9100 Series

TL/1 Reference Manual

PN 899906
APRIL 1991
©1991 John Fluke Mfg. Co., Inc.
All rights reserved. Litho in U.S.A.

FLUKE®

CUSTOMER NOTICE

THROUGHOUT THIS MANUAL, ALL INSTANCES OF 9100A AND 9105A ALSO APPLY TO THE 9100FT AND 9105FT.

Contents

Section	Title	Page
	Where Am I?	**xi**
1.	**Overview**	**1-1**
1.1.	INTRODUCTION	1-1
1.2.	ORGANIZATION OF THIS MANUAL	1-2
2.	**TL/1 Language Conventions**	**2-1**
2.1.	NAME CONVENTIONS	2-2
2.1.1.	File and Directory Names	2-3
2.1.2.	Program Names	2-4
2.1.3.	Device Names	2-4
2.1.4.	Device List	2-6
2.1.5.	Reference Designator Names	2-7
2.1.6.	Ref Pin Names	2-7
2.2.	DATA TYPES	2-8
2.2.1.	Numeric	2-8
2.2.2.	Floating-Point	2-9
2.2.3.	String	2-10
2.3.	ARRAYS	2-11
2.4.	OPERATORS	2-11
2.4.1.	Arithmetic Operators	2-12
2.4.2.	Relational Operators	2-12
2.4.3.	Logical Operators	2-13
2.4.4.	String Operators	2-14
2.4.5.	String Functions	2-14
2.4.6.	Bit Shifting Operators	2-15
2.4.7.	Bit Mask Operators	2-15

2.5. ORDER OF EVALUATION OF OPERATORS 2-16
2.6. CONDITIONAL EXPRESSIONS 2-17
2.7. FUNCTIONS 2-20
2.7.1. Special Functions 2-20
2.7.2. Pod Functions 2-20
2.7.3. I/O Module and Probe Functions 2-20
2.7.4. Type Conversion Functions 2-21
2.8. TL/1 STATEMENT CONVENTIONS 2-22

3. TL/1 Alphabetical Reference 3-1

TL/1 Commands are listed alphabetically

Appendices

A. ASCII Codes A-1

B. Control Codes for Monitor and Operator's Display B-1

B.1. ERASING B-1
B.2. CURSOR CONTROL SEQUENCES B-2
B.3. DISPLAY ATTRIBUTES B-2
B.4. DISPLAY MODE SEQUENCES B-2
B.5. TAB STOPS B-3
B.6. EDITING CONTROL B-3
B.7. ANNUNCIATOR CONTROL B-3
B.8. BEEPER CONTROL B-4
B.9. SPECIAL DISPLAY CODES FOR THE OPERATOR'S DISPLAY B-4
B.10. DISPLAY CHARACTERS FOR THE MONITOR B-5

C. Operator's Keypad Mapping to TL/1 Input C-1

D. Programmer's Keyboard Mapping to TL/1 Input D-1

Section Title Page

E. I/O Module Clip/Pin Mapping E-1

F. TL/1 Reserved Words F-1

G. Handling Built-In Fault Conditions G-1

G.1. OVERVIEW G-1
G.2. ARGUMENT NAMES G-2
G.3. RAM TEST FAULT CONDITIONS G-4
G.4. ROM TEST FAULT CONDITIONS G-7
G.5. BUS TEST FAULT CONDITIONS G-8
G.6. MEMORY INTERFACE POD FAULT CONDITIONS G-9
G.7. GENERIC FAULT CONDITIONS G-10
G.8. PRIMITIVE FAULT CONDITIONS G-10
G.9. I/O FAULT CONDITIONS G-11
G.10. ARGUMENTS USED WITH BUILT-IN TESTS G-12

H. Generating Built-In Fault Messages H-1

H.1. OVERVIEW H-1
H.1.1. Symbols H-2
H.1.2. Message Variables H-3
H.1.3. Argument Names H-3
H.2. HOW TO READ THE FAULT MESSAGE TABLES H-5
H.3. FAULT MESSAGE TABLES H-6

I. Pod-Related Information I-1

I.1. POD CALIBRATION AND OFFSETS I-1
I.2. POD INFORMATION FOR 9100A/9105A USERS I-2
I.3. SUMMARY OF 80186 POD SUPPLEMENTAL INFORMATION I-4

J. 9100A/9105A ERROR NUMBERS J-1

J.1. INTRODUCTION J-1
J.2. ERROR NUMBERS J-1

Section Title Page

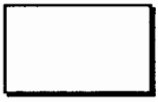
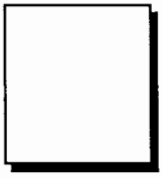
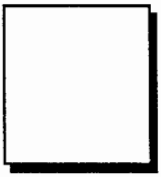
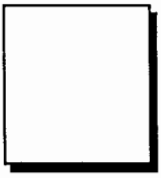
K. 9100 Series Software Error Report Form K-1

Index

Figures

Figure	Title	Page
3-1:	TL/1 Metasyntax Notation	3-3
3-2:	TL/1 Syntax Notation	3-5
I-1:	Calibration and Offset Example Waveforms	I-3
J-1:	Error Numbers	J-2

Where Am I?

Getting Started		A description of the parts of the 9100A/9105A, what they do, how to connect them, and how to power up.
Automated Operations Manual		How to run pre-programmed test or troubleshooting procedures.
Technical User's Manual		How to use the 9100A/9105A keypad to test and troubleshoot your Unit Under Test (UUT).
Applications Manual		How to design test or troubleshooting procedures for your Unit Under Test (UUT).
Programmer's Manual		How to use the programming station with the 9100A to create automated test or troubleshooting procedures.
TL/1 Reference Manual		A description of all TL/1 commands arranged in alphabetical order for quick reference.

Section 1
Overview

INTRODUCTION 1.1.

The *TL/1 Reference Manual* is one of a set of manuals that helps you to program your 9100A/9105A productively using the TL/1 programming language. You will find answers to questions about specific TL/1 statements and functions most quickly in this manual since the reference section is organized in alphabetical order.

However, if you are learning TL/1, or if you require more general information about programming concepts, you will probably want to first refer to the "Overview of TL/1" section of the *Programmer's Manual.*

ORGANIZATION OF THIS MANUAL 1.2.

The remaining sections of this manual are organized in the following order:

2. TL/1 Language Conventions -

   Name Conventions: How to assign names that denote variables, programs, files, directories, and devices.

   Data Types: Syntax, rules, and restrictions which affect numeric, floating point, and string variables.

   Arrays: Syntax, rules, and restrictions which apply to tables and arrays of numeric and string variables.

   Operators: A listing and description of symbols that create a new value from one or more existing values (operands).

   Order of Evaluation: A description of how the precedence of operators determines the order in which the operators' actions are performed.

   Conditional Expressions: How to construct conditional expressions, which control execution of block statements.

   TL/1 Statement Conventions: General conventions for TL/1 statements. This section describes the differences between simple statements and block statements.

   Functions: General conventions for TL/1 built-in functions.

3. TL/1 Alphabetical Reference - Each TL/1 function and statement in alphabetical order, provided with a summary of the command, programming examples, and references to additional explanations in the *Programmer's Manual.*

The Appendices, which follow the previous sections, contain the following information:

A. ASCII Codes - A table which provides ASCII character codes in hexadecimal and decimal notation, and their respective character representations.

B. Control Codes for Monitor and Operator's Display - Character codes which perform cursor movement and set video attributes on the monitor and the operator's display.

C. Operator's Keypad Mapping to TL/1 Input - Cross-listing of operator's keypad keys and the character codes which represent them.

D. Programmer's Keyboard Mapping to TL/1 Input - Cross-listing of non-standard programmer's keyboard keys and the non-standard character codes which represent them.

E. I/O Module Clip/Pin Mapping - Tables which indicate the correspondence between these sets of pins.

F. TL/1 Reserved Words - Alphabetized listing of TL/1 reserved words.

G. Handling Built-in Fault Messages in TL/1 Programs - Listings of built-in fault conditions and their corresponding arguments.

H. Raising Built-in Fault Messages in TL/1 Programs - Tables which show the relationship of fault messages to the arguments provided to fault condition handlers.

I. Pod-Related Information - Provides a summary of the pod-specific information available in the *Supplemental Pod Information for 9100A/9105A User's Manual.*

J. 9100A/9105A Error Codes - Provides a listing of possible errors the 9100A/9105A can encounter during operation.

K. 9100 Series Software Error Report Form.

Section 2
TL/1 Language Conventions

A number of conventions apply to TL/1 language statements. Variable and program names must follow a particular format, operators are evaluated in a specific order, data types and arrays are subject to certain restrictions, and all TL/1 statements must be entered in a consistent manner.

This section describes the following conventions:

- Name Conventions.
- Data Types.
- Arrays.
- Operators.
- Order of Evaluation of Operators.
- Conditional Expressions.
- Functions.
- TL/1 Statement Conventions.

NAME CONVENTIONS 2.1.

TL/1 rules require that you provide a name for each program, variable, file, and directory that you create. The TL/1 keywords which appear in the "TL/1 Alphabetical Reference" section of this manual follow the same convention.

This section describes the name conventions used in TL/1.

