; a value which determines the branch and a list of labels which define the locations where the command can branch. The value can be a constant, variable or array reference. The label in the list to which the CASE-GOTO branches is determined by the value. A branch is taken to the first label if the value is zero; to the second label if the value is a one; to the third if the value is a two; etc. If the value is negative or if there is no label in the list for that value, no branch is taken and the next statement in sequence is executed.

Forward referencing and undefined labels are handled the same as for the GOTO statement.

Syntax:

```
CASE value GOTO <label list>;
```

Example:

```
* PROCEDURE example(val);           -- define a procedure
> CASE val GOTO L0,L1,L2;           -- value 0 thru 2 branch
> TYPE "No branch taken";
> RETURN;
> <<L0>>
> TYPE "Branch to L0";
> RETURN;
> <<L1>>
> TYPE "Branch to L1";
> RETURN;
> <<L2>>
> TYPE "Branch to L2";
> RETURN;
> END;
* CALL example(0);                   -- goto label 0
Branch to L0
* CALL example(1);                   -- goto label 1
Branch to L1
* CALL example(2);                   -- goto label 2
Branch to L2
* CALL example(3);                   -- no label for 3
No branch taken
* CALL example(-1);                  -- negative takes none
No branch taken
*
```

6.5.4 ON-GOTO And ON-CLEAR

In FASTBUS systems the number of error conditions which may be encountered is quite large. It is impractical to repeatedly check for all possible responses, yet it is desirable to know when an unexpected response occurs. Exception trapping solves this problem. To define an exception trap the ON-GOTO command is used. Associated with this command is a list of conditions which can be trapped. Whenever a condition occurs for which a trap has been set, FDL transfers control to the label indicated in the ON-GOTO command.

The ON-GOTO command takes two arguments; the exception to trap and the label to which control is passed. There are currently 33 different trappable conditions. When the given condition(s) become true, FDL transfers control to the labeled statement. Exception trapping may bridge procedures. For example, if procedure A sets up an exception trap, calls procedure B and procedure B generates the exception, procedure B is exited and control is passed to the label within procedure A. It is also possible to set up a different exception trap in each CALLED procedure for the same condition. The new exception trap overrides the any traps set up in previous procedures.

As with GOTO and IF-GOTO, ON-GOTO can only be used within procedures. The label referenced by ON-GOTO must reside within the same procedure even though ON-GOTOs remain in effect over procedure CALLs. Forward referencing is handled the same as for the GOTO and IF-GOTO cases.

To clear an ON-GOTO exception trap, the user issues the ON-CLEAR command. ON-CLEAR removes any traps for the given conditions at the current procedure level only. When FDL returns to the previous procedure, any CLEARED traps are restored.

The ON-GOTO and ON-CLEAR commands can take the key word ALL. If ALL is used, all 33 possible conditions will be set up for an exception trap or cleared.

The exception conditions available for ON-GOTO are:

FBERR	FASTBUS fatal error
FBWRN	FASTBUS non-fatal error (warning)
FBPAR	FASTBUS parity error
FBBUS	FASTBUS interface is bus master
FBADR	FASTBUS interface has an address connection
RTOBUS	Arbitration timeout
RTOADR	Primary address cycle timeout
SS1ADR	Slave status = 1 on primary address cycle
SS2ADR	Slave status = 2 on primary address cycle
SS3ADR	Slave status = 3 on primary address cycle
SS4ADR	Slave status = 4 on primary address cycle
SS5ADR	Slave status = 5 on primary address cycle
SS6ADR	Slave status = 6 on primary address cycle
SS7ADR	Slave status = 7 on primary address cycle
RTONTA	Secondary address cycle timeout
SS1NTA	Slave status = 1 on secondary address cycle
SS2NTA	Slave status = 2 on secondary address cycle
SS3NTA	Slave status = 3 on secondary address cycle
SS4NTA	Slave status = 4 on secondary address cycle
SS5NTA	Slave status = 5 on secondary address cycle
SS6NTA	Slave status = 6 on secondary address cycle
SS7NTA	Slave status = 7 on secondary address cycle
RTODAT	Data cycle timeout
SS1DAT	Slave status = 1 on data cycle
SS2DAT	Slave status = 2 on data cycle
SS3DAT	Slave status = 3 on data cycle
SS4DAT	Slave status = 4 on data cycle
SS5DAT	Slave status = 5 on data cycle
SS6DAT	Slave status = 6 on data cycle
SS7DAT	Slave status = 7 on data cycle
DIFFER	Difference detected in a COMPARE command
EOF\$INPUT	End of input file
EOF\$OUTPUT	End of output file (not supported)

The exception conditions are only checked at certain times during execution of a procedure. The DIFFER condition is only checked when the user issues a COMPARE command. Those conditions which deal with FASTBUS responses and status are only checked after a FASTBUS command is executed. The FASTBUS exception conditions are even checked in a specific order since it is possible to have more than one condition become true after executing a FASTBUS command. The order is:

1. RTOBUS - Arbitration timeout
2. RTOADR - Primary address cycle timeout
3. RTONTA - Secondary address cycle timeout
4. RTODAT - Data cycle timeout
5. FBPAR - FASTBUS read parity error
6. SSn - Any fatal non-zero slave status
7. FBERR - FASTBUS fatal error
8. FBWRN - FASTBUS warning
9. FBADR - FASTBUS address connection
10. FBBUS - FASTBUS bus master

Syntax:

```
ON <condition list> GOTO label;
ON <condition list> CLEAR;
ON ALL GOTO label;
ON ALL CLEAR;
```

Example:

```
* SET ERROR ON;           -- turn error message on
* ADDRESS/CS 10h;         -- anyone at 10h
fberr  rtoadr
* SET ERROR OFF;          -- turn error message off
* PROCEDURE example;      -- define example
> ON RTOADR GOTO noonehome; -- setup to trap rtoadr
> ADDRESS/CS 10h;         -- address 10h again
> TYPE "Device at 10h"; -- get here if no trap
> GOTO quit;              -- exit
> <<noonehome>>;          -- no one home label
> TYPE "No one home at 10h"; -- get here if we trap
> <<quit>>;               -- quit label
> END;                    -- end the procedure
* CALL example;           -- invoke the example
No one home at 10h
*
```

6.6 RETURN

The RETURN command allows the user to return from a CALLED procedure without branching to the END statement. RETURN takes no arguments.

Syntax:

```
RETURN;
```

Example:

```
* PROCEDURE example;           -- define a procedure
> RETURN;                      -- return immediately
> TYPE "This will not be typed";
> END;
* CALL example;                -- invoke the example
*
```

6.7 PROMPT

The PROMPT command allows the user to return to the interactive level of FDL immediately. No procedure returning is done. This is useful if the user has detected a fatal error and wants to abort the procedure. It is possible to be several CALLs deep when such an error is detected and the PROMPT command gives the user a graceful way to exit. The PROMPT command takes no arguments.

Syntax:

```
PROMPT;
```

Example:

```
* PROCEDURE example;           -- define a procedure
> CALL example2;               -- call example2
> TYPE "After call to example2";
> END;
* PROCEDURE example2;          -- define example2
> TYPE "Call to example2";     -- type something
> PROMPT;                      -- get back to prompt
> END;
* CALL example;                -- do it
Call to example2
*
```

6.8 RECURSIVE CALLS

In FDL, recursive calling, i.e. a procedure calling itself, is legal and can prove beneficial. Because of the local storage mechanism and parameter passing, each call to a procedure is independent from any other calls to the same procedure. The nesting depth for recursive calling is limited to the same depth as other CALLs. The following is an example of recursion.

```
* PROCEDURE factorial(n, OUT ans);
> --
> -- Recursively compute the factorial of a number in n.
> -- the result is returned in ans.
> --
> IF ((n <> 0) AND (n <> 1)) GOTO compute;
>   ans := 1;
>   RETURN;
>
- <<compute>>
- ALLOCATE nminus1, tmp;           -- get some local space
> nminus1 := n - 1;               -- get one less
> CALL factorial(nminus1,tmp);    -- compute factorial
> ans := n * tmp;
> END;
* ALLOCATE a;                     -- get a variable
* CALL factorial(1,a);            -- factorial of 1
* TYPE a:d;
      1
* CALL factorial(2,a);            -- factorial of 2
* TYPE a:d;
      2
* CALL factorial(3,a);            -- factorial of 3
* TYPE a:d;
      6
* CALL factorial(4,a);            -- factorial of 4
* TYPE a:d;
      24
*
```


6.9 CONTINUE

It is possible to halt the execution of a procedure by type two control-Cs (^C) in succession. When two control-Cs are typed, FDL stops executing the procedure and output a message.

>>>Procedure abort by control-C

Control is returned to the terminal at the interactive level. If desired, the interrupted procedure can resume execution from where it was halted. If the CONTINUE command is issued after execution has been halted, the procedure will resume execution. If any other command is issued instead of CONTINUE, the context of the interrupted procedure is lost and the procedure cannot be continued.

The CONTINUE command can also make a procedure continue from a PROMPT command. Execution will resume with the command immediately following the PROMPT.

Syntax:

CONTINUE;

Example:

```
* PROCEDURE example;           -- define a procedure
> TYPE "Before the PROMPT";    -- a little message
> PROMPT;                      -- halt
> TYPE "After the PROMPT";     -- more message
> END;
* CALL example;                -- invoke the procedure
Before the PROMPT
* CONTINUE;                    -- continue from PROMPT
After the PROMPT
* CONTINUE;                    -- no where to go
>>>Cannot continue
* CALL example;                -- try again
Before the PROMPT
* DIRECT;                      -- destroy procedure context
```

PROCEDURE EXAMPLE

```
* CONTINUE;                    -- context is lost
>>>Cannot continue
*
```

CHAPTER 7

UTILITIES

FDL has a set of command statements which are classified as utilities. They allow the user to manipulate procedures which have previously been defined. Saving and retrieving procedures from an RT11 disk device are possible with these utilities.

It is possible to create new command names in place of those which are currently in the FDL vocabulary. It is often useful when working at the interactive level to create abbreviations for some of the long FDL command names.

The SHOW command is used to display the many FDL parameters and tables. Many of the features of the SHOW command are used by the maintainer when developing new releases of FDL. However the user will find many of these features useful when developing and debugging complex procedures.