In TL/1, valid names meet the following requirements:

- A name must begin with a letter (A-Z, a-z,) or the character "@", or "_".
- A name can contain letters, numbers, and the characters "@" "$", and "_".
- A name must be distinguishable from reserved names (keywords).
- A name must have 255 or fewer characters. However, shorter name lengths are suggested as the debugger cannot process long variable names.

Names are case-sensitive; the names WXYZ, Wxyz, and wxyz denote different entities.

If a name is enclosed in single quote characters ('), it can be spelled the same as a keyword; the single quotes distinguish the name from the keyword. Single-quoted names can also contain spaces or punctuation marks. For example, the following variable names are valid:

```
'name containing spaces'
'name-containing-dashes'
'to'        (a name spelled like a keyword)
'test.101'
```

The single quotes are not part of the name (for example, foo and 'foo' are the same).

File and Directory Names 2.1.1.

Every file and directory has a name. A file or directory name must meet the following requirements:

- A name can have no more than 10 characters.
- A name consists only of letters, numbers, underscore characters "_", and periods ".".
- A name must begin with either a letter or a number.

File and directory names are not case-sensitive; "TEST1" is the same name as "test1". Two files or directories can have the same name if they have different types. For example, a program named TEST1 is distinct from a text document named TEST1. Two files of the same type can have the same name if they are in different directories. The program DEMO in the program library does not conflict with the program DEMO in a UUT directory. The names PARTLIB, PROGLIB, HELPLIB, and PODLIB can only be given to a part library, program library, help library, and pod library, respectively. You cannot name a program PODLIB, for example.

The names of directories that are limited to one per user disk or files that are limited to one per UUT directory are predetermined. These items and their names are:

Items limited to one per user disk

File and directory names appear in TL/1 as strings of characters surrounded by double-quote (") characters.

Directory	*Name*	*Type*
user disk (hard drive)	HDR	USERDISK
user disk (floppy drive 1)	DR1	USERDISK
user disk (floppy drive 2)*	DR2	USERDISK
part library	PARTLIB	LIBRARY
program library	PROGLIB	LIBRARY
pod library	PODLIB	LIBRARY
help library	HELPLIB	LIBRARY

Items limited to one per UUT directory

File	*Name*	*Type*
reference designator list	REFLIST	REF
node list	NODELIST	NODE

* On the 9105A only.

Program Names 2.1.2.

Because program names must match the name of the file that stores the program, they are subject to the following additional requirements:

- A program name contains from 1 to 10 characters.
- A program name consists of only letters, numbers, underscore characters (_), and periods (.). If a period is used, the program name must be enclosed in single quotes.
- A program name cannot be the same as the name of a built-in function.
- TL/1 requires that the program name be capitalized exactly the same in the program statement that defines the program and in every execute statement that invokes the program. The capitalization of letters is ignored when a program is looked up on the disk. Thus, it is not possible to define two program names that differ only in case (such as PROG1 and prog1).
- User programs must not use the names of Fluke-provided programs in the PODLIB.

Device Names 2.1.3.

TL/1 is a language designed for testing and troubleshooting. For this reason, it has built into it a convenient method for referring to the probe, to an I/O module, to a clip module (which fits into an I/O module), or to a component on a UUT (to which a clip module is attached).

When TL/1 refers to a pod, it uses the following name:

"/pod"

When TL/1 refers to the probe, it uses the following name:

"/probe"

The 9100A/9105A can have up to four I/O modules connected to it. The following are the valid I/O module names:

Name	*Description*
"/mod1"	I/O module 1
"/mod2"	I/O module 2
"/mod3"	I/O module 3
"/mod4"	I/O module 4

Each I/O module can have up to two clip modules connected to it. The clips are referred to as "A" or "B" depending on the side of the I/O module that they are connected to. The following are valid clip module names:

Name	*Description*
"/mod1A"	Clip module A of I/O module 1
"/mod1B"	Clip module B of I/O module 1
"/mod2A"	Clip module A of I/O module 2
"/mod2B"	Clip module B of I/O module 2
"/mod3A"	Clip module A of I/O module 3
"/mod3B"	Clip module B of I/O module 3
"/mod4A"	Clip module A of I/O module 4
"/mod4B"	Clip module B of I/O module 4

The 9100A/9105A can have an IEEE-488 interface installed to be used as either a talker/listener or as a controller. The IEEE-488 interface is opened in one of two ways:

"/ieee"	for the IEEE-488 interface
"/ieee/address list"	for one or more devices attached to the IEEE-488 interface (controller only)

An address list is a list of comma-separated IEEE-488 addresses. Each address is either a single radix 10 number, indicating the device address or a pair of numbers separated by a colon character, indicating the primary and secondary address of the device. For example:

"/ieee/1"	for the device at address 1
"/ieee/2,4:10"	for the group consisting of the device at address 2 and the device with primary address 4 and secondary address 10

The reference designator is another type of device name that is often used with TL/1 commands. This reference designator is a one to six character string that names a component on the UUT. Some typical examples are shown below:

U22 u17 SW3 R44 J5 J1A

Device List 2.1.4.

Many of the probe and I/O module commands allow a list of devices to be specified. The device list has device names separated by commas (no spaces are allowed). The following are valid device lists:

"/mod1,/mod2,/mod3"

"/probe,/mod1A,/mod2"

Reference Designator Names 2.1.5.

A reference designator is another type of device name that is often used with TL/1 commands. Reference designator names must meet the following requirements:

- A name consists only of letters, numbers, underscore characters "_", and periods ".".
- A name must begin with either a letter or a number.
- A name can have no more than six characters.
- Names are not case sensitive.

Some typical examples of reference designator names are shown below:

U22 u17 SW3 conn1 12a J5

Ref Pin Names 2.1.6.

A reference designator pin name (ref pin name) identifies a unique pin on the UUT. The name combines a reference designator name and a pin name, forming a name that is unique to a single pin on the UUT.

Ref pin names are composed of three parts: a reference designator name followed by a dash, followed by a pin name or pin number. The first part of the ref pin name (the reference designator name) must meet the requirements described in paragraph 2.1.4. above. The last portion of the ref pin name (the pin name or pin number) must meet the following requirements:

- Pin numbers can range from 1 to 255.
- Pin names consist only of letters, numbers, underscore characters "_", and periods ".".
- Pin names must begin with either a letter or a number.
- Pin names can have no more than eight characters.
- Pin names are not case sensitive.

Some typical examples of ref pin names are shown below:

U22-3
u17-40
conn1-b4

DATA TYPES 2.2.

A TL/1 variable can represent one of three data types: numeric, floating-point, or string. TL/1 also allows declaration of arrays of any of these data types. However, arrays aren't considered a separate data type.

Numeric 2.2.1.

The numeric type is the set of integers from +4,294,967,295 to 0. There are no negative numbers in TL/1. Each integer can represent a binary 32-bit data word as well as a numeric quantity. If a numeric constant is preceded by a "$" character, the digits are interpreted as hexadecimal (base 16) digits, and the allowed digits are 0-9 and A-F. Numeric values not preceded by a "$" character are interpreted as decimal, and the allowed digits are 0-9.

The following numeric values represent the number seventeen or the binary data word 10001 (left-most bit is most significant):

17	(decimal)
$11	(hexadecimal)

The following numeric values represent the number two hundred and fifty-five or the binary data word 11111111:

255	(decimal)
$FF	(hexadecimal)

When you represent numbers in hex, you must use only the digits 0-9 and the capital letters A-F. For example, $ABC represents a hexadecimal number, but AbC represents a variable name.

Floating-Point 2.2.2.

The floating-point type uses the IEEE standard for double-precision floating-point numbers. The full range of numbers is supported; however, representations of the non-numeric entities Infinity and NaN (not a number) are not implemented.

Floating constants can be represented in either fixed-point or scientific format. The following are examples of valid floating-point constants:

Fixed-point format:

```
1.2
0.1
-3.09
200.
```

Scientific format:

```
0.1E-03
-2E3
10.e+02
-99.9e23
```

Note that minus signs are allowed for floating-point constants. They are also allowed in front of any floating-point expression.

String 2.2.3.

A string is a list of zero to 255 characters enclosed in double-quote characters ("). All ASCII character codes are allowed. Non-printing characters are represented by a backslash character (\) followed by a two-digit hexadecimal number. A backslash followed by a double-quote character (\") represents the double-quote character. A backslash followed by a backslash (\\) represents the backslash character. Including a backslash sequence in a string will allow the printing of the following characters:

\HH	prints:	character represented by HH (where H represents any valid hex-code digit.)
\"	prints:	"
\\	prints:	\
\nl	prints:	new-line (a carriage return)

The following examples illustrate various strings and their interpretations:

String	*Interpretation*
""	the empty string
"hello world"	hello world
"\"hello world\""	"hello world"
"\09\09 hello"	<TAB><TAB> hello

A string is manipulated as one entity. The area allocated to a string variable changes with its value; a string variable does not need to have dimensions as does an array.

ARRAYS 2.3.

An array contains either numeric, floating-point, or string values associated with one variable name. Each element (value) stored in an array is identified uniquely by its subscript (number) or sequence of subscripts. For example,

```
X[3,5,1]
```

identifies a unique element in the array named X associated with the subscript sequence 3 - 5 - 1.

An array may have one or more subscripts represented by numeric expressions. For example,

```
a[1]
v[l+2]
table[i,j]
```

are valid names of array elements.

Arrays must be declared before being used so that a sufficiently large storage area will be reserved for array values; no implicit declaration is possible. Using a subscript outside the dimensions specified in the array declaration results in an error.

OPERATORS 2.4.

An operator is a symbol that creates a new value from one or more existing values (operands). Operand values may be the results of expressions, constants, or values of invocations. Each operator is marked with the word operator in the upper right corner of its description in Section 3.

Some operators have two forms; a symbol consisting of punctuation characters, and a short name. The two forms denote the same operator and may be used interchangeably.

The following sections summarize TL/1 operators.

Arithmetic Operators 2.4.1.

Arithmetic operators take numeric or floating-point operands and produce a result of the same type. Both operands must be of the same type.

Operator	*Description*	*Comments*
+	Produces an integer or floating-point sum.	
-	Produces an integer or floating-point difference.	
*	Produces an integer or floating-point product.	
/	Produces an integer or floating-point quotient.	Examples: 9/2 produces a quotient of 4. 3.0/2.0 produces a quotient of 1.5.
%	Produces an integer remainder.	Example: 9%2 produces a remainder of 1.
-	Multiplies the operand by -1.0 (floating-point only).	

Relational Operators 2.4.2.

Relational operators are used in conditional statements to compare magnitudes of quantities. These operators take two operands of the same type (two integer numbers, two floating-point numbers, or two strings) and produce a logical numeric result. If the condition is true, the result is a numeric 1, and if the condition is false, the result is a numeric 0.

Operator	*Description*	*Comments*
=	Equal to.	
<>	Not equal to.	
<	Less than.	
<=	Less than or equal to.	
>	Greater than.	
>=	Greater than or equal to.	
not	Negation of a logical expression.	Example: x = not y

Logical Operators 2.4.3.

Logical operators take numeric operands and produce numeric results or take string operands representing binary numbers and produce string results, which also represent binary numbers. The operands cannot be floating-point numbers.

Operator	*Description*	*Comments*
& and	Logical AND.	Example: 7 & 3 produces 3.
| or	Logical OR.	Example: 4 | 3 produces 7.
^ xor	Logical exclusive OR.	Example: 7 ^ 2 produces 5.
~ cpl	One's complement.	Each 1 in the operand is changed to a 0, and each 0 is changed to a 1

String Operators 2.4.4.

String operators are used to analyze or modify string operands.

Operator	*Description*	*Comments*
+	Appends a string expression to the end of another string expression.	For example, *a* + *b* is a string where string *b* is appended to the end of string *a*.
len	Counts the number of characters in a string operand.	See the *len* operator in the "TL/1 Alphabetical Reference" section of this manual.

String Functions 2.4.5.

String functions are used to analyze or extract substrings of string arguments.

Function	*Description*	*Comments*
mid	Copies a string of specified length and position from the string operand.	See the *mid* command in the "TL/1 Alphabetical Reference" section of this manual.
instr	Returns the position at which a sub-string is found in a string.	See the *instr* command in the "TL/1 Alphabetical Reference" section of this manual.
isval	Determines if a string is a suitable argument to *val*.	See the *isval* command in the "TL/1 Alphabetical Reference" section of this manual.
isflt	Determines if a string is a suitable argument to *fval*.	See the *isflt* command in the "TL/1 Alphabetical Reference" section of this manual.

token	Extracts a token from a string.	Used for scanning fields in strings. See the *token* command in the "TL/1 Alphabetical Reference" section of this manual.

Bit Shifting Operators 2.4.6.

The following operators shift the bits of a numeric operand either to the right or to the left. The bit locations vacated by shifted bits are filled with zeros.

Operator	*Description*	*Comments*
<< *shl*	Shifts the operand left by one or more bits.	See the *shl* command in the "TL/1 Alphabetical Reference" section of this manual.
>> *shr*	Shifts the operand right by one or more bits.	See the *shr* command in the "TL/1 Alphabetical Reference" section of this manual.

Bit Mask Operators 2.4.7.

These operators calculate a numeric value based on setting bits in a bit mask or they provide information about bit mask operands.

Operator	*Description*	*Comments*
bitmask	Calculates a number by setting all bits from bit 0 through the specified bit.	See the *bitmask* command in the "TL/1 Alphabetical Reference" section of this manual.
setbit	Calculates a number by setting a specified bit.	See the *setbit* command in the "TL/1 Alphabetical Reference" section of this manual.

lsb	Returns the position of the least-significant set bit in the operand.	See the *lsb* command in the "TL/1 Alphabetical Reference" section of this manual.
msb	Returns the position of the most-significant set bit in the operand.	See the *msb* command in the "TL/1 Alphabetical Reference" section of this manual.

ORDER OF EVALUATION OF OPERATORS 2.5.

The precedence of operators determines the order in which the operators' actions are performed. Operators with higher precedence are considered before operators with lower precedence. From highest to lowest, the precedence of operators in TL/1 is:

1. cpl, setbit, msb, lsb, bitmask, len, not, - (floating-point only)
2. *, /, %
3. +, -
4. shl, shr
5. =, <>, <, >, <=, >=
6. and
7. or, xor

According to this order, the expression

```
a + b * msb c
```

is equivalent to

```
(a + (b * (msb c))).
```

The value of "msb c" is evaluated first. Then the result is multiplied by the value of b and this result is added to the value of a.

Parentheses modify the order in which expressions are evaluated, overriding the order of precedence. For example, in the expression

(a + b) * (c - d)

"a + b" and "c - d" are evaluated before the multiplication is performed.

If, after parentheses and the order of precedence are considered in the evaluation of expressions, several expressions of the same precedence exist, they are evaluated left-to-right. For example, in the expression:

a + b * c - d

"b * c" is evaluated first. The result is added to the value of a, then the value of d is subtracted from that sum.

CONDITIONAL EXPRESSIONS 2.6.

The *if* statement, *if* block, *loop* block, and *for* block execute statements under control of a condition. This condition is a logical expression that evaluates to either true (non-zero) or false (zero). The examples below show that the conditional expression can compare numeric expressions, floating-point, or string expressions.

Example 1:

```
if a = $2E then print "SUCCESS!"
```

Example 2:

```
if ans = "yes" then
     .      ! Any statements here are executed
     .      ! only if the string variable is
     .      ! equal to yes
end if
```

Example 3:

```
if b < 3.52 then
      .      ! Any statements here are executed
      .      ! only if the floating-point
      .      ! variable is less  than 3.52
end if
```

Example 4:

```
if f  < > 0.0 then
      .      ! Any statements here are executed
      .      ! only if the floating-point
      .      ! variable is not equal to the
      .      ! floating-point value 0.0
end if
```

You can use both logical operators and relational operators in a single conditional expression as shown below. However, you should be careful to check the order of evaluation of such an expression to be sure that you have written it to do what you want.

Example 5:

```
            ! loop until either x = $2FE3 or
            ! until y = 100
loop until x = $2FE3 or y = 100
      .
      .
      .
end loop
```

Example 6:

```
            ! this doesn't AND b with c
if a < b and c < d then
```

You should be careful to use the result of a logical comparison to set flag variables as shown in the following example.

```
flag1 = (a < b)   ! The flag bit is located in
flag2 = (c < d)   ! bit 0 of both operands
if flag1 and flag2 then x = 1
```

If different bits are used for flag values, incorrect results can occur when doing logical operations. The example below shows what you should not do.

```
if a < b then flag1 = 1   ! Bit 0 is affected.
if c < d then flag2 = 2   1 Bit 1 is affected.
                          ! Bit 0 of flag1 is
                          ! not ANDed with
                          ! bit 1 of flag2.
if flag1 and flag2 then x = 1
```

Two of the most important operators used with TL/1 are the *passes* condition and the *fails* operators. Each of these is described in detail in the "TL/1 Alphabetical Reference" section of this manual. The passes and fails operators test whether a test function has completed without reporting any faults. An example of each condition is shown below.

Example 1:

```
if testbus fails then   ! Test the termination
     y = 0              ! status after using
else                    ! the testbus command,
     y = 1              ! and set flag y to
end if                  ! zero if the test
                        ! fails and to one if
                        ! the test passes.
```

Example 2:

```
if testbus passes then
     print "Wonderful"
else
     print "It's troubleshooting time"
end if
```

FUNCTIONS 2.7.

TL/1 provides over 130 built in functions to perform basic operations on numbers and strings, to communicate with displays, keyboards, and ports. Basic operations are also performed to control the pod, I/O modules, and probe, to collect measurements, and to test UUT circuits like busses, RAM, and ROM.

Although each function has a different name and list of arguments, they are all invoked using the same syntax. See the *execute* statement for more information on how functions are invoked. All functions are marked with the word function in the upper right corner of their description in Section 3.

A function has zero or more arguments. Each argument is an expression resulting in a number or string. Refer to "How Programs and Functions are Invoked" in Section 3 of the *9100 Series Programmer's Manual* for more information on calling and returning data from functions.

Special Functions 2.7.1.

Special functions are functions with special restrictions on the way they may be invoked. Special functions must be invoked using the keyword notation, where the function name is followed by a comma-separated list of the argument name followed by an expression giving the value of the argument.

Pod Functions 2.7.2.

Pod functions control the microprocessor emulation pod to read and write UUT data and perform tests on UUT circuits including busses, RAM, and ROM.