7.1 DIRECT

The DIRECT command is used to find out which procedures have been defined. DIRECT takes no arguments. It will output to the terminal a list of procedures, in alphabetical order, and the parameters which they require. If there are no procedures currently defined, FDL outputs an error message.

```
>>>Directory is empty
```

Syntax:

```
DIRECT;
```

Example:

```
* DIRECT;                                -- get a directory
>>>Directory is empty
* PROCEDURE a;                            -- define procedure a
> END;
* DIRECT;                                -- get a directory now

PROCEDURE A

* PROCEDURE c(x,y,OUT z);                 -- define c
> END;
* DIRECT;

PROCEDURE A
PROCEDURE C(X,Y,OUT Z)

* PROCEDURE b;                            -- define b
> END;
* DIRECT;

PROCEDURE A
PROCEDURE B
PROCEDURE C(X,Y,OUT Z)

*
```

7.2 DELETE

The DELETE command is used to remove a defined procedure from FDLs directory. The DELETE command takes a list of procedure names as its arguments. The keyword ALL may also be used to purge all procedures from the directory. If an attempt is made to delete a procedure which has not been defined, FDL outputs an error message.

```
>>>No such procedure
```

Syntax:

```
DELETE ALL;  
DELETE <procedure list>;
```

Example:

```
* DIRECT;                                -- get a directory  
  
PROCEDURE A  
PROCEDURE B  
PROCEDURE C(X,Y,OUT Z)  
  
* DELETE a,b;                            -- delete a and b  
* DIRECT;                                -- get another directory  
  
PROCEDURE C(X,Y,OUT Z)  
  
* DELETE a;                              -- try to delete a again  
>>>No such procedure  
* DELETE ALL;                            -- delete all  
* DIRECT;  
>>>Directory is empty  
*
```

7.3 INDIRECT FILES

FDL supports indirect files. This allows FDL to read an RT11 disk file containing FDL statements instead of requiring input from the terminal. FDL will read from the given file until it reaches the end of file. When the end of file is reached, FDL returns control to the terminal.

FDL is directed to accept input from an RT11 file by using the @ command, which takes one argument. This argument must be an RT11 file specification enclosed in double quotes (""); the same format is used by the OPEN command. The @ command can also take a switch; /SILENCE. When the @ is used without using the /SILENCE switch, all statements read from the file are displayed on the terminal as they are parsed. If /SILENCE is issued, the statements will not be displayed.

If a file specification is given for which a file does not exist or cannot be opened by RT11 for someother reason (i.e. output only device), FDL aborts the @ command and outputs an error message.

```
>>>Indirect file not found
```

Indirect files may contain any FDL statement except the EDIT command, which will be defined later. Indirect files can be nested (i.e. an indirect file can contain the @ command). The maximum depth of nesting is an installation dependent parameter. The user can find the maximum depth by using the SHOW STATUS command (defined later in this chapter).

FDL supports a startup indirect file based on the indirect file structure. When the FDL program is first entered, after initialization, FDL looks for the file "DK:FDLINI.FDL". If this file is found, FDL processes in the indirect file mode. This feature can be used if the user wishes to change default FDL parameters to a set more suited to particular needs.

Syntax:

```
@          "<file specification>";
@/SILENCE "<file specification>";
```

Example:

Assume the file DK:FACTOR.FDL contains the following

```
PROCEDURE factorial(n, OUT ans);
IF ((n NE 0) AND (n NE 1)) GOTO compute;
    ans := 1;
    RETURN;
<<compute>>
ALLOCATE nminus1, tmp;          -- get some local space
nminus1 := n - 1;               -- get one less
CALL factorial(nminus1,tmp);    -- compute factorial
ans := n * tmp;
END;                            -- end define
```

Now invoke the indirect file

```
* @/SILENCE "FACTOR";          -- read in factor.fdl
* DIRECT;                      -- get a directory

PROCEDURE FACTORIAL(N,OUT ANS)

* ALLOCATE a;                  -- allocate a variable
* CALL factorial(4,a);         -- get factorial of 4
* WRITE a:d;                   -- output to terminal
                                24
*
```

7.3.1 SILENCE

The SILENCE command disables the displaying of input from an indirect file. The SILENCE command takes no arguments. It cannot be used within a procedure definition and only has meaning when read from an indirect file. If SILENCE is issued from the terminal, FDL ignores the statement. When the SILENCE command is processed by FDL, further input from the indirect file is not displayed on the terminal.

Syntax:

```
SILENCE;
```

Example:

Assume the file DK:EXAMPL.FDL contains the following

```
PROCEDURE a;  
END;  
SILENCE;                -- no more output  
PROCEDURE b;  
END;
```

Now invoke the indirect file

```
* @"EXAMPL";            -- process the file  
* PROCEDURE a;  
* END;  
* SILENCE;              -- no more output  
* DIRECT;               -- issued by the user
```

```
PROCEDURE A  
PROCEDURE B
```

*

7.3.2 REVIVE

The REVIVE command is the inverse of the SILENCE command. REVIVE takes no arguments. It cannot be used within a procedure definition and only has meaning when read from an indirect file. If REVIVE is issued from the terminal, FDL ignores the statement. When the REVIVE command is processed by FDL, further input from the indirect file is displayed on the terminal.

Syntax:

```
REVIVE;
```

Example:

Assume the file DK:EXAMPL.FDL contains the following

```
PROCEDURE a;  
END;  
REVIVE;                -- turn output on  
PROCEDURE b;  
END;
```

Now invoke the indirect file

```
* @/SILENCE"EXAMPL";    -- process the file  
* PROCEDURE b;  
* END;  
* DIRECT;               -- issued by the user
```

```
PROCEDURE A  
PROCEDURE B
```

```
*
```


7.4 SAVE

The SAVE command is used to save defined procedures in a disk file. These files can be read at a later time using the indirect file command, @, to redefine the procedures. There are two formats to the SAVE command. The user can save selected procedures or all procedures. To save selected procedures, a list of procedures (without the parameter list) is given as the first argument. To save all procedures, no list of procedures is given. The second argument for both formats is always an RT11 file specification, enclosed in double quotes ("), into which the procedure are to be saved. The default file is "DK:FDL.FDL". If the file specified currently exists, the old file will be overwritten and its contents will be lost. If a procedure listed has not been defined, FDL outputs an error message and goes to the next procedure name in the list.

```
>>>No such procedure
>>>Procedure: procedure-name
```

Syntax:

```
SAVE <procedure list> ON "<file specification>";
SAVE ON "<file specification>";
```

Example:

```
* PROCEDURE example;           -- define a procedure
> WRITE "This is an example";  -- write something
> END;
* CALL example;                -- use it
This is an example
* DIRECT;                      -- get a directory
```

PROCEDURE EXAMPLE

```
* SAVE example ON "EXAMPL";    -- save example
* DELETE ALL;                  -- delete it
* DIRECT;
>>>Directory is empty
* @/SILENCE "exempl";          -- read it back
* DIRECT;                      -- get a directory
```

PROCEDURE EXAMPLE

```
* CALL example;                -- invoke it
This is an example
*
```

7.5 WAIT

The WAIT command is issued to suspend the execution of FDL for a specified time. The argument indicates the number of time units the delay is to take. One time unit is approximately 1/60 of a second. This feature requires the RT11 monitor support of the Mark Time feature (.MRKT). If the .MRKT feature is not supported, WAIT returns immediately.

Syntax:

WAIT value;

Example:

```
* ALLOCATE a;           -- get a variable
* a := 60;              -- 60 ticks
* WAIT a;               -- sleep for 1 second
*
```

7.6 EXIT

The EXIT command is used to exit from FDL and return control to the RT11 monitor. This is the only way to exit FDL. FDL will not allow an exit via a double control-C. Control-Cs are trapped. If a procedure is running at the time two control-Cs are typed, the procedure is aborted and a message is printed.

>>>Procedure aborted by control-C

Control-Cs typed at any other time are ignored by FDL.

Syntax:

EXIT;

Example:

```
.RUN FDL
FASTBUS Diagnostic Language    Version 3(74)
Initialization . . .
Complete

* EXIT;                      -- exit FDL
```

7.7 COMPRESS

Internally FDL maintains many tables and disk files. These tables contains such data as strings, names, procedure source code and procedure parsed code (pcode). As FDL is used and procedures are created and deleted, these tables become fragmented and contain "holes" which are unuseable. To reclaim these holes, compression must be done.

Compression can be automatic or manual. When automatic compression is used, FDL will compress its tables and files whenever a new procedure is defined. Since compression can take quite a long period of time (especially on a small floppy disk based system), automatic compression can be disabled via the SET COMPRESS command. When automatic compression is disabled, the user must periodically compress FDL's tables and files manually with the COMPRESS command. If compression is not done, FDL's tables and files can become full. When this occurs, an error message will be output to the terminal. No usefull work can be done in FDL until compression is done.

Syntax:

```
COMPRESS;  
  
SET COMPRESS ON;  
SET COMPRESS OFF;
```

Example:

```
* COMPRESS;                == compress manually  
* SET COMPRESS ON;         -- allow auto compression  
* SET COMPRESS OFF;       -- no auto compression  
*
```

7.8 CREATE

The CREATE command creates synonyms for FDL command names and key words. CREATE takes two arguments separated by a space or equal sign. The first argument is the new name which FDL will recognize as a command. The name must not be in use as a procedure or variable. The second argument is the current name for the command. After the new name is created, the old name will still be recognized by FDL.

The CREATE command can only be used if no procedures are defined. This is because the defined procedure could contain ALLOCATE statements which contain a variable name which is the same as the created command name. Though this would not cause any problem with this implementation of FDL, the user could become confused if the procedure were reparsed. The CREATED command name is now a reserved name and cannot be used as variable or procedure names.

Syntax:

```
CREATE new name = reserved name;
CREATE new name reserved name;
```

Example:

```
* CREATE typ = TYPE;           -- create typ from type
* TYPE "This is a type command";
This is a type command
* TYP "This is also a type command";
This is also a type command
* CREATE TYPE = SHOW;          -- TYPE is already defined
^
>>>User names can only be CREATED from system names
* CREATE a = b;                -- b is not a system name
>>>User names can only be CREATED from system names
* PROCEDURE example;          -- define a procedure
> END;
* CREATE x = CREATE;           -- should fail
>>>Cannot CREATE new names when PROCEDURES are defined
```

7.9 PURGE

The PURGE command deletes a user defined system name which was defined via the CREATE command. PURGE takes only one argument which is the name to purge.

Syntax:

```
PURGE user name;
```

Example:

```
* CREATE typ = TYPE;           -- create typ
* TYP "This is a string";
This is a string
* PURGE TYP;                   -- purge typ
* typ "This is a string";
^
>>>Expecting :=
* PURGE typ;                   -- try again
>>>Not a user defined system name
* PURGE TYPE;                 -- purge a reserved name
>>>Not a user defined system name
*
```

7.10 SHOW

The SHOW command is used to inspect various FDL parameters. These include the System Parameter Block, FASTBUS message flags, and other FDL compile time parameters. SHOW accepts several different key words as arguments. Any number of key words can be given to one SHOW command. Some of the features of the SHOW command are used in debugging new versions of FDL.

7.10.1 SHOW DIRECT

SHOW DIRECT performs the same action as the DIRECT command. A list of procedures and their arguments are listed on the terminal. If no procedures have been defined, FDL will output a message indicating an empty directory.

Syntax:

```
SHOW DIRECT;
```

Example:

```
* SHOW DIRECT;           -- get a directory
>>>Directory is empty
* PROCEDURE t;           -- define a procedure
> END;
* SHOW DIRECT;           -- get a directory
```

```
PROCEDURE T
```

```
*
```

7.10.2 SHOW DETAIL

As with the DIRECT key, DETAIL lists the procedure directory on the terminal, however additional information about the procedures is displayed along with the procedure names. The starting and ending locations within the instruction space (code array) and the location of the procedure source within the working scratch file are given. If an error was detected during the parsing phase, a message is given indicating this condition.