I/O Module and Probe Functions 2.7.3.

These functions control the parallel I/O modules and the probe to read and write bit streams to the UUT and collect signatures.

Type Conversion Functions 2.7.4.

The following functions perform conversions between various data types.

Function	*Description*	*Comments*
ascii	Converts a single-character string into its ASCII code number.	See the *ascii* function in the "TL/1 Alphabetical Reference" section of this manual.
chr	Converts a number from 0 through FF (hexadecimal) or from 0 through 255 (decimal) into a single ASCII character.	See the *chr* function in the "TL/1 Alphabetical Reference" section of this manual.
str	Converts a number into its string representation.	See the *str* function in the "TL/1 Alphabetical Reference" section of this manual.
val	Converts a string representing a number into the appropriate numeric value.	See the *val* function in the "TL/1 Alphabetical Reference" section of this manual.
cflt	Converts a numeric value to a floating-point value.	See the *cflt* function in the "TL/1 Alphabetical Reference" section of this manual.
cnum	Converts a floating-point value to a numeric value.	See the *cnum* function in the "TL/1 Alphabetical Reference" section of this manual.

fval	Converts a string representing a number into the appropriate floating-point value.	See the *fval* function in the "TL/1 Alphabetical Reference" section of this manual.
fstr	Converts a floating-point value into its default string representation.	See the *fstr* function in the "TL/1 Alphabetical Reference" section of this manual.

TL/1 STATEMENT CONVENTIONS 2.8.

Statements control and modify the order of execution in TL/1. The parts of each statement are separated by reserved keywords, so each statement has a unique syntax.

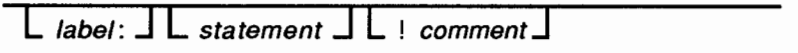
TL/1 defines two types of statements: simple statements, and block statements. A simple statement performs a single action. Block statements delimit the beginning and end of blocks and control the execution of the statements they enclose.

Block statements always come in pairs; a block beginning statement that begins with a name (if, loop, program), and a block ending statement that has the form 'end name' (end if, end loop, end program). Statement lines between the block beginning and ending statement are controlled by the block.

Simple statements and statements that delimit statement blocks are marked with the word(s) statement or statement block in the upper right corner of their description in Section 3.

In TL/1, each line contains either a single block statement or one or more simple statements separated by backslashes (\). The statements are executed in order from left to right. The line may begin with an optional label, followed by a colon (:). The statement may be followed by an optional comment, with an exclamation point (!) separating the statement and comment.

Syntax Diagram:



Arguments:

label	The name of the labeled line.
statement	Any valid program statement.
comment	The text of a comment.

The label, statement, and comment are all optional; a blank line may be labeled, and lines do not need to be labeled. Label names follow the same conventions used for naming variables. See the section, "Name Conventions," earlier in this section for additional information. The number of labels available to the programmer is limited only by available memory, however the availability of powerful block-structured commands in TL/1 eliminates the need for most labels.

A label identifies a line for subsequent use. A labeled line begins with a label followed by a colon (:). A label is unique for the entire program. No more than one line can be labeled with a particular name.

A comment begins with an exclamation mark (!) and continues until the end of the line. TL/1 ignores comments. Comments make the program easy to read and understand, and should be used to point out an action that is implied or not obvious.

Example 1:

```
                                ! A line labeled jail
jail: if read addr a <> 0 then fault bad_value
```

Example 2:

```
                                ! A simple statement.
write addr $1000, data $21
```

Example 3:

```
a = 1 \ b = 2 \ c = 3     ! A simple statement
                          ! list.
```

Example 4:

```
i = 1
loop while i <= 10          ! A block beginning
                            ! statement.
     write addr + i, data $A0 + i
     i = i + 1              ! The loop block will
end loop                    ! be executed ten
                            ! times.
```

Section 3
TL/1 Alphabetical Reference

Throughout this alphabetical reference section, the syntax of various TL/1 statements is described in both textual (meta-syntactic) and diagrammatic form.

The metasyntax notation follows these rules:

- Words that are not enclosed in angle brackets (<>) are *required* words and are to be used literally.
- Words that represent names and values you supply are delimited by angle brackets (< >).
- A word or group of symbols separated by one or more solid vertical bars (|) indicates that one, and only one, of the group should be chosen.
- A word or group of symbols enclosed in square brackets [ ] are optional. If the first character in the group is a comma (,), this comma is included as a delimiter only when another optional group precedes it.

NOTE

In the declare command, square brackets are used literally to define array dimensions.

- A word or group of symbols enclosed in braces { } can be repeated any number of times, separated by commas (,).
- <device list> refers to one or more device names, separated by commas (,).
- <expression list> refers to one or more expressions, separated by commas (,).
- <variable list> refers to one or more variables, separated by commas (,).
- <statement list> refers to one or more TL/1 statements, separated by backslashes (\).

Refer to Figure 3-1 for an example of metasyntax notation.

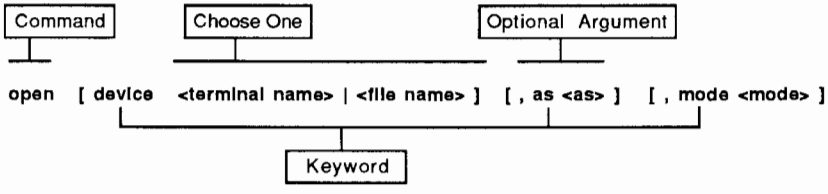
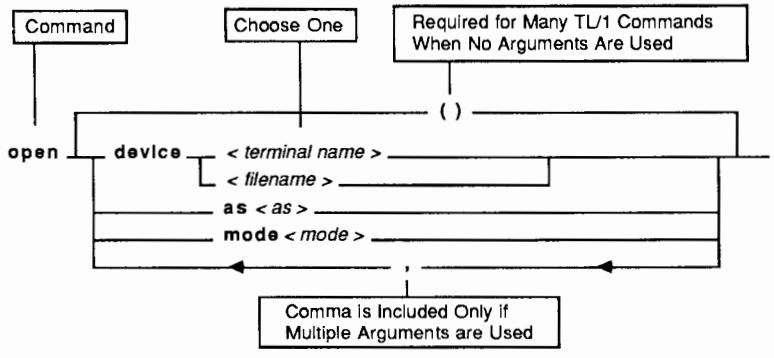


Figure 3-1: TL/1 Metasyntax Notation

The syntax diagrams follow these rules:

- Keywords - Words to be used literally appear in boldface.
- Arguments - Words that represent names and values you supply appear in italics and are delimited by angle brackets (<>). For example, the word "filename" in a syntax diagram represents the name of a file that you specify.
- Solid lines - Solid lines connect keywords or symbols, and programmer-supplied values. These lines represent the syntax path, read from left to right. Vertical paths represent options; horizontal paths with arrowheads represent optional repeat loops.
- Ellipses (...) are used to connect syntax diagrams which, due to length, span multiple lines. The ellipses indicate "continue to type."
- <device list> refers to one or more device names, separated by commas (,).
- <expression list> refers to one or more expressions, separated by commas (,).
- <variable list> refers to one or more variables, separated by commas (,).
- <statement list> refers to one or more TL/1 statements, separated by backslashes (\).

Refer to Figure 3-2 for a description of the syntax diagrams.



Read syntax paths from left to right, unless an arrow-head indicates a loop. Vertical paths represent options.

Figure 3-2: TL/1 Syntax Notation

On the first page of each command in the upper right corner below the command name, is the syntactic category for that command. The categories (function, special function, statement, statement block, and operator) are described in Section 2 of this manual.

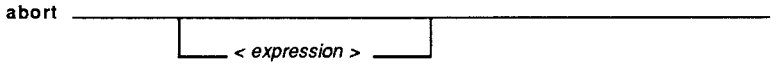
Each command within the alphabetical reference section contains information under some or all of the following headings:

- Syntax - a metasyntactic (textual) description of the syntax for a particular TL/1 function or command.
- Syntax Diagram - a diagram which illustrates the syntax for a particular TL/1 function or command.
- Description - a description of the TL/1 function or command.
- Argument(s) - a description of the arguments which the programmer provides to the TL/1 function. Some arguments are optional, and have a default value if another value is not specified.
- Returns - the value returned by a function.
- Example - one or more TL/1 programming examples.
- Remarks - additional information about a command or function.
- Related Commands - other TL/1 commands which pertain to the command discussed. Refer to these commands within this section for related information.
- For More Information - a reference to other materials that contain additional information about this command. References are to the "Overview of TL/1" section of the *Programmer's Manual,* appendices within this manual, and Fluke pod manuals.

Syntax:

```
abort [<expression>]
```

Syntax Diagram:



Description:

Performs a multilevel return statement.

Example:

```
program control

 function test
  handle         ! any fault raised within function
                 ! test is handled here
   print "aborting test" ! represents any other
                        ! handler actions
   fault         ! preserve 'fails' termination
                 ! status
   abort         ! causes function test to terminate
  end handle

  if (some_condition) then ! if some faulty
                           ! behavior is detected
   fault 'test fault'
  end if
 end function

 loop
  if test() passes then    ! often this program has
   print "Pass"            ! no knowledge that the
                           ! called function aborts.
  else                     ! only interested in
                           ! passes/fails information.
   print "Fail"
  end if
 end loop
end program
```

In the example, program 'control' calls function 'test' repeatedly. This is typical of a production test setup that calls the test for each UUT. If the test fails, the procedure abandons testing of that board and prints a failure message. The default handler is invoked if any faults occur, and the fault statement preserves the (failure) termination status of the test. The abort statement returns control to the caller of function 'test', which in this case is program 'control'. In actual examples, there are likely to be many layers of function calls inside function 'test'.

Remarks:

The *abort* command may be used in two ways. First *abort* can terminate execution of the entire test program when the decision to terminate is made in a deeply nested subprogram. The command can also terminate execution of a sub-test within a program when a fault condition is handled. The second use permits simple go/no-go tests to be written, and the command makes decisions on continuing tests once certain faults are detected.

When the *abort* command is executed in the main-line code, TL/1 terminates execution. If the optional expression is specified, the value of that expression is returned, as if by the top-level program.

When the *abort* command is executed within a fault condition handler, all invoked programs and functions are terminated, up to the program or function that activated the handler. (This is the program or function that contains the handler definition block.)

If the optional expression is specified, the value of that expression is returned, as if by the program or function that activated the handler.

When the *abort* command is executed within a fault condition exerciser, the exerciser terminates as if a *return* statement had been executed in the top-level exerciser block.

If a program or function invocation returns a value, a value must be supplied to all places the invocation returns, including all *abort* commands that cause the invocation to return. The type of value returned must be the same in all *return* and *abort* commands.

The *abort* command does not affect the termination status (for example, whether the program passes or fails). If the test has failed, a *fault* command should precede the abort.

Related Commands:

fault, function, handle, program, refault, return

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

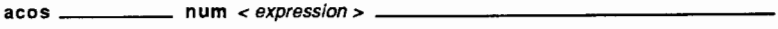
acos function

IEEE-488

Syntax:

```
acos num <expression>

acos (<expression>)
```

Syntax Diagram:



Description:

Returns the inverse cosine function (in radians) of the floating-point argument value.

Arguments:

expression	The argument (cosine) value, which must be in the range: $-1.0 \leq num \leq 1.0$.

Returns:

A floating point number in radians.

Examples:

```
theta = acos (0.0)
theta = acos num f
```

Remarks:

An error is generated if the argument value is outside the allowable range.

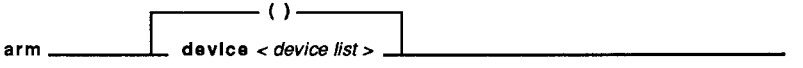
Related Commands:

cos

Syntax:

```
arm device <device list>

arm (<device list>)

arm ()
```

Syntax Diagram:



Description:

Specifies the beginning of an *arm . . . readout* block. Arms the response gathering hardware of the specified I/O modules or probe to start capturing signatures, levels, and count information. If the counter mode is "freq", a frequency measurement occurs at the *arm* statement. The actual point at which response gathering starts depends on UUT signals and on the settings of start, stop, clock, and enable when the sync mode is set to "ext" for the probe or an I/O module.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")

Example 1:

```
mod = clip ref "u3", pins 40

arm device mod
     rampdata addr 0, data 0, mask $FF
     rampdata addr 0, data 0, mask $FF00
readout device mod
```

Example 2:

```
arm device "/mod1,/mod2"
     .
     .
readout device "/mod1,/mod2"
```

Example 3:

```
arm ()              ! The probe is the default device
     .
     .
readout ()          ! The probe is the default device
```

Example 4:

```
modlist = "/mod1,/mod2,/mod3,/mod4"

arm device modlist
     .
     .
readout device modlist
```

Example 5:

```
devlist = "/mod1"       ! name the device
                        ! set threshold levels
threshold device devlist, level "ttl"
                        ! set counter mode
counter device devlist, mode "transition"
                        ! sync device to external
sync device devlist, mode "ext"
edge device devlist, start "+", stop "+", clock "+"
connect device devlist, start "U3-1", stop "U7-8",
 clock "U4-8"
                        ! ignore enable line
enable device devlist, mode "always"

arm device devlist      ! start the response capture
                        ! apply the stimulus
     rampdata addr $8FFF, data 0, mask $FF
```

(example is continued on the next page)

```
                              ! check that response
                              ! gathering is complete
        status = checkstatus device devlist

        if status <> $F then
             if (status and 1) = 0 then
                  reason = "no valid clock seen"
             else if (status and 2) = 0 then
                  reason = "no valid enable seen"
             else if (status and 4) = 0 then
                  reason = "no valid start seen"
             else if (status and 8) = 0 then
                  reason = "no valid stop seen"
             end if
             fault_flag = 1
        else
             fault_flag = 0
        end if
                              ! terminate the response
                              ! capture
readout device devlist
```

Remarks:

The *arm . . . readout* block begins with an arm command and ends with a readout command. These commands control the I/O module or probe by activating and deactivating the response gathering hardware.

The arm command clears the signature, level, and count registers and starts the capture of new readings. The *readout* command stops the capture of data and makes the results available. After *readout* is executed, the captured response data is accessible through the *sig, count,* and *level* functions until another *arm . . . readout* block for the same device is executed.

When you use external sync, you use *checkstatus* to determine when to exit the *arm . . . readout* block. You, the programmer, are responsible for providing a means of exiting the block when *checkstatus* indicates the response capture is complete and successful. See the *checkstatus* command for more information.

Incomplete response data may be caused by any of the following:

- An external stop line was specified, but the stop signal was not active before *readout* was executed.
- The programmable stop counter was activated (through the *edge* and *stopcount* commands) but the specified number of clock pulses had not been counted before *readout* was executed.
- An external start line was specified, but the start signal was not active before *readout* was executed.
- An external enable line was specified, but the enable signal was not active before *readout* was executed.
- An external clock line was specified, but a clock signal was not received before *readout* was executed.

Related Commands:

checkstatus, count, counter, edge, enable, level, readout, setoffset, sig, stopcount, strobeclock, sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
ascii char <character>

ascii (<character>)
```

Syntax Diagram:

Description:

Finds the ASCII code number that represents the single character in the operand string.

Arguments:

character	Any character; a string which consists of a single character.

Returns:

The numeric value of the ASCII code number that represents the single character in the operand string.

Example:

```
x = ascii ("A")     ! the variable x is set to
                    ! the hex value 41 (decimal 65)
```

Remarks:

An error is generated if the argument string is not exactly one character long.

Related Commands:

chr

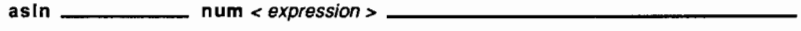
asin function

IEEE-488

Syntax:

```
asin num <expression>

asin (<expression>)
```

Syntax Diagram:



Description:

Returns the inverse sine function (in radians) of the floating-point argument value.

Arguments:

expression	The argument (sine) value, which must be in the range: $-1.0 \leq num \leq 1.0$

Returns:

A floating point number in radians.

Examples:

```
theta = asin (0.0)
theta = asin num f
```

Remarks:

An error is generated if the argument value is outside the allowable range.

Related Commands:

sin

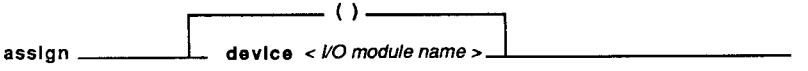
assign function

IEEE-488

Syntax:

```
assign device <I/O module name>

assign (<I/O module name>)

assign ()
```

Syntax Diagram:



Description:

Allocates a specific I/O module to the programmer, resets the internal variables that store connection data for that module, and returns an identifier associated with the I/O module.

Arguments:

I/O module name	The I/O module name ("/mod1", "/mod2", "/mod3", or "/mod4"). (Default = "/mod1")

Returns:

An identifier string for the I/O module.

Example 1:

```
mod2 = assign device "/mod2"
```

Example 2:

```
iomod = assign device "/mod4"
```

Example 3:

```
stimulus = assign ("/mod3")
```

Remarks:

You use *assign* instead of *clip* in order to display your own messages (rather than those displayed by *clip*), or to control module selection (the *clip* function lets the user select the module). This function returns a string which identifies the selected module.

Related Commands:

clip

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
<variable> = <expression>
```

Syntax Diagram:

Description:

Assigns a value to a variable. The variable on the left of the equal sign takes the value of the expression on the right side. The data type of the expression must be the same as the data type of the variable. A previously undeclared variable is declared implicitly with the assignment statement to be a local variable of the same type as the expression used.

Arguments:

variable	Assignment variable name.
expression	Any valid expression.

Examples:

```
a = $15          ! variable a set to hex 15

y = a + 1        ! variable y set to value of
                 ! variable a plus 1

z = 10           ! variable z set to decimal 10
```

```
s = "Hello"     ! variable s is set to the
                ! string value "Hello".

f = 3.2         ! variable f is set to the
                ! floating-point value 3.2
```

Remarks:

If the data type of the expression does not match the data type of the variable, an error is generated.