Syntax:

SHOW DETAIL;

Example:

```
* SHOW DETAIL;           -- get a detailed directory
>>>Directory is empty
* PROCEDURE t;           -- define procedure t
> END;
* PROCEDURE s;           -- define procedure s
> xyzz;                  -- generate an error
>>>Expecting :=
> END;
* SHOW DETAIL;           -- give some detail
```

PROCEDURE S

```
Syntax errors
Name table id    135
Swap file        2 to    2
Source file      3 to    5
```

PROCEDURE T

```
Name table id    134
Swap file        1 to    1
Source file      1 to    2
```

*

7.10.3 SHOW VARIABLE

SHOW VARIABLE displays on the terminal a list of all variables currently allocated. When issued from the interactive level, only GLOBAL variables will be displayed. When issued from within a procedure, only LOCAL variables will be displayed.

Syntax:

```
SHOW VARIABLE;
```

Example:

```
* SHOW VARIABLE;                -- display globals

Global allocations
  Variable : ERR$STATUS [ 0h.. 16h]

* PROCEDURE example(arg);        -- define a procedure
> ALLOCATE a;                    -- allocate a local variable
> SHOW VARIABLE;                 -- display variables
> END;

* CALL example(0);               -- invoke example

Local allocations for procedure EXAMPLE
  Variable : A
  Parameter : ARG

*
```


7.10.4 SHOW NAMES

SHOW NAMES displays on the terminal all the names FDL has stored in its name table. These include all reserved, user defined, procedure and variable names. This command is used primarily for debugging FDL.

Syntax:

SHOW NAMES;

Example:

* SHOW NAMES;

ADDRESS	ALL	ALLOCATE	AND
ARBITRATE	BINARY	BUS	BURST
CALL	CASE	CLEAR	CLOCK
CLOSE	COM	COMPARE	COMPRESS
CONTINUE	CREATE	DEALLOCATE	DECIMAL
DELETE	DETAIL	DIFFER	DIRECT
DIV	EDIT	END	EOF\$INPUT
EOF\$OUTPUT	EQ	ERROR	ERWADR
ERWDAT	ERWNTA	EXIT	FBADR
FBBUS	FBERR	FBLOCK	FBPAR
FBWRN	FEND	GA	GE
GET	GETBLK	GETNTA	GOTO
GT	HEX	HOLDERR	IF
IN	INBLK	INBLKNTA	INNTA
INTERFACE	LA	LE	LT
MAP	MOD	MODE	NAMES
NE	NOT	OCTAL	OFF
ON	OPEN	OR	OUT
OUTBLK	OUTBLKNTA	OUTNTA	PARAMETER
PROCEDURE	PROMPT	PURGE	PUT
PUTBLK	PUTNTA	RANDOM	READ
RELEASE	RESET	RETRY	RETURN
REVIVE	ROTATE	RTOADR	RTOBUS
RTODAT	RTONTA	SAVE	SET
SHIFT	SHOW	SILENCE	SS1ADR
SS2ADR	SS3ADR	SS4ADR	SS5ADR
SS6ADR	SS7ADR	SS1DAT	SS2DAT
SS4DAT	SS5DAT	SS6DAT	SS7DAT
SS1NTA	SS2NTA	SS3NTA	SS4NTA
SS5NTA	SS6NTA	SS7NTA	STATUS
STRING	SWAP	TYPE	VARIABLE
VECTOR	VERSION	WAIT	WARN
WRITE	WRITELN	XOR	ERR\$STATUS

7.10.5 SHOW STRING

SHOW STRING displays the character strings currently stored in FDL's string file. This command is used when debugging new releases of FDL.

Syntax:

```
SHOW STRING;
```

Example:

```
* SHOW STRING;                -- no strings defined
>>>No strings defined
* TYPE "This is a string";    -- define a string
This is a string
* SHOW STRING;

1(16) "This is a string"

* TYPE "New line";
New line
* SHOW STRING;

1(16) "This is a string"
2( 8) "New line"

*
```

7.10.6 SHOW VERSION

SHOW VERSION displays the FDL and microcode version numbers on the terminal.

Syntax:

```
SHOW VERSION;
```

Example:

```
* SHOW VERSION;                -- show the version

FDL version 3(74)
Microcode version 5(124)

*
```

7.10.7 SHOW PARAMETER

SHOW PARAMETER displays the System Parameter Block and the state of the FASTBUS message flags on the terminal. These parameters can be changed via the SET command described in chapter 4.

Syntax:

```
SHOW PARAMETER;
```

Example:

```
* SHOW PARAMETER;           -- display the parameters
```

System Parameter Block

Hardware burst size	[100h]
Clock cycle time	[0h]
Retries	[5h]
Arbitration vector	[1h]
Primary address cycle error response	[33333333h]
Secondary address cycle error response	[33333333h]
Data cycle error response	[33333333h]

FDL switches

Auto compression	[ON]
Hold FASTBUS on errors	[OFF]
FASTBUS error messages	[OFF]
FASTBUS warning messages	[OFF]
FASTBUS status messages	[OFF]

*

7.10.8 SHOW STATUS

SHOW STATUS displays on the terminal the current utilization of some of FDL's internal tables. Items such as variable allocation space, character string space and instruction space utilization are displayed.

Syntax:

```
SHOW STATUS;
```

Example:

```
* SHOW STATUS;
```

```
-- display FDL status
```

Storage utilization

Allocatable memory storage	23 out of	32767 words
Name table entries	136 out of	200 names
Swap in storage	0 out of	500 records

Disk file utilization

Indirect files	0 out of	3 files
Source file	0 out of	1200 lines
Stack file	0 out of	150 records
Swap file	0 out of	12800 records
String file	0 out of	600 records

```
*
```

7.10.9 SHOW ALL

SHOW ALL generates a display which is a combination of those generated by other SHOW key words. It is equivalent to typing the command,

SHOW VERSION,PARAMETER,STATUS;

Syntax:

SHOW ALL:

Example:

```
* SHOW ALL;                -- show all
```

FDL version 3(74)

Microde version 5(124)

System Parameter Block

```
Hardware burst size      [ 100h ]
Clock cycle time        [ 0h ]
Retries                 [ 5h ]
Arbitration vector      [ 1h ]
Primary address cycle error response [ 33333333h ]
Secondary address cycle error response [ 33333333h ]
Data cycle error response [ 33333333h ]
```

FDL switches

```
Auto compression           [ ON ]
Hold FASTBUS on errors     [ OFF ]
FASTBUS error messages     [ OFF ]
FASTBUS warning messages   [ OFF ]
FASTBUS status messages    [ OFF ]
```

Storage utilization

Allocatable memory storage	23 out of	32767 words
Name table entries	136 out of	200 names
Swap in storage	0 out of	500 records

Disk file utilization

Indirect files	0 out of	3 files
Source file	0 out of	1200 lines
Stack file	0 out of	150 records
Swap file	0 out of	12800 records
String file	0 out of	600 records

*

CHAPTER 8

EDITOR

Once a procedure has been defined, it is sometimes useful to be able to change statements within that procedure. FDL has a simple editor which allows one to inspect, insert, replace and delete statements within a procedure. When the editing session is complete, FDL reparses the procedure to check for syntax errors. The old copy of the procedure is deleted and the new version takes its place.

8.1 INVOKING THE EDITOR

The editor is invoked with the EDIT command. EDIT takes one argument, the name of the procedure to edit. If the procedure has been defined, FDL displays the number of lines used by the procedure along with the maximum number of lines available for use by the procedure. The maximum number of lines is based upon the size of the temporary work file FDL was able to open on the disk. Edit files are described in section 8.3. After the editor has been initialized, FDL outputs the edit command prompt, "\$", indicating that it is ready for edit commands.

Syntax:

```
EDIT procedure-name;
```

Example:

```
* PROCEDURE example;           -- define a procedure
> END;
* EDIT example;                 -- edit example
    2 lines, 16384 lines available
$
```

The editor can only be invoked from the interactive level and cannot be invoked from within an indirect file. If the EDIT command is used within a procedure or indirect file, FDL ignores the command and issues one of two different error messages.

```
>>>Statement not allowed in a PROCEDURE definition
```

or

```
>>>Statement not allowed in an indirect file
```

8.2 EDITOR COMMANDS

There are eight different commands available in the editor. All commands consist of a single letter. Some commands take a number as their argument; some take a single letter; while some take no argument at all. Commands and letter arguments can be given in upper or lower case. Numeric arguments can be positive or negative. Commands and their arguments may be separated by spaces but spaces are not required. Only one command per line is allowed. Commands are terminated by a carriage return.

All editor commands perform their actions relative to a location pointer. This location pointer, referred to as DOT, is initially set to the first line of the procedure. The value of DOT can be changed with the move command (M), and the beginning command (B). The list command (L) will display a selected number of lines beginning at DOT. Insert (I), delete (D) and replace (R) also begin their actions based on the current value of DOT.

The following sections describe each of the eight commands in detail. Included in each section is an example of the usage of the command. These examples assume the following has previously been executed by the user.

Example:

```
* PROCEDURE example(xyz);      -- define a procedure
> ALLOCATE a,b,c;              -- allocate some variables
> a := 1;
> b := 2;
> c := 3;
> WRITE a,b,c,xyz;
> END;
* EDIT example;                -- edit example
    7 lines, 16384 lines available
$
```


8.2.1 Move (M)

The M command changes the current value of DOT. M takes a numeric argument, positive or negative. The value of this argument is added to DOT to create a new pointer. The line at the new value is displayed on the terminal. If the value of the argument causes DOT to be moved before the first line or after the last line, DOT is set to the first line or last line respectively. If no argument is given, a value of 1 is assumed.

Example:

```
$ M
  ALLOCATE a,b,c;           -- allocate some variables
$ M -1
  PROCEDURE example(xyz);   -- define a procedure
$ M 1000
  END;
$ M -4
  a := 1;
$ M -2
  PROCEUDRE example(xyz);   -- define a procedure
$
```

8.2.2 Beginning (B)

The B command move the value of DOT to the first line of the procedure. No lines are displayed on the terminal.

Example:

```
$ B
$ M 1
  ALLOCATE a,b,c;           -- allocate some variables
$ B
$ M 1
  ALLOCATE a,b,c;           -- allocate some variables
$
```

8.2.3 List Lines (L)

The L command lists source lines in the procedure being edited on the terminal. L takes a numeric argument, N, which is the number of lines to display. If the argument is positive, N lines are displayed starting at the current value of DOT. If the argument is negative, L lists N lines starting N lines before DOT and ending on DOT. The value of DOT is not changed. If no argument is given, a value of 1 is assumed.

Example:

```
$ B
$ L 2
  PROCEDURE example(xyz);      -- define a procedure
  ALLOCATE a,b,c;              -- allocate some variables
$ L
  PROCEDURE example(xyz);      -- define a procedure
$ M 1000
  END;
$ L 10
  END;
$ L -10
  PROCEDURE example(xyz);      -- define a procedure
  ALLOCATE a,b,c;              -- allocate some variables
  a := 1;
  b := 2;
  c := 3;
  WRITE a,b,c,xyz;
  END;
$
```

8.2.4 Insert (I)

The I command inserts lines into the edited procedure. I takes a numeric argument, N, which is the number of lines to insert. If N is positive, the insertion is done after the line pointed to by DOT. If N is negative, the insertion is done before the line pointed to by DOT. If no argument is given, a value of 1 is assumed. After the I command is issued, the editor outputs a different prompt (!). When the ! prompt is given, the editor is asking for lines of text to be inserted into the procedure. When all the lines have been inserted, FDL switches back to the edit command prompt (\$). If there are not enough lines available within the work file to insert all requested lines, FDL shortens the request to the maximum number of lines which can be inserted. When the insert is complete, DOT points at the last line inserted.