Related Commands:

declare

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

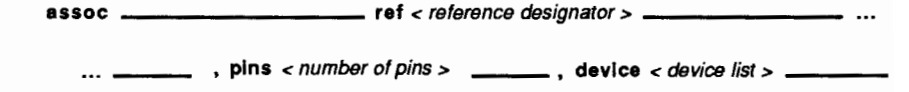
assoc function

IEEE-488

Syntax:

```
assoc ref <reference designator>, pins <number of
     pins>, device <device list>

assoc (<reference designator>, <number of pins>,
     <device list>)
```

Syntax Diagram:



Description:

Associates an I/O module or clip module with a UUT component. Unlike the *clip* command, the *assoc* command requires that the specified I/O module or clip module already be connected to the component.

Arguments:

reference designator	Reference designator indicating the name of the component to which the I/O module is already clipped.
number of pins	Number of pins associated with this reference designator. Valid range is any even number between 2 and 254.

device list	I/O module name, clip module name, or combinations of these. An I/O module name refers to a device of 40 pins.

Example 1:

```
devicelist = "/mod1,/mod2B,/mod3A,/mod4B"
assoc ref "U1", pins 80, device devicelist
arm device devicelist
        .
        .
        .
readout device devicelist
crc = sig device "U1", pin 1
```

Example 2:

```
assoc ref "U23", pins 14, device "/mod4A"
```

Remarks:

When using fixturing, the placement of the I/O module clips is preset so it is unnecessary and undesirable to press the ready button on each of the clips (as required by the *clip* command). In this case, the *assoc* command should be used.

The *assoc* command is functionally equivalent to the *clip* command except that the device list is set in the TL/1 program by the programmer rather than being determined by the I/O module button that is pressed.

NOTE

The operator must position the clip(s) so that pin 1 of the clip(s) is connected to pin 1 of the component before the execution of the assoc command.

You should use the *clip* command instead of the *assoc* command if you want to perform the following operations:

- Suspend operations until a ready button is pressed on the I/O module.
- Display the standard clip messages.

Related Commands:

assign, clip

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

atan function

IEEE-488

Syntax:

```
atan num <expression>

atan (<expression>)
```

Syntax Diagram:

Description:

Returns the inverse tangent function (in radians) of the floating-point argument value.

Arguments:

expression	The argument (tangent) value.

Returns:

A floating-point value in radians.

Examples:

```
theta = atan (0.0)
theta = atan num f
```

Remarks:

Argument values of infinity are not supported, since the implementation of floating-point does not support IEEE infinity.

Related Commands:

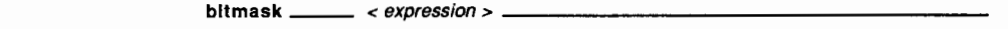
tan

bitmask
operator

Syntax:

```
bitmask <expression>
```

Syntax Diagram:



Description:

Generates a bitmask in which all the bits from bit 0 (the least-significant bit) through the bit specified by the expression are set.

Arguments:

expression	An expression that yields a number from 0 through 31 (decimal), or a number from 1 through 1F (hexadecimal).

Returns:

A bitmask with all the bits set from bit 0 through the bit specified by the expression.

Examples:

```
x = bitmask 3        ! the variable x is set to F
                     ! (bits 0 through 3 are set)
```

```
x = $F and bitmask 2 ! the variable x is set to 7
                     ! (bits 0 through 2 are set)
```

Remarks:

An error is generated if the argument is greater than 31 (decimal).

Related Commands:

setbit

Syntax:

```
cflt num <expression>

cflt (<expression>)
```

Syntax Diagram:

Description:

Converts the numeric argument to the equivalent floating-point value. The argument can be any valid numeric value.

Arguments:

expression	The numeric argument value.

Returns:

A floating-point number equivalent of the value of the argument.

Examples:

```
f = cflt ($32)
f = cflt num 109
```

Related Commands:

cnum

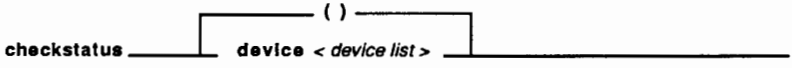
checkstatus function

IEEE-488

Syntax:

```
checkstatus device <device list>

checkstatus (<device list>)

checkstatus ()
```

Syntax Diagram:



Description:

Checks whether response gathering is complete within an *arm . . . readout* block. The return value of *checkstatus* indicates the result of this check.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")

Returns:

A code indicating whether response gathering is complete (see Remarks).

Example:

```
          ! This example uses checkstatus to check
          ! for an incomplete responses error.

sync device "/mod1", mode "ext"

arm device "/mod1"
     cnt = 0
     loop while ((checkstatus device "/mod1") <> $F
      and cnt < 100)
               ! Remember precedence of <>
               ! operator
          cnt = cnt + 1
     end loop
     if cnt = 100 then
               ! response gathering incomplete
          n = checkstatus device "/mod1"
     else      ! response gathering is complete
          n = $FF
     end if
readout device "/mod1"
```

Remarks:

The *checkstatus* function usually appears on the right side of an assignment statement (=), or within the context of a more complex expression.

The *checkstatus* function has real meaning only for external sync. In this case, the returned value is comprised of 32 bits with the 4 least-significant bits containing the status of response gathering. (Bit 0 is the least-significant bit.)

Bit	*Signal*	*Value*
4-31	*none*	(always 0)
3	Stop received	(1 = yes, 0 = no)
2	Start received	(1 = yes, 0 = no)
1	Enable received	(1 = yes, 0 = no)
0	Data clocked	(1 = yes, 0 = no)

For the other sync modes, (internal, pod, and freerun), the returned value will always be the number F (hexadecimal).

Related Commands:

arm, readout, sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

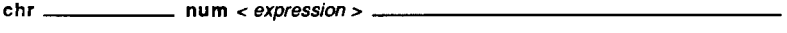
chr function

IEEE-488

Syntax:

```
chr num <expression>

chr (<expression>)
```

Syntax Diagram:



Description:

Returns a string consisting of the single ASCII character that corresponds to the numeric operand.

Arguments:

expression	A numeric expression that yields a value from 0 through FF (hexadecimal) or 0 through 255 (decimal).

Returns:

A string consisting of the single ASCII character that corresponds to the numeric operand.

Examples:

```
x = chr(7)     ! x is set to Ctrl-G (bell)

x = chr($23)   ! x is set to ASCII character "#"

x = chr(35)    ! x is set to ASCII character "#"
```

Remarks:

An error is generated if the numeric expression is greater than decimal 255.

Related Commands:

ascii

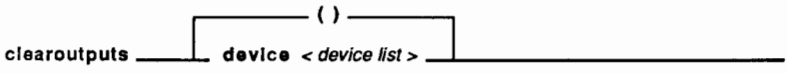
clearoutputs function

IEEE-488

Syntax:

```
clearoutputs device <device list>

clearoutputs (<device list>)

clearoutputs ()
```

Syntax Diagram:



Description:

Turns off the I/O module output drivers.

Arguments:

device list	I/O module name, clip module name, or reference designator. (Default = "/mod1")

Example:

```
iomod = clip ref "u21",pins 40
      .
      .
storepatt device "u21",pin 1, patt "01010101"
      .
writepatt device "u21", mode "latch"
      .
      .
clearoutputs device "u21"
```

Remarks:

Using clearoutputs is functionally equivalent to using both *storepatt* and *writepatt* to 3-state all pins. Devices can be cleared through specification of an I/O module, a clip on an I/O module, or a reference designator.

Related Commands:

clearpatt, storepatt, writepatt

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

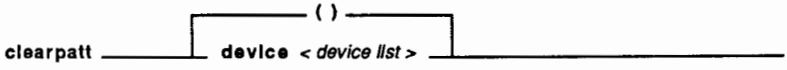
clearpatt function

IEEE-488

Syntax:

```
clearpatt device <device list>

clearpatt (<device list>)

clearpatt ()
```

Syntax Diagram:



Description:

Discards the output patterns previously saved with the *storepatt* command.

Arguments:

device list	I/O module name, clip module name, reference designator. (Default = "/mod1")

Example:

```
clearpatt device "/mod4"
```

Related Commands:

clearoutputs, storepatt, writepatt

For More Information:

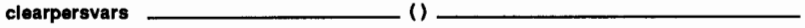
- The "Overview of TL/1" section of the *Programmer's Manual.*

clearpersvars
function

Syntax:

```
clearpersvars ()
```

Syntax Diagram:



Description:

Clears the values of currently active persistent variables. The term "currently active" means the subset of the set of persistent variables that is known so far by the TL/1 program executing the *clearpersvars* command. The value received as a result depends on the variable type:

numeric	0
floating	0.0
string	""

Example:

For each of the following example programs, assume that the persistent variable set initially contains:

```
Name    Type        Value

pv1     numeric     3
pv2     string      "foo"
pv3     string      "bar"
```

After executing the following program:

```
program itis
     declare persistent numeric pv1
     declare persistent string pv2
     clearpersvars ()
end program
```

the persistent variable set contains:

```
Name     Type          Value

pv1      numeric       0
pv2      string        ""
pv3      string        "bar"
```

After executing the following program:

```
program mobility
     declare persistent string pv3
     function foober
        declare persistent string pv2
     end function
     clearpersvars ()
end program
```

the persistent variable set contains:

```
Name     Type          Value

pv1      numeric       3
pv2      string        "foo"
pv3      string        ""
```

Remarks:

The *clearpersvars* command only affects the values associated with variables in the persistent variable set, and not whether a variable is a member of the set.