Example:

```
$ B
$ M 4
  c := 3;
$ I 2
! a := xyz * 2;
! b := b + c;
$ L
  b := b + c;
$ B
$ L 100
  PROCEDURE example(xyz);          -- define a procedure
  ALLOCATE a,b,c;                  -- allocate some variables
  a := 1;
  b := 2;
  c := 3;
  a := xyz * 2;
  b := b * c;
  WRITE a,b,c,xyz;
  END;
$
```

8.2.5 Replace (R)

The R command replaces source lines within the edited procedure. R takes a numeric argument, N, which can be positive or negative. If the argument is positive, R replaces N lines starting with the current DOT. If the argument is negative, R replace N number of lines starting N lines before DOT and ending on DOT. If no argument is given, a value of 1 is assumed. After the R command has been issued, the editor issues the replacement prompt (!). As with the insert command, the editor is requesting lines of source which will replace those which currently exist within the procedure. The value of DOT is changed to point at the last line replaced.

Example:

```
$ B
$ L
  PROCEDURE example(xyz);      -- define a procedure
$ R
! PROCEDURE example(OUT xyz);  -- define a procedure
$ L 10
  PROCEDURE example(OUT xyz);  -- define a procedure
  ALLOCATE a,b,c;              -- allocate some variables
  a := 1;
  b := 2;
  c := 3;
  a := xyz * 2;
  b := b * c;
  WRITE a,b,c,xyz;
  END;
$ M 6
  b := b * c;
$ R -2
! xyz := xyz * 2;
! b := a + b + c;
$ L
  b := a + b + c;
$ B
$ L 100
  PROCEDURE example(OUT xyz);  -- define a procedure
  ALLOCATE a,b,c;              -- allocate some variables
  a := 1;
  b := 2;
  c := 3;
  xyz := xyz * 2;
  b := a + b + c;
  WRITE a,b,c,xyz;
  END;
$
```

8.2.6 Exit (E)

The E command causes FDL to exit the editor and return to the interactive level. E takes a single letter as its argument. The argument can be one of 3 different letters or blank. If the E command is given without an argument the following takes place.

If the procedure had been changed during the edit session, FDL deletes the old copy of the procedure. The new procedure is copied into the disk file "DK:EDIT.FDL". This file is subsequently processed by FDL as an indirect file.

If the procedure was not changed, FDL leaves the old copy of the procedure and discards the edit source procedure. A message is output to the terminal indicating that no changes had been made.

```
>>>No changes to procedure
```

The arguments of the E command are used to determine alternative actions during the exit. The following are legal exit commands.

1. ES - Exit with a silent parse

Same as the E command except the file "DK:EDIT.FDL" is processed with the /SILENCE switch (echoing is off).

2. EQ - Exit without any changes

EQ exits the editor but disregards any changes made during the edit session. The old procedure is retained and the new procedure is discarded.

3. EF - Exit with a forced parse

EF exits the editor and forces FDL to reparse the procedure even if no changes had been made during the edit session.

Example:

```
$ B
$ L 100
  PROCEDURE example(OUT xyz);  -- define a procedure
  ALLOCATE a,b,c;              -- allocate some variables
  a := 1;
  b := 2;
  c := 3;
  xyz := xyz * 2;
  b := a + b + c;
  WRITE a,b,c,xyz;
  END;

$ E
* PROCEDURE example(OUT xyz);  -- define a procedure
> ALLOCATE a,b,c;              -- allocate some variables
> a := 1;
> b := 2;
> c := 3;
> xyz := xyz * 2;
> b := a + b + c;
> WRITE a,b,c,xyz;
> END;
*
```

8.2.7 Help (H)

The H command lists a brief description of all the commands available within the editor. Help takes no arguments.

Example:

```
$ H
```

```
Commands are:
```

```
  L [n]    List n lines
  M [n]    Move n lines
  I [n]    Insert n lines
  D [n]    Delete n lines
  R [n]    Replace n lines
  B        Move to the beginning of procedure
  E        Exit editor
  ES       Exit editor with silent parse
  EQ       Exit editor but make no changes
  EF       Exit editor forcing a parse
  H        Type this text
```

```
n defaults to 1 if ommitted
```

```
$
```

8.3 EDITOR FILES

The editor uses disk files during the editing session. When the editor is first entered, FDL attempts to open a temporary work file. If FDL is unable to open this file, the edit session is aborted and an error message is given.

>>>Unable to open EDIT file

The size of the temporary file will vary depending on the current usage of the disk. The size of the file determines the maximum number of lines the edited procedure can have. If this temporary file is not large enough to hold all the source lines of the procedure to be edited, the edit session is aborted and an error message is given.

>>>EDIT file too small for procedure

Both of these errors can occur because there is not enough room on the default disk, "DK:". The user should return to the RT11 monitor and delete unneeded files from the disk with the DELETE command. The disk should also be compressed with the SQUEEZE command to remove any fragmentation of unused disk blocks. See the RT11 manuals for information on the DELETE and SQUEEZE commands.

When the edit session is complete and the E command has been given, FDL creates the disk file "DK:EDIT.FDL". Again it is possible to be unable to open the edit file or for the edit file to be too small. If either of these errors occurs while exiting the editor, the old procedure is retained and the new procedure is discarded. The nature of the FDL editor and the RT11 operating system make these errors unlikely while exiting.

APPENDIX A
RESERVED NAMES

The following names are reserved for use by FDL only.
The user cannot use these names for procedures, variables or
labels names.

ADDRESS	ALL	ALLOCATE	AND
ARBITRATE			
BINARY	BUS	BURST	
CALL	CASE	CLEAR	CLOCK
CLOSE	COM	COMPARE	COMPRESS
CONTINUE	CREATE		
DEALLOCATE	DECIMAL	DELETE	DETAIL
DIFFER	DIRECT	DIV	
EDIT	END	EOF\$INPUT	EOF\$OUTPUT
EQ	ERR\$STATUS	ERROR	ERWADR
ERWDAT	ERWNTA	EXIT	
FBADR	FBBUS	FBERR	FBLOCK
FBPAR	FBWRN	FEND	
GA	GE	GET	GETBLK
GETNTA	GOTO	GT	
HEX	HOLDERR		
IF	IN	INBLK	INBLKNTA
INNTA	INTERFACE		
LA	LE	LT	
MAP	MOD	MODE	
NAMES	NE	NOT	

OCTAL	OFF	ON	OPEN
OR	OUT	OUTBLK	OUTBLKNTA
OUTNTA			
PARAMETER	PROCEDURE	PROMPT	PURGE
PUT	PUTBLK	PUTNTA	
RANDOM	READ	RELEASE	RESET
RETRY	RETURN	REVIVE	ROTATE
RTOADR	RTOBUS	RTODAT	RTONTA
SAVE	SET	SHIFT	SHOW
SILENCE	SS1ADR	SS2ADR	SS3ADR
SS4ADR	SS5ADR	SS6ADR	SS7ADR
SS1DAT	SS2DAT	SS3DAT	SS4DAT
SS5DAT	SS6DAT	SS7DAT	SS1NTA
SS2NTA	SS3NTA	SS4NTA	SS5NTA
SS6NTA	SS7NTA	STATUS	STRING
SWAP			
TYPE			
VARIABLE	VECTOR	VERSION	
WAIT	WARN	WRITE	WRITELN
XOR			

APPENDIX B

COMMAND SYNTAX SUMMARY

This appendix lists the commands available within FDL in alphabetical order giving a brief description of the syntax. All key words are in upper case; user supplied arguments are in lower case. Optional switches and/or arguments are given in braces "{ }". VarN refers to an allocated variable, whereas valN refers to a variable or constant. Arrays of variables use the square bracket construct, var[a..b], to indicate the range of the operation.

ADDRESS {/CS}{/BR} val;

Address a FASTBUS module whose primary address is given by val with the mode specified by the switches /CS and /BR.

ALLOCATE {/GLOBAL} var1 {,var2,...varn};

Allocate a variable or variables. Local variables are allocated unless the /GLOBAL switch is used. Variables allocated at interactive level are always global.

ARBITRATE;

Arbitrate for FASTBUS mastership.

CALL procedure {(vall,...,valn)};

Transfer control to a new procedure with the option of passing arguments.

CASE val GOTO label0 {,label1,...,labeln};

Conditionally branch to a label based upon the value of val. If val is zero the branch is to label0; if one, the branch is to label1; etc. If the value is negative or if no label has been defined for the value, no branch is taken.

CLOSE {/TT}{/IN}{/OUT};

Close the I/O device specified by the given switch. Only one switch can be given per CLOSE command.

COMPARE {/TT}{/OUT} var1[m..n],var2[x..y];

Compare the contents two arrays. If any differences are detected, they are written to the output device given by the switch. If no output device is specified, no output is generated. The logical variable DIFFER is set to true if there are differences and false if the arrays contain the same values.

COMPRESS;

Compress internal FDL tables and files.

CONTINUE;

Continue a procedure which has been interrupted by either the PROMPT command or by the user typing a double control-C.

CREATE user name = system name;

Create a name which is a synonym for the given system name. The new name will perform the same action as the system name. The system name remains defined.

DEALLOCATE {/GLOBAL} var1 {,var2,...,varn};
DEALLOCATE {/GLOBAL} ALL;

Deallocate the specified allocated variables. Local variables are deallocated unless the /GLOBAL switch is used. This command always deallocates globals if issued from interactive level.

DELETE procedure;
DELETE ALL

Delete the specified procedure from the directory.
IF ALL is used, all defined procedures are deleted.

DIRECT;

Output the directory of defined procedures on the terminal.

EDIT procedure;

Edit the procedure given. See chapter 8 for a list of the editor commands.

END;

Ends the definition of a procedure. END is only legal if a procedure is being defined.

EXIT;

Exit FDL. Any files current active with the OPEN command are closed prior to exiting.

FBLOCK;

Begin the definition of a FASTBUS block. Only FASTBUS I/O commands are legal within an FBLOCK.

FEND;

End the current FBLOCK definition.

GET {/PARITY}/{/FIFO} var;

Read a single word from the FASTBUS module currently addressed and store the data in the given variable. The switches /PARITY and /FIFO can be used to modify the action of the command.

GETBLK {/PARITY}{/FIFO} var1[range],var2;

Read a block of words from the FASTBUS module currently addressed. The data is stored in the array given by var1. The range of the array implies the number of data words to read. The number of words read is returned in the variable given by var2. The switches /PARITY and /FIFO can be given to modify the action of the command.

GETNTA {/PARITY} var;

Read the contents of the NTA register from the FASTBUS module currently addressed. The data read is stored in the variable given by var. The switch /PARITY can be used to modify the action of the command.

GOTO label;

Transfer control to the specified label within the current procedure. GOTOs are only legal within procedures.

IF condition GOTO label;

If the given condition is true, transfer control to the given label. IF-GOTOs are only legal within procedures.

IN {/switch(s)} val,var;

Read a single data word from the FASTBUS module whose primary address is given by val. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. The data read is stored in the variable given by var. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switches /PARITY and /FIFO can be used to modify the action of the command.

INBLK {/switch(s)} val,var1[range],var2;

Read a block of data words from the FASTBUS module whose primary address is given by val. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. The data read is stored in the array given by var1. The number of words to read is implied by the range given on the array. The number of words read is stored in the variable given by var2. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switches /PARITY and /FIFO can be used to modify the action of the command.

INNNTA {/switch(s)} val1,val2,var;

Read a single data word from the FASTBUS module whose primary address is given by val1. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. After completing the primary address, a secondary address cycle is performed with the data taken from val2. The data received on the following read cycle is stored in the variable given by var. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switches /PARITY and /FIFO can be used to modify the action of the command.

INBLKNNTA {/switch(s)} val1,val2,var1[range],var2;

Read a block of data words from the FASTBUS module whose primary address is given by val1. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. After the primary address cycle, a secondary address cycle is performed with the data taken from val2. The data received on the following read cycle is stored in the array given by var1. The number of words to read is implied by the range given on the array. The number of words read is stored in the variable given by var2. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switches /PARITY and /FIFO can be used to modify the action of the command.

MAP {/TT}{/OUT} val1 {,val2,...,valn};

Scan FASTBUS logical groups given by val1 through valn, returning the id and geographical and logical addresses of each device found.

MODE BINARY;
MODE DECIMAL;
MODE HEX;
MODE OCTAL;

Changes the default output mode to the base binary, decimal, hexadecimal, or octal.

OPEN {/IN}{/OUT} "dev:file.ext";

Open the file specified by "dev:file.ext" for the logical device given by switches. The logical devices /IN and /OUT must be opened via the OPEN statement prior to use. The terminal, device /TT, is always considered open and does not need a specific open statement prior to use.

OUT {/switch(s)} val,var;

Write a single data word to the FASTBUS module whose primary address is given by val. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. The data to be written is taken from the variable given by var. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switch /FIFO can be used to modify the action of the command.

OUTBLK {/switch(s)} val,var1[range],var2;

Write a block of data words to the FASTBUS module whose primary address is given by val. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. The data to be written is taken from the array given by var1. The number of words to write is implied by the range given on the array. The number of words written is stored in the variable given by var2. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switch /FIFO can be used to modify the action of the command.

OUTNTA {/switch(s)} val1,val2,var;

Write a single data word to the FASTBUS module whose primary address is given by val1. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. After completing the primary address, a secondary address cycle is performed with the data taken from val2. The data to write is taken from the variable given by var. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switch /FIFO can be used to modify the action of the command.

OUTBLKNTA {/switch(s)} val1,val2,var1[range],var2;

Write a block of data words to the FASTBUS module whose primary address is given by val1. The mode used during the primary address cycle is specified by the optional switches /CS and /BR. After the primary address cycle, a secondary address cycle is performed with the data taken from val2. The data to write is taken from the array given by var1. The number of words to write is implied by the range given on the array. The number of words written is stored in the variable given by var2. If the switch /HOLD is used, the FASTBUS module remains addressed when the command completes. The switch /FIFO can be used to modify the action of the command.

PROCEDURE procedure {(var1,...varn)};

Begin defining a standard procedure. The FDL prompt changes from a "*" to a ">". All statements entered are parsed (syntax is checked) but not interpreted (not executed). The definition is terminated when the END statement is entered. The argument list defines the dummy parameter list thru which the user can pass arguments with the CALL statement.

PROMPT;

When used from within a procedure the PROMPT statement returns control to interactive level.

PURGE user name;

Purge the given user name from the name table. The name must have been defined via the CREATE command.

PUT {/FIFO} var;

Write a single word to the FASTBUS module currently addressed. The data to write is taken from the variable given by var. The switch /FIFO can be used to modify the action of the write.

PUTBLK {/FIFO} var1[range],var2;

Write a block of words to the FASTBUS module currently addressed. The data to write is taken from the array given by var1. The range of the array implies the number of data words to write. The number of words written is returned in the variable given by var2. The switch /FIFO can be given to modify the action of the command.

PUTNTA val;

Perform a secondary address cycle to the FASTBUS device currently addressed with the data given by val.

RANDOM var1 {,var2,...,varn};

Fill a variable(s) or an array of variables with data generated by a random number generator.

READ {/TT}{/IN} var1 {,var2,...,varn};

Input data values from the device specified by the switch into the variable list given. Values supplied must be integers in either decimal, hexadecimal, binary or octal form. If no switch is given, /TT is assumed. The system variable EOF\$INPUT is set to true when the end of the input file is reached.

RELEASE ADDRESS;

Release the FASTBUS module currently address.

RELEASE BUS;

Release FASTBUS mastership.

RESET BUS;

Reset FASTBUS by setting the FASTBUS RB line.

RESET INTERFACE;

Reset the FASTBUS interface to its power on state.

RETURN;

Return from a procedure call

REVIVE;

Resume the echoing of input from an indirect file.

SAVE {procedure1,...,proceduren} ON "dev:file.ext";

Save the specified procedures in the file "dev:file.ext". If no procedures are given, all defined procedures are saved in the file.

SET BURST val;
SET CLOCK val;
SET RETRY val;
SET VECTOR val;
SET ERWADR val;
SET ERWNTA val;
SET ERWDAT val;

Set the given system parameter to the value specified by val.

SET COMPRESS {ON}{OFF};
SET HOLDERR {ON}{OFF};
SET ERROR {ON}{OFF};
SET WARN {ON}{OFF};
SET STATUS {ON}{OFF};

Turn the given FDL switch ON or OFF.

SHOW ALL;
SHOW DETAIL;
SHOW DIRECT;
SHOW NAMES;
SHOW PARAMETER;
SHOW STATUS
SHOW STRING
SHOW VARIABLE;
SHOW VERSION;

Display the given information on the terminal.

SILENCE;

Turn off echoing of source line input from the indirect file.

TYPE {vall,...,valn};
TYPE {"string"};

Write the given values and/or strings to the terminal. A carriage return/line feed is output at the end of the execution of this command.

WAIT val

FDL will sleep for the specified amount of time. The time period to sleep is given in 1/60 of a second units.

WRITE {/TT}/{/OUT} {vall,...,valn};
WRITE {/TT}/{/OUT} {"string"};

Write the given values and/or strings to the output device given by the switch. If no device is specified, /TT is assumed. No carriage return/line feed is output at the end of this command.

WRITELN {/TT}/{/OUT} {vall,...,valn};
WRITELN {/TT}/{/OUT} {"string"};

Write the given values and/or strings to the output device given by the switch. If no device is specified, /TT is assumed. A carriage return/line feed is output at the end of this command.

APPENDIX C

KNOWN PROBLEMS AND BUGS

The following is a list of known bugs and deficiencies in FDL 3(74). These bugs may be fixed in future releases.

1. PASCAL-2 memory management errors

There is nothing which can be done about these errors except keep as much memory free for the PASCAL heap and stack. This problem could get better or worse with future developments in FDL.

2. Non-system devices references crash FDL

This is another related memory problem. If the handler for the device cannot fit into free memory, a bad fetch occurs. All FDL status, including defined procedures, is lost.

3. Code array overflow crashes FDL

The code array holds the pcode as it is generated during a procedure definition. If the code array becomes full, FDL prints the error message

```
>>>Code array overflow
>>>Saving all procedures
>>>Procedures saved on DK:PROCS.FDL
```

In future releases FDL will recover gracefully from such an error.

4. Working file overflows crash FDL

If the output, source, swap, or save files become full, FDL will crash in an ungraceful manner (no defined procedures are saved on the disk). In the future these overflows will be trapped and handled properly.

5. Array assignment problems

Certain array assignments are not legal, however the parser does not catch them as syntax errors. The user should NOT rely upon these statements functioning the same in future releases of FDL. The following are the statements in question.

1. `array[n..m] := val;`

Assigning a single value to an array with a range results in only the first entry in the array being assigned. The val argument should be enclosed in square brackets to indicate a range assignment.

2. `array := val;`

If an array is referenced without a subscript, a subscript of 0 is assumed (i.e. array is equivalent to `array[0]`). If the array was not allocated such that 0 was a legal subscript, a subscript out of bounds error will occur.

3. `var := [range];`

Variables are treated by FDL as an array with a range of `[0..0]`. Therefore any array references with a variable are acceptable. In the above case the variable will be assigned the first value given in the range.

APPENDIX D

FUTURE ADDITIONS TO FDL

The following are a examples of possible additions to FDL. This does not imply a commitment that such additions will be made. Suggestions for other changes, improvements and additions are welcome.

1. Fast diagnostic loops

It is possible with the UPI to create a list which contains an infinite loop. Status block output can also be suppressed. A new FDL structure similar to FBLOCK/FEND (i.e. DBLOCK/DEND) could be created to utilize these features for fast, infinite loops of FASTBUS cycles. Exit from the loop would be via control-C.

2. Database connections

Connect FDL into the system descriptor database to retrieve the geographical or logical address of a FASTBUS module by an ASCII name. The parser in FDL partially supports this feature with the GA and LA functions. GA and LA have not been defined in this document since they are not currently connected to any database but the names are reserved for future use. The intent of these functions is to return the appropriate FASTBUS primary address of a module whose ASCII name is given as an argument to GA or LA. For example:

```
var1 := GA("256K memory module");  
var2 := LA("256K memory module");
```

After executing the above two statements, var1 would contain the geographical primary address of the module and var2 would contain the logical primary address of the module.

3. EOF\$OUTPUT implemented

The intent of EOF\$OUTPUT is to provide a system variable and exception trap to indicate when the /OUT logical device becomes full. Under RT11, it is very likely that the output file can become full. PASCAL-2 however does not have any way of telling the user program when the output file is full (the eof function is always true). Perhaps this will change with future releases of the PASCAL-2 compiler.

4. Array to array assignments by range

Array to array assignments by range are illegal (single element reference are legal). For example:

```
array1[m..n] := array2[x..y];
```

Such a construct would be useful. Currently the arrays must be copied using a loop with an IF-GOTO.

5. Range assignments to arrays

Range assignments to arrays of the form [x..y] are illegal. For example:

```
array[m..n] := [x..y];
```

Such a construct would be useful for such diagnostics as address testing of memory modules.

APPENDIX E
EXAMPLES OF PROCEDURES

The following examples are used to demonstrate some of the features of FDL. These examples will run if entered into FDL, v3(74).