Note that it is stated that the *clearpersvars* command only affects the set of persistent variables known so far by the currently executing TL/1 program. If the program contains a declaration for a persistent variable, but has not processed it, the variable will not be cleared.

Related Commands:

resetpersvars

clip function

IEEE-488

Syntax:

```
clip ref <reference designator>, pins <number of
     pins>

clip (<reference designator>, <number of pins>)
```

Syntax Diagram:

Description:

Prompts the user with a message to clip over the specified component and to press the button on the clip module when ready. After the button is pressed, the *clip* command returns a string identifying the I/O module and the clip module that were selected.

Arguments:

reference designator	Reference designator indicating the name of the component to which the I/O module should be clipped.
number of pins	Number of pins associated with this reference designator. Valid range is any even number between 2 and decimal 254.

Returns:

A string that identifies the selected module(s) and clip module(s).

Example 1:

```
iomod = clip ref "u1", pins 24
```

Example 2:

```
iomod = clip ("u1",24)
```

Remarks:

The *clip* command prompts the operator to select a clip module, clip it to a specific component and press the ready button on the clip module. The *clip* command determines which clip module the operator has selected by returning a string which identifies the module name and button (A or B) pressed. You use this name as an argument in TL/1 function calls to the I/O module.

This command allows you to specify the component to which an I/O module should be clipped. The 9100A/9105A software remembers that the I/O module is clipped to this component.

NOTE

The operator must position the clip so that pin 1 of the clip is connected to pin 1 of the component.

You should use the *assign* command instead of the *clip* command if you do not want to perform any of the following actions:

- Prompt the operator.
- Suspend operations until a ready button is pressed on the I/O module.
- Display the standard clip messages.

Related Commands:

assign

For More Information:

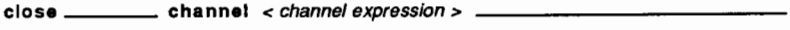
- The "Overview of TL/1" section of the *Programmer's Manual.*

close
function

Syntax:

```
close channel <channel expression>

close (<channel expression>)
```

Syntax Diagram:



Description:

Closes the I/O channel whose channel number matches the value of an expression.

Arguments:

channel expression	Numeric expression which evaluates to a valid channel number.

Example 1:

```
n = open device "/term1", as "output"
      .
      .
close channel n
```

Example 2:

```
kp = open device "file1", mode "unbuffered"
      .
      .
close (kp)
```

Remarks:

In large programs, it is important to close each file channel once the program has executed. The total number of channels that may be open at one time is limited.

Related Commands:

input, input using, open, poll, print, print using

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

cnum function

IEEE-488

Syntax:

```
cnum num <expression>

cnum (<expression>)
```

Syntax Diagram:

Description:

Converts the floating-point argument to the nearest numeric value (by rounding).

Arguments:

expression	The floating-point argument value, which must be within the range $0.0 \leq \text{expression} \leq 2^{32}-1$.

Returns:

A numeric value.

Examples:

```
n = cnum (32.0)          ! n is set to 32
n = cnum (100.6)         ! n is set to 101
n = cnum num 10E2        ! n is set to 1000
n = cnum num -33.0       ! this results in an error
```

Related Commands:

cflt

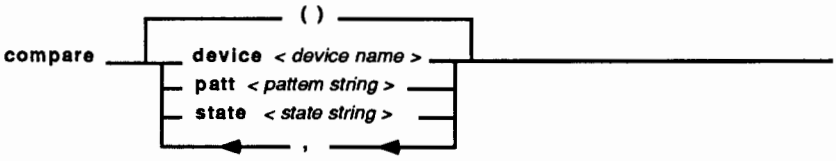
compare function

IEEE-488

Syntax:

```
compare [device <device name>] [, patt <pattern
     string>] [, state <state string>]

compare (<device name>, <pattern string>, <state
     string>)

compare ()
```

Syntax Diagram:



Description:

Monitors pins on an I/O module for the occurrence of the specified pattern and generates a "iomod_dce" fault condition when a match occurs.

Arguments:

device name	I/O module name, or clip module name. (Default = "/mod1")
pattern string	String expression for the comparison pattern. The left-most character in the pattern string corresponds to pin 1. (Default = "1")
state string	"enable" or "disable" comparison. (Default = "enable")

Example 1:

```
compare device "/mod2", patt "1011100XXXX11X",
 state "enable"
```

Example 2:

```
compare ("/mod1A", "1111111111111111", "disable")
```

Example 3:

```
program test9
     .
     .
     handle iomod_dce        ! This handler is called
                             ! whenever the bit
                             ! pattern specified in a
         print "successful DCE"! compare command
                             ! matches the pattern on
                             ! pins being monitored
                             ! by an I/O module
     end handle
     .
     .
     compare device "/mod1", patt "10111XXX001"
     .
     .
end program
```

Remarks:

A fault condition is generated when the specified bit pattern is detected. The *compare* command might be used to generate a fault condition when a particular UUT address is accessed or when particular data is on the data bus.

The pattern is specified as a single string that may contain the following characters:

1: high
0: low (or invalid levels)
x or X: don't care

Note that both low levels and invalid levels are considered to be low when making comparisons.

The characters in the string are mapped onto pins with the left-most character corresponding to pin 1. For example, if you want to detect when pins 1, 2, and 3 are high and when pins 12, 13, and 14 are low (and you do not care about the state of pins 4-11), you specify the pattern "111XXXXXXXX000".

You can specify the pattern in terms of clip pins or I/O module pins. If the device has more pins than the pattern, the excess pins are ignored.

The *compare* command may be used to compare a pattern string with up to 40 pins if they are all on the same I/O module and depending on what clip module is used.

The data compare equal (DCE) condition is detected as soon as it occurs, but the resulting DCE fault condition is raised only when an I/O module or probe function is executed or when a pod-access function is executed.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

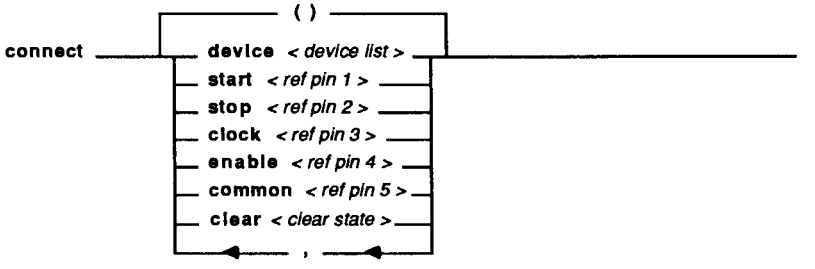
connect function

IEEE-488

Syntax:

```
connect [device <device list>] [, start
     <ref pin 1>] [, stop <ref pin 2>] [, clock
     <ref pin 3>] [, enable <ref pin 4>] [, common
     <ref pin 5>] [, clear <clear state>]

connect (<device list>, <ref pin 1>, <ref pin 2>,
      <ref pin 3>, <ref pin 4>, <ref pin 5>,
      <clear state>)

connect ()
```

Syntax Diagram:



Description:

Prompts the operator to connect the external lines (START, STOP, CLOCK, ENABLE, and COMMON) for the probe or an I/O module.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
ref pin 1	Pin to which external start should be connected. (Default = "not used")

ref pin 2	Pin to which external stop should be connected. (Default = "not used")
ref pin 3	Pin to which external clock should be connected. (Default = "not used")
ref pin 4	Pin to which external enable should be connected. (Default = "not used")
ref pin 5	Pin where the common lead should be connected. (Default = "not used")
clear	"yes" or "no" (Default = "no")

Example 1:

```
mod = clip ref "U22",pins 40
connect device mod, start "U33-1", stop "U18-2",
 enable "U44-2", clock "U4-5", common "U5-7"
```

Example 2:

```
connect device "/mod3", start "U2-7", stop "U22-8"
```

Example 3:

```
connect ("/mod1","U3-6","U12-10","U15-6",
 "U2-16","TP1","no")

     ! This connect command prompts the operator
     ! to connect the external lines of I/O module
     ! #1 as follows:
     !
     !    START  to U3  pin 6
     !    STOP   to U12 pin 10
     !    CLOCK to U15 pin 6
     !    ENABLE to U2  pin 16
     !    COMMON to test point 1
```

Example: 4

```
connect device "/mod3", clear "yes"

    ! Resets all connection data to "not used"
```

Remarks:

The *connect* command prompts the operator to connect the external lines (START, STOP, CLOCK, ENABLE, and COMMON) for the probe or an I/O module and press the ready button on the probe or I/O module adapter. Program execution is suspended until a Ready button is pressed.

If all the external lines are already positioned correctly, the operator is not required to press the ready button. The current positions are displayed on the operator's display. The *connect* command also causes the system to update its internal table of the UUT locations of the external lines.

The clear argument for the *connect* command can be used to reset all connection data for the specified device list. If the clear argument is "yes", all connection data is reset to default values ("not used"). If the clear argument is "no", the other arguments in the *connect* command are used to set connection data. If any changes in connection data result from a *connect* command, the operator is prompted to make the connections.