Procedure "factorial" will compute the factorial of a given number using recursion. The number is given as argument n; the result is returned through argument ans.

```
PROCEDURE factorial(n, OUT ans);
  IF (n > 1) goto L0;      -- 2 or greater
  ans := 1;                -- 0 and 1 are always 1
  RETURN;

<<L0>> ALLOCATE nminus1, tmp; -- get some variables
        nminus1 := n - 1;    -- get previous factorial
        CALL factorial(nminus1,tmp);
        ans := n * tmp;      -- compute the factorial
  END;
```

Procedures "mapsystem" and "mapgroup" map the system or a given group by attempting to read each geographical address. If a module responds, the id number read from CSR0 is displayed on the terminal.

```

PROCEDURE mapsystem;
ALLOCATE g,group;

    g :=0;                                -- Start with group zero
<<L0>> group := g SHIFT -24;              -- 8 bit SIs
    CALL mapgroup(group);                 -- Map the group
    g := g + 1;                           -- Next group please
    IF (g _< 256) GOTO L0;                 -- 256 groups to map
    END;

PROCEDURE mapgroup(group);
ALLOCATE pad,adr,id;
ON ALL CLEAR;                            -- Clear all external traps
ON RTOBUS GOTO L0;                        -- Arbitration error
ON SS1ADR GOTO L0;                        -- SI far side busy
ON SS3ADR GOTO L0;                        -- SI far side contention
ON FBERR GOTO L1;                         -- Skip if any other error

    group := group AND 0FFFFFF00h;
    adr := 0;                             -- Start with slot 0
<<L0>> pad := adr OR group;               -- Make a primary address
    INNTA/CS pad,0,id;                    -- Read the device id
    TYPE "Address ",pad:H:10," has device ",id:H:10;
<<L1>> adr := adr + 1;                    -- Move to the next address
    IF (adr _< 256) GOTO L0;              -- 256 slots per group
    RELEASE BUS;
    END;

```

Procedures "adrtst" and "adrmem" are used together as a general purpose memory test. Argument adr is the primary address of the module to test. Argument bsize is the size of the buffer to use; the larger bsize, the faster the diagnostic will run. Argument msize is the size of the memory in 32-bit words.

```

PROCEDURE adrtst(adr,bsize,msize);
  ALLOCATE mem;

  TYPE "Address test";
  mem := 0;
<<L0>> IF ((mem + bsize) <= msize) GOTO L1;
      bsize := (msize - mem);
<<L1>> CALL adrmem(adr,mem,bsize);
      mem := mem + bsize;
      IF (mem < msize) GOTO L0;
      TYPE "End of test";
      END;

PROCEDURE adrmem(adr,memadr,bsize);
  ALLOCATE b[1..bsize],c[1..bsize],i;

  i := 1;                -- Initialize the increment
<<L0>> b[i] := (memadr+i-1); -- Address in address
      i := i + 1;         -- Increment the index
      IF (i < bsize) GOTO L0; -- Loop for all

<<L1>> OUTBLKNTA adr,memadr,b[1..bsize],i;
      IF (NOT FBERR) GOTO L2; -- Branch if no errors
      IF RTOBUS GOTO L1;     -- Arbitration error
      IF SS3ADR GOTO L1;     -- SI contention
      TYPE "Output FASTBUS error ",ERR$STATUS:H;
      IF ((NOT RTOADR) AND (NOT SS2ADR)) GOTO L1;
      PROMPT;

<<L2>> c[1..bsize] := [0];    -- Initialize input to zero
      INBLKNTA adr,memadr,c[1..bsize],i;
      IF (NOT FBERR) GOTO L3; -- Branch if no errors
      IF RTOBUS GOTO L2;     -- Arbitration error
      IF SS3ADR GOTO L2;     -- SI contention error
      TYPE "Input FASTBUS error ",ERR$STATUS:H;
      IF ((NOT RTOADR) AND (NOT SS2ADR)) GOTO L2;
      PROMPT;

<<L3>> COMPARE/TT b[1..bsize],c[1..bsize];
      END;

```

APPENDIX F

ERROR MESSAGES

The following is a list of the error messages which FDL will output to the terminal. A brief explanation of the error, a possible cause for the message, and the appropriate action the user should take are given with each error message.

>>>Array range discrepancy, smaller range used

The COMPARE command expects as its arguments two arrays whose given ranges are equal. If the ranges are not the same size, this message is output and the smallest range is used for the comparison length.

>>>Attempt to deallocate non-existent variable

An attempt was made to deallocate a variable which had never been allocated.

>>>Branch statements only allowed in procedures

Branch statements, like labels, can only be used within a procedure. The statement is ignored.

>>>Cannot ALLOCATE/DEALLOCATE system variables

An attempt was made to ALLOCATE or DEALLOCATE a system variable. System variable names cannot be used for local variables. They can never be DEALLOCATED.

>>>Cannot close device /TT

The /TT logical device cannot be closed for any reason.

>>>Cannot continue

The CONTINUE command must be the first command issued after a procedure has been interrupted. If any other command is issued, the procedure context is destroyed and cannot be continued.

>>>Cannot create new names when procedures are defined

The CREATE command can only be used if no procedures are defined. Use the DELETE ALL statement to remove all defined procedures before attempting to CREATE new commands.

>>>Cannot DEALLOCATE procedure parameter variables

Procedure arguments are allocated as local variables when the procedure is executing. These local variables cannot be deallocated with the DEALLOCATE command. They are automatically deallocated when the procedure is exited.

>>>Cannot open device /TT

The /TT logical device is always open for input and output to the terminal. This cannot be changed with the OPEN command.

>>>Code array overflow

The incore array is too small for the procedure being defined. The procedure must be split into two or more smaller procedures before they can be defined to FDL. This is a fatal error which causes FDL to exit to RT11.

>>>Constant overflow

There were too many characters in the constant beginning entered. FDL allows for 40 characters. Since the maximum number of characters which every need be entered is 33 (binary mode plus 32 digits), this error should never occur unless the user is a wiseguy. Solution, stop trying to break FDL.

>>>Data array processed in ascending order

An array was given to a FASTBUS block transfer command with its first subscript larger than the second subscript. FASTBUS commands always read or write data from the array in ascending order.

>>>DK:ERRORS.FDL not found

The file ERRORS.FDL is required on the device DK:. FDL will not run until this file (which is distributed with FDL) is supplied.

>>>Directory is empty

A DIRECT command was issued and no procedures were defined.

>>>Division by zero

An attempt was made to divide by zero. The result of the operation is zero. FDL only issues this message as a warning.

>>>Duplicate label in procedure

The label being defined has already been defined elsewhere within the procedure. Label names must be unique within a procedure. The procedure must be corrected with the editor before the procedure can be invoked.

>>>EDIT file full

There is no room to insert any lines in the procedure being edited. To increase the size of the edit file, the user must return to the RT11 monitor, delete unneeded files and squeeze the disk to create a larger free hole.

>>>EDIT file too small for procedure

The EDIT file is too small for the procedure to be edited. The user must free space on the RT11 device being used by deleting unused files and squeezing the disk.

>>>error N

An internal error of type N has occurred. Please contact the maintainer of FDL and give the details surrounding the cause of this error.

>>>Expecting ,

A comma was expected as a separator between two arguments. The statement must be reentered with a comma at the indicated location.

>>>Expecting ;

FDL was expecting an end of statement character, a semicolon, but found some other character. The statement must be reentered and terminated with a semicolon.

>>>Expecting :=

FDL was parsing an assignment statement but did not find the assignment symbol after a variable name. This can be caused by mistyping an FDL command name. The statement must be reentered with either corrected spelling or a := after the variable name.

>>>Expecting]

FDL was parsing an array subscript or range, but a terminating right bracket was not found. The statement must be reentered with a right bracket.

>>>Expecting] or ..

FDL was parsing an array subscript but did not find a right bracket to terminate the subscript nor a .. construct indicating a range. The statement must be reentered with a square bracket in the correct location.

>>>Expecting (

Some commands (i.e. GA and LA functions) require an argument enclosed in parentheses. The parser did not see a left parenthesis at the location indicated by the uparrow. The statement must be reentered.

>>>Expecting (or ;

This error can occur for CALL and PROCEDURE statements if the procedure name was followed by something other than a (or ;. Most likely the semicolon has been forgotten. The statement must be reentered and terminated correctly.

>>>Expecting)

FDL was parsing an argument list for a procedure and either saw a semicolon or saw two arguments which were not separated by a comma. The statement must be reentered correcting the mistake.

>>>Expecting >>

FDL was parsing a label but did not see the label definition terminator. The statement must be reentered with a correct syntax.

>>>Expecting a SET key (BURST, ERWADR, ERROR, etc)

FDL was parsing a SET command but did not find a legal SET key. This could be caused by a misspelling. The statement must be reentered with a legal key.

>>>Expecting a SHOW key (ALL, DIRECT, etc)

FDL was parsing a SHOW command but did not find a legal SHOW key. This could be caused by a misspelling. The statement must be reentered with a legal key.

>>>Expecting a string

A string was expected by the parser but was not found. Commands such as @ and SAVE require a string within the statement. The statement must be reentered with a string at the correct location.

>>>Expecting an identifier

This error will occur if a statement does not begin with a symbolic name. This can occur if a control character (i.e. control-A) has been typed, or if the name begins with a numeric character (names must begin with alpha characters only). The statement must be reentered with a correct name.

>>>Expecting BUS or INTERFACE

The RESET command takes only two possible keys, BUS and INTERFACE, as its argument. If neither one of these keys was found, this error will result. The statement must be reentered with a correct key.

>>>Expecting condition (FBERR, SSINTA, RTOADR, etc)

The ON command requires a condition (or a list of conditions) as its argument. This error is most commonly caused by a misspelling. The statement must be reentered with a legal condition.

>>>Expecting constant

Only a constant is allowed at the position in the statement indicated by the uparrow. This could be caused by a misspelling, illegal character, or forgetting to begin a hexadecimal constant with a numeric character (i.e. 0FFh). The statement must be reentered with a legal constant.

>>>Expecting constant, variable name, or data base request

The statement requires one of the above at the location indicated by the uparrow. This error can be the result of an assignment statement which terminated immediately after the :=. The statement must be reentered with a correct syntax.

>>>Expecting GOTO

A GOTO key is required in IF and ON commands. Only the GOTO command is allowed in a conditional. IF-WRITE, etc, are not possible. A branch around the statement which is to be conditional must be done with an IF-GOTO.

>>>Expecting I/O device (i.e. /TT, /OUT)

FDL expected a logical device switch after the command. The statement must be reentered with a logical device switch.

>>>Expecting label name

All branch statements (GOTO, IF-GOTO, ON-GOTO) require a label name be given. This error will result if a legal name is not found in the correct position within the statement. The statement must be reentered with a legal label name.

>>>Expecting mode indicator (H,O,B,D)

The tag after the colon on the WRITE, WRITELN and TYPE commands can only be one of four possible letters. The statement must be reentered with the proper output mode letter.

>>>Expecting ON

The SAVE command requires the ON key between the procedure list and the string which contains the RT11 file specification. The statement must be reentered with the ON key in the correct position.

>>>Expecting ON or OFF

Some SET commands require the ON or OFF key as the second argument. The statement must be reentered with the correct key.

>>>Expecting output mode (ie OCTAL, DECIMAL)

The MODE command requires one of four different key words as its argument. Either a key was not given or the key was misspelled. The statement must be reentered with a correctly spelled key.

>>>Expecting procedure name

Many FDL commands (CALL, EDIT, DELETE, etc) require the name of a procedure. This error only indicates that a symbolic name which COULD be a procedure was not found at the correct place within the statement and not that the procedure is not defined. This error can occur if a control character has been entered or if the name does not begin with an alpha character. The statement must be reentered with a correct name.

>>>Expecting variable name

A variable is required at the location indicated by the uparrow. Some FDL commands will not accept a constant for certain arguments. This error can also be caused by an illegal name. This error does not imply that a variable has not been allocated. The statement must be reentered with a variable at the location indicated.

>>>Expression stack overflow

The expression was so complex the expression stack, which FDL uses to compute the expression, was too small. The expression must be split into several statements.

>>>Expression stack underflow

This is an internal FDL error. Please contact the maintainer.

>>>GA and LA database functions not implemented

The GA and LA database functions are currently not implemented. Please refrain from attempting to use the feature until they are.

>>>Illegal argument for EDIT command

The argument given with the editor command is illegal for that command.

>>>Illegal EDIT command

The command issued while in the editor is not a legal command. The editor must be exited before interactive FDL commands can be used.

>>>Illegal replacement variable for OUT parameter

Procedure parameters which are prefaced with the OUT key word, must be replaced in the CALL command with a variable. Constants cannot be used. The CALL statement must be reentered without a constant where any OUT parameters are used.

>>>Illegal sequence of operations

This error will occur if FASTBUS data I/O primitives (GET, PUT, etc) are used when there is no FASTBUS module currently addressed.

>>>Illegal statement in FBLOCK

Only FASTBUS related commands can be used within the body of an FBLOCK. The FEND command must be issued before other types of statements can be used.

>>>Incorrect number of arguments

The CALL statement does not have the same number of arguments as the procedure it is calling. The CALL statement must be reentered with the correct number of arguments.

>>>Indirect file not found

FDL was unable to open the indirect file given with the @ command. Either the file name has been misspelled or the file does not exist on the specified RT11 device.

>>>Indirect files nested too deep

Nested indirect files are limited to a given maximum, usually three. An attempt was made to go beyond this limit.

>>>Input device not allowed

The /IN logical device is not allowed with output only commands (i.e. WRITE).

>>>Input file not found

The file specification given in the OPEN statement was not found on the given RT11 device.

>>>Input file not open

A reference was made to the /IN logical device, however it was never opened with the OPEN command.

>>>Insufficient memory for allocation

There is not enough memory to allocate the variable. The number of words available for allocation is given along with this error message. Some variable(s) must be deallocated before this variable can be allocated.

>>>Interface timeout

The FASTBUS master interface does not response. This is most likely due to a hardware failure in the interface.

>>>Invalid character in constant

A character was present in a constant which is not allowed in the radix indicated. For example, 0Fb, will generated this error. This error usually occurs when a hexadecimal constant is being entered and the "h" which indicates hexadecimal is forgotten. The statement must be reentered with a legal constant.

>>>Invalid constant (read)

An invalid constant was entered to the READ command. The constant is ignored and the READ command attempts to read a new constant.

>>>Labels must be within procedures

Labels can only be used within procedures. No branching is allowed at the interactive level of FDL. The statement is ignored.

>>>Line truncated after 85 characters

The maximum number of characters per input line is 85 characters. All characters beyond this limit are ignored.

>>>List buffer storage exceeded

The FBLOCK structure contained more FASTBUS statements than there was room for in the list buffer. The FBLOCK must be split into smaller blocks.

>>>Name in use as a procedure or variable

The name to be defined with the CREATE command must not be in use for other FDL structures. The name must first be freed before it can be defined as a command.

>>>No changes to procedure

This message is output when the editor is exited and no changes have been made to the procedure which was edited.

>>>No such procedure

A reference was made to a procedure which has not been defined. Either the procedure must first be defined or the reference must be changed.

>>>Non-existent memory trap

The FASTBUS master interface attempted to access non-existent memory. Please contact the maintainer. This could be the result of a computer hardware problem.

>>>Not a user defined system name

The name given in the PURGE command was not defined with the CREATE command.

>>>Not defining a procedure

This error message is generated when the END command is issued outside the body of a procedure.

>>>Not defining FBLOCK

The FEND command was issued outside the body of an FBLOCK.

>>>Output device not allowed

The /OUT logical device is not allowed with input only commands (i.e. READ).

>>>Output file not open

A reference was made to the /OUT logical device, however it was never opened with the OPEN command.

>>>Overflow in arithmetic operation

FDL only supports 32-bit signed integers. An expression created a number which required more than 32-bits of information. The expression must be changed to prevent the overflow.

>>>Parse stack not empty

This is an internal error. Please contact the maintainer.

>>>Parse stack overflow

The parse stack is too small for the statement which is being parsed. This error can occur if the statement contains an exceptionally large number of arguments. A range assignment to an array which contains a large number of entries may also cause this error. The statement should be split into several statements. The statement which generated this error is ignored.

>>>Procedure abort by control-C

The procedure was interrupted because two consecutive control-Cs were typed on the terminal. The procedure can resume execution if the CONTINUE command is issued.

>>>Procedure already exists

An attempt was made to define a procedure which was already defined. A different name must be used for the procedure or the currently defined procedure must be deleted.

>>>Procedure contains syntax errors

One or more parsing errors occurred while the procedure was being defined. The procedure cannot be invoked until the errors are corrected and reparsed.

>>>Procedures out of order during compression

This is an internal FDL error. Please contact the maintainer.

>>>Procedures nested too deeply

The maximum number of nested CALLs which can be performed is limited to the depth of the stack. An attempt was made to go deeper into calls than the stack allowed. The depth of the stack can be found with the SHOW STATUS command.

>>>Semantic stack not empty

This is an internal error. Please contact the maintainer.

>>>Semantic stack overflow

This error can occur for the same reasons as the "Parse stack overflow". The statement must be split into several smaller statements.

>>>String truncated after 85 characters

The maximum number of characters per string is 85 characters. All characters beyond this limit are ignored.

>>>Statement not allowed in a procedure definition

Not all statements are allowed within the body of a procedure. The procedure must be terminated with the END command before the statement can be entered.

>>>Statement not allowed in an indirect file

Some statements (EDIT, CONTINUE) cannot be used within an indirect file. The statement is ignored.

>>>String file full

The string file has become full. The user should issue the COMPRESS command to delete any unused strings. If all strings are in use, a procedure must be deleted and a compression performed before the statement which generated this error can be issued.

>>>Subscript out of range

An array was referenced with a subscript which is beyond the defined bounds of the array. The array name is given after this message.

>>>Unable to open CODE file

FDL was unable to open the temporary work file for storing the pcode. Disk space must be freed before FDL can be used.

>>>Unable to open EDIT file

FDL was unable to open an EDIT work file. This is usually caused by a full directory on the RT11 device being used. More space must be freed on this disk before the editor can be used.

>>>Unable to open output file

FDL was unable to open the given file for output. This can be caused by many RT11 related problems (i.e. the directory being full). It can also be caused if the RT11 device specified is an input only device (ie the card reader).

>>>Unable to open SOURCE file

FDL was unable to open the temporary work file to hold the source lines. Disk space must be freed before FDL can be used.

>>>Unable to open STACK file

FDL was unable to open the temporary work file for the CALL/RETURN stack. Disk space must be freed before FDL can be used.

>>>Unable to open STRING file

FDL was unable to open the temporary work file for storing strings. Disk space must be freed before FDL can be used.

>>>Unresolved label (GOTO, IF-GOTO, ON-GOTO)

A reference was made within the procedure to a label which was never defined. The procedure must be edited and the label defined before the procedure can be invoked.

>>>User names can only be created from system names

The second argument of the CREATE must be a system reserved name.

>>>Variable already allocated

An attempt was made to allocate a variable which had already been allocated. The variable must first be deallocated before it can be reallocated (i.e. to change the bounds of an array). Parameters within a procedure are considered variables local to that procedure and cannot be allocated within the procedure.

>>>Variable not allocated

A reference was made to a variable which was never defined. This error usually occurs because of a misspelled name. Either the reference must be changed or the variable must be defined.

>>>Wait failed (lacks monitor support)

The WAIT command failed either because the .MRKT program request, an RT11 system service, is not supported, or there were not enough queue elements. Contact the maintainer for more information.

>>>Wait value cannot be negative

The value of the argument for the WAIT command cannot be negative. Only values between 0 and 07FFFFFFh are allowed.

INDEX

!	3-9, 3-19
#	3-9, 3-19
&	3-9, 3-19
*	3-9, 3-11
+	3-9, 3-12
-	3-9, 3-12
/GLOBAL	6-4 to 6-5
/IN	5-1 to 5-3, 5-5
/OUT	5-1 to 5-4, 5-7, 5-9
/SILENCE	7-4
/TT	5-1, 5-4 to 5-5, 5-7, 5-9
<	3-9, 3-16
<=	3-9, 3-16
<>	3-9, 3-17
=	3-9, 3-17
>	3-9, 3-16
>=	3-9, 3-17
@	7-4
ADDRESS	4-12, 4-14, 4-18, A-1, B-1, B-8
ALL	7-20, A-1, B-10
ALLOCATE	3-2, 6-4, A-1, B-1
AND	3-9, 3-18, A-1
ARBITRATE	4-12, 4-14, A-1, B-1
Arithmetic	3-8
Array	
allocation	3-2, 6-4
assignment	3-5
deallocation	3-4, 6-5
range	3-6
Assignment statement	2-2, 3-1, 3-5
BINARY	5-6, A-1, B-6
BURST	4-26, A-1, B-9
BUS	4-12, 4-18, 4-28, A-1, B-8
CALL	6-1 to 6-2, 6-6, A-1, B-1
CASE	A-1
CASE-GOTO	6-8, 6-11, B-2
CLEAR	A-1
CLOCK	4-26, A-1, B-9
CLOSE	5-1, 5-3, A-1, B-2
COM	3-9 to 3-10, A-1
Command statement	2-2
Command statements	
FASTBUS	4-1
I/O	5-1
procedures	6-1
utilities	7-1

Comments	2-3
COMPARE	5-1, 5-7, A-1, B-2
COMPRESS	7-10, A-1, B-2, B-9
Compression	
automatic	7-10
manual	7-10
Constants	3-1, 3-5, 3-7
CONTINUE	6-17, A-1, B-2
CREATE	7-11, A-1, B-2
DEALLOCATE	3-4, 6-5, A-1, B-2
DECIMAL	5-6, A-1, B-6
DELETE	7-3, A-1, B-3
DETAIL	7-14, A-1, B-10
DIFFER	5-7, 6-13 to 6-14, A-1
DIRECT	7-2, 7-13, A-1, B-3, B-10
Display strings	3-20
Display variables	3-20
DIV	3-9, 3-11, A-1
DK:FDLINI.FDL	7-4
EDIT	8-1 to 8-2, A-1, B-3
Editor	
B	8-4
beginning	8-4
commands	8-3
E	8-8
errors	8-11
exit	8-8
files	8-11
H	8-10
help	8-10
I	8-6
insert	8-6
invoking	8-2
L	8-5
list lines	8-5
M	8-4
move	8-4
R	8-7
replace	8-7
END	6-2, A-1, B-3
EOF\$INPUT	5-5, 6-13, A-1
EOF\$OUTPUT	6-13, A-1
EQ	3-9, 3-15, A-1
ERR\$STATUS	4-21, A-1
ERROR	4-22, A-1, B-9
ERWADR	4-27, A-1, B-9
ERWDAT	4-27, A-1, B-9
ERWNTA	4-27, A-1, B-9
Exception traps	6-12, 6-14
EXIT	7-9, A-1, B-3
Expressions	3-1, 3-5
False	3-8

FASTBUS

command syntax	4-3, 4-12
complete operation commands	4-1 to 4-2
hold bus on error	4-25
list operations	4-1, 4-19
messages	4-22 to 4-23
order of operations	4-5
primitive commands	4-1, 4-11
status	4-1
system parameter block	4-26, 7-18
system variables	4-20 to 4-21
FASTBUS exceptions	6-12, 6-14
FASTBUS master interfaces	1-1
FASTBUS messages	
error	4-22
status	4-22
warning	4-22
FASTBUS Standard Routines	1-1
FBADR	4-20, 6-13, A-1
FBBUS	4-20, 6-13, A-1
FBERR	4-20, 6-13, A-1
FBLOCK	4-1, 4-19, A-1, B-3
FBPAR	4-20, 6-13, A-1
FBWRN	4-20, 6-13, A-1

FDL

command syntax	B-1
compression	7-10
data types	3-1
directory	7-2 to 7-3
editor	8-1
environment	1-1
error messages	F-1
examples	E-1
exit	7-9
features	1-2
futures	D-1
how to use	2-1
indirect files	7-4
introduction	1-1
known problems	C-1
legal names	3-2
prompt	2-1, 2-3, 4-19, 6-2
reserved names	A-1
status	7-19
version	7-17
FEND	4-19, A-1, B-3

GA	A-1
GE	3-9, 3-15, A-1
GET	4-12, 4-17, A-1, B-3
GETBLK	4-12, 4-16, A-1, B-4
GETNTA	4-12, 4-15, A-1, B-4
GOTO	6-8 to 6-9, A-1, B-4
GT	3-9, 3-14, A-1

HEX 5-6, A-1, B-6
 HOLDERR 4-25, A-1, B-9

 I/O statements 5-1
 IF A-1
 IF-GOTO 6-8, 6-10, B-4
 IN 4-2 to 4-3, 4-7, A-1, B-4
 INBLK 4-3, 4-8, A-1, B-5
 INBLKNTA 4-3, 4-8, A-1, B-5
 Indirect files 7-4, 7-6 to 7-7
 INNNTA 4-3, 4-7, A-1, B-5
 INTERFACE 4-28, A-1
 Interpretive phase 2-1, 2-5
 Introduction 1-1
 IORFI 1-1

 LA A-1
 Labels 6-8 to 6-12
 LE 3-9, 3-14, A-1
 Logical device
 /IN 5-1
 /OUT 5-1, 5-7, 5-9
 /TT 5-1, 5-7, 5-9
 Logical device /IN 5-2 to 5-3, 5-5
 Logical device /OUT 5-2 to 5-4
 Logical device /TT 5-4 to 5-5
 Logical value 3-8
 LT 3-9, 3-14, A-1

 MAP 5-1, 5-9, A-1, B-5
 MOD 3-9, 3-12, A-1
 MODE 5-6, A-1
 BINARY 5-6, B-6
 DECIMAL 5-6, B-6
 HEX 5-6, B-6
 OCTAL 5-6, B-6

 NAMES 7-16, A-1, B-10
 NE 3-9, 3-15, A-1
 NOT 3-9 to 3-10, A-1

 OCTAL 5-6, A-2, B-6
 OFF A-2
 ON A-2
 ON-CLEAR 6-12
 ON-GOTO 6-8, 6-12
 OPEN 5-1 to 5-2, A-2, B-6
 Operator
 ! 3-9, 3-19
 # 3-9, 3-19
 & 3-9, 3-19
 * 3-9, 3-11
 + 3-9, 3-12
 - 3-9, 3-12
 < 3-9, 3-16

<=	3-9, 3-16
<>	3-9, 3-17
=	3-9, 3-17
>	3-9, 3-16
>=	3-9, 3-17
AND	3-9, 3-18
COM	3-9 to 3-10
DIV	3-9, 3-11
EQ	3-9, 3-15
GE	3-9, 3-15
GT	3-9, 3-14
LE	3-9, 3-14
LT	3-9, 3-14
MOD	3-9, 3-12
NE	3-9, 3-15
NOT	3-9 to 3-10
OR	3-9, 3-18
priorities	3-8
ROTATE	3-9, 3-13
SHIFT	3-9, 3-13
SWAP	3-9 to 3-10
types	3-8
unary minus	3-9, 3-11
unary plus	3-9, 3-11
XOR	3-9, 3-18
Operators	3-1, 3-5, 3-8
OR	3-18, A-2
OUT	4-2 to 4-3, 4-9, 6-6, A-2, B-6
OUTBLK	4-3, 4-10, A-2, B-6
OUTBLKNTA	4-3, 4-10, A-2, B-7
OUTNTA	4-3, 4-9, A-2, B-7
Output	
default mode	5-6
radix	3-21
strings	3-23, 7-17
width	3-22
PARAMETER	7-18, A-2, B-10
Parsing	
assignment statement	2-4
command statement	2-4
comments	2-3
errors	2-3
multiple lines	2-3
Parsing phase	2-1, 2-3
PROCEDURE	6-1 to 6-2, A-2, B-7
Procedure	
branching	6-8 to 6-12
continue	6-17
delete	7-3
directory	7-2, 7-13 to 7-14
edit	8-1
end	6-2, 6-9
exception trapping	6-12

exit	6-15
halt	6-17
interrupt	6-17
invoking	6-2
labels	6-8, 6-10 to 6-12
names	6-2
parameters	6-6
prompt	6-15
recursion	6-2, 6-16
return	6-15
save	7-8
variables	6-4 to 6-5
Procedures	6-1 to 6-2
PROMPT	6-15, A-2, B-7
PURGE	7-12, A-2, B-7
PUT	4-12, 4-17, A-2, B-8
PUTBLK	4-12, 4-16, A-2, B-8
PUTNTA	4-12, 4-15, A-2, B-8
 RANDOM	 3-24, A-2, B-8
Range	
assignment	3-6
expression	3-6
READ	5-1, 5-5, A-2, B-8
RELEASE	4-12, 4-18, A-2, B-8
ADDRESS	4-12, 4-18, B-8
BUS	4-12, 4-18, B-8
RESET	4-28, A-2
BUS	4-28, B-9
INTERFACE	4-28, B-9
RETRY	4-27, A-2, B-9
RETURN	6-15, A-2, B-9
REVIVE	7-7, A-2, B-9
ROTATE	3-9, 3-13, A-2
RTll devices	5-1
RTll files	5-1 to 5-2, 7-4, 7-8
RTOADR	4-20, 6-13, A-2
RTOBUS	4-20, 6-13, A-2
RTODAT	4-20, 6-13, A-2
RTONTA	4-20, 6-13, A-2
 SAVE	 7-8, A-2, B-9
SET	4-22, 4-25 to 4-27, 7-10, A-2, B-9
BURST	4-26, B-9
CLOCK	4-26, B-9
COMPRESS	7-10, B-9
ERROR	4-22, B-9
ERWADR	4-27, B-9
ERWDAT	4-27, B-9
ERWNTA	4-27, B-9
HOLDERR	4-25, B-9
RETRY	4-27, B-9
STATUS	4-22, B-9
VECTOR	4-26, B-9

WARN	4-22, B-9
SHIFT	3-9, 3-13, A-2
SHOW	7-13 to 7-20, A-2
ALL	7-20, B-10
DETAIL	7-14, B-10
DIRECT	7-13, B-10
NAMES	7-16, B-10
PARAMETER	7-18, B-10
STATUS	7-19, B-10
STRING	7-17, B-10
VARIABLE	7-15, B-10
VERSION	7-17, B-10
SILENCE	7-6, A-2, B-10
SS1ADR	4-20, 6-13, A-2
SS1DAT	4-20, 6-13, A-2
SS1NTA	4-20, 6-13, A-2
SS2ADR	4-20, 6-13, A-2
SS2DAT	4-20, 6-13, A-2
SS2NTA	4-20, 6-13, A-2
SS3ADR	4-20, 6-13, A-2
SS3DAT	4-20, 6-13, A-2
SS3NTA	4-20, 6-13, A-2
SS4ADR	4-20, 6-13, A-2
SS4DAT	4-20, 6-13, A-2
SS4NTA	4-20, 6-13, A-2
SS5ADR	4-20, 6-13, A-2
SS5DAT	4-20, 6-13, A-2
SS5NTA	4-20, 6-13, A-2
SS6ADR	4-20, 6-13, A-2
SS6DAT	4-20, 6-13, A-2
SS6NTA	4-20, 6-13, A-2
SS7ADR	4-20, 6-13, A-2
SS7DAT	4-20, 6-13, A-2
SS7NTA	4-20, 6-13, A-2
Statement	
assignment	2-2, 3-1, 3-5
command	2-2
labels	6-8
STATUS	4-22, 7-19, A-2, B-9 to B-10
STRING	7-17, A-2, B-10
Strings	3-20, 3-23
SWAP	3-9 to 3-10, A-2
Syntax errors	2-3
System defined names	7-11 to 7-12, 7-16
System parameter block	4-26, 7-18
System variables	
DIFFER	5-7
EOF\$INPUT	5-5
FBADR	4-20
FBBUS	4-20
FBERR	4-20
FBPAR	4-20
FBWRN	4-20
RTOADR	4-20
RTOBUS	4-20

RTODAT	4-20
RTONTA	4-20
SS1ADR	4-20
SS1DAT	4-20
SS1NTA	4-20
SS2ADR	4-20
SS2DAT	4-20
SS2NTA	4-20
SS3ADR	4-20
SS3DAT	4-20
SS3NTA	4-20
SS4ADR	4-20
SS4DAT	4-20
SS4NTA	4-20
SS5ADR	4-20
SS5DAT	4-20
SS5NTA	4-20
SS6ADR	4-20
SS6DAT	4-20
SS6NTA	4-20
SS7ADR	4-20
SS7DAT	4-20
SS7NTA	4-20
True	3-8
TYPE	3-20, 5-6, A-2, B-10
output radix	3-21
output strings	3-23
output width	3-22
Unary minus	3-9, 3-11
Unary plus	3-9, 3-11
UNIBUS Processor Interface	1-1
UPI	1-1, 4-1, 4-21, 4-26
User defined names	7-11 to 7-12, 7-16
Utilities	7-1
VARIABLE	7-15, A-2, B-10
Variable	
allocation	3-2, 6-4, 7-15
assignment	2-2, 3-5
deallocation	3-4, 6-5
display	3-20
global	6-4 to 6-5
local	6-4 to 6-5
Variables	3-1, 3-5
VECTOR	4-26, A-2, B-9
VERSION	7-17, A-2, B-10
WAIT	7-9, A-2, B-10
WARN	4-22, A-2, B-9
WRITE	5-1, 5-4, 5-6, A-2, B-10
WRITELN	5-4, 5-6, A-2, B-10
XOR	3-9, 3-18, A-2