Related Commands:

sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
cos angle <expression>

cos (<expression>)
```

Syntax Diagram:

Description:

Returns the cosine function of the floating-point argument value.

Arguments:

expression	A floating-point value, expressed in radians.

Returns:

A floating-point value between -1.0 and 1.0.

Examples:

```
f = cos (0.0)
f = cos angle theta
```

Related Commands:

acos, natural

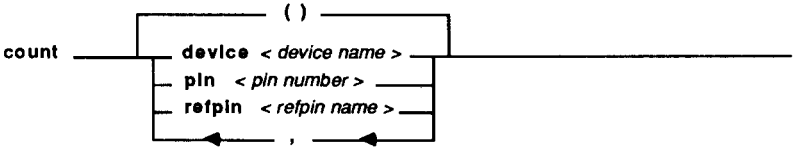
count function

IEEE-488

Syntax:

```
count [device <device name>] [, pin <pin number>]
     [, refpin <refpin name>]

count (<device name>, <pin number>, <refpin name>)

count ()
```

Syntax Diagram:



Description:

Reads the count or frequency data for one pin. This command will return useful information only after an arm . . . readout block has taken a measurement.

Arguments:

device name	I/O module name, clip module name, probe name, or reference designator. (Default = "/probe")
pin number	Pin number. (Default = 1)
refpin name	Specifies the device and pin in string format. The refpin argument is used to override the device and pin values. (Default = "")

Returns:

The count or frequency (a number). Bit 31 (decimal) is set high if the count overflows.

Example 1:

```
arm device "/probe"
  .
  .
  .
readout device "/probe"

probecount = count device "/probe"
```

Example 2:

```
arm device "/mod1"
  .
  .
  .
readout device  "/mod1"

count1 = count device "/mod1", pin 1
count2 = count device "/mod1", pin 2
count3 = count device "/mod1", pin 3
```

Example 3:

```
mod = clip ref "U3", pins 24

arm device mod
     execute stim_prog
     loop while checkstatus(mod) <> $F
     end loop
readout device mod

modcount = count device "U3", pin 22
```

Example 4:

```
mod = clip ref "U1", pins 20

arm device mod
   .
   .
   .
readout device mod

modcount = count refpin "U1-A"
           ! refpin is used because
           ! the pin name is a string
           ! value, not a number.
```

Remarks:

The *count* function returns the count or frequency for one pin. The data can be requested in terms of an I/O module pin or a component pin.

The count or frequency can be requested for a specific pin of an I/O module by specifying the module name ("/mod1", "/mod2", etc.) as the device argument. The pin argument is interpreted as an I/O module pin. Refer to Appendix E for tables that show what I/O module pin numbers to use for every possible clip module.

If a component name ("U1", "U2", etc.) is specified as the device argument, the pin argument is interpreted as a component pin. The *count* function determines the I/O module and pin number that corresponds to the specified component pin. The indicated component must have been previously named in a *clip* command.

If the string value for refpin is not a null string (""), the values of the device and pin arguments are ignored.

The *count* function should be used only after the execution of an *arm . . . readout* block.

The counter mode to be used by the *count* function is set by the *counter* command.

Related Commands:

arm, counter, level, readout, sig

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

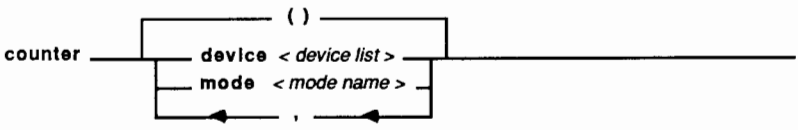
counter function

IEEE-488

Syntax:

```
counter [device <device list>] [, mode <mode name>]

counter (<device list>, <mode name>)

counter ()
```

Syntax Diagram:



Description:

Sets the counter mode for the probe or an I/O module. Allowable modes are transition count or frequency.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
mode name	"transition", or "freq". (Default = "transition")

Example 1:

```
mod = clip ref "u1", pins 40
counter device mod, mode "transition"
```

Example 2:

```
counter ("/mod1", "freq")
```

Example 3:

```
counter device "/probe,/mod1", mode "transition"
     ! the whole list of devices uses
     ! the specified mode
```

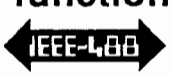
Related Commands:

arm, count, readout

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

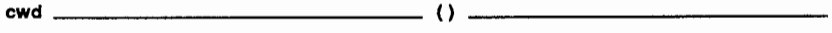
cwd function



Syntax:

```
cwd ()
```

Syntax Diagram:



Description:

Returns the current working directory as a string.

Returns:

The current working directory is returned as a string. If program execution began in a UUT directory, *cwd* returns a string of the form "/userdiskname/uutname" as in "/HDR/ABC". If execution began in the podlib, *cwd* returns a string of the form "/userdiskname/PODLIB/podname" as in "/HDR/PODLIB/80286". If execution began in the proglib, *cwd* returns a string in the form "/userdiskname/PROGLIB", as in "/DR1/PROGLIB".

Example:

```
d = cwd()
if instr (d,"PODLIB") or instr (d,"PROGLIB") then
  print "Current directory is not a UUT".
end if
```

Remarks:

The current working directory is the directory from which program execution began, not the directory of the program currently being executed. This distinction is important for programs in the proglib and podlib.

The string returned by *cwd* can be useful when constructing absolute file names for text files.

Related Commands:

filestat, open, close

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

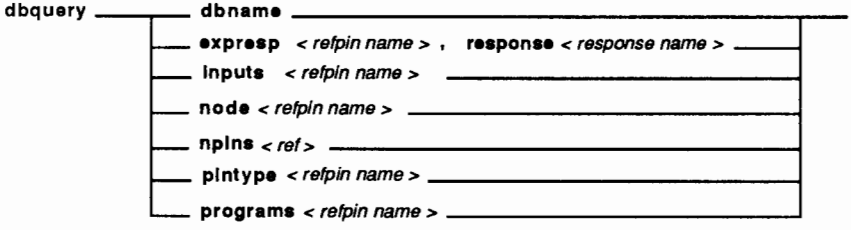
dbquery
special function

IEEE-488

Syntax:

```
dbquery dbname

dbquery expresp <refpin name>, response <response
     file name>

dbquery inputs <refpin name>

dbquery node <refpin name>

dbquery npins <ref>

dbquery pintype <refpin name>

dbquery programs <refpin name>
```

Syntax Diagram:



Description:

The *dbquery* commands allow a TL/1 program to retrieve information from the Compiled UUT Database.

Options:

dbname — (Has no argument value)

This option returns a string containing the name of the UUT compiled database ("GFIDATA" or "UFIDATA"). If the UUT does not contain a compiled database, an empty string is returned.

expresp — <refpin name>, response <response file name>

This option returns the expected response data for the named pin and stimulus program. The data is returned as a comma-separated list of "type=value" pairs, where type is "sig", "alvl", "clvl", or "count", and value is the data that appears in the response file. The list will only contain the types of response data that will be used to compare.

inputs — <refpin name>

This option returns a string containing a comma-separated list of related input pins for the named pins. An empty string is returned if the database does not contain the named pin or it has zero related inputs.

node — <refpin name>

This option returns a string containing a comma-separated list of pins that are members of the same node as the named pin. If the database is for UFI (which does not use a nodelist), an empty string is returned. If the database is for GFI and the named pin did not appear in the nodelist, an empty string is returned.

npins — <ref>

This option returns a string containing the number of pins on the named reference designator. If the database does not contain the named reference designator, an empty string is returned.

pintype <refpin name>

This option returns a string identifying the pin type of the named pin ("INP", "OUT", "BID", "PWR", "GND", or "UNU"). An empty string is returned if the database does not contain the named pin.

programs <refpin name>

This option returns a comma-separated list of TL/1 stimulus programs that will be used to test the named pin. This includes programs that test the pin as an input and programs that test it as an output. If the database does not contain the named pin, or if the database does not describe how to test the named pin, an empty string is returned.

Example 1:

```
        ! print the list of pins that are on the same
        ! node as U1-25
print "node = ", (dbquery node "U1-25")
```

Example 2:

```
        ! print the number of pins on R33
n = dbquery npins "R33"
print "R33 has ", n, " pins"
```

Example 3:

```
        ! This example prints the list of programs
        ! that are used to test U1-b4
list = dbquery programs "U1-b4"
  print "U1-b4 is tested by the following programs: ", list
```

Remarks:

The compiled UUT database is used by the resident GFI software. The 9100A/9105A will automatically load the database off disk and into memory the first time a *dbquery* command is executed. However, you must use the *gfi clear* command to unload the database when you are finished accessing it. Failure to do so will decrease the amount of memory available to the 9100A/9105A.

Refer to the *Programmer's Manual* for a description of GFI and for information on how to create a UUT database for GFI.

Related Commands:

gfi clear

For More Information:

- The "Guided Fault Isolation (GFI)" section of the *Programmer's Manual.*

declare (block form)
statement block

Syntax:

```
declare
```

Syntax Diagram:

Description:

Specifies the beginning of a declaration block.

Example 1:

```
declare
     numeric u          ! local numeric
     string v           ! local string
     numeric w = 0      ! local numeric with default
                        ! value zero
     floating f = 1.3 ! local floating with default
end declare
```

Example 2:

```
declare
     global numeric 'last_address'
      persistent string lastmsg
end declare
```

Remarks:

Declarations take effect for the entire block that encloses them. All declarations must appear before any executable statements in the enclosing block.

Persistent variables may not be declared as arrays.

Arguments to the enclosing block may not be declared as global or persistent variables.

Related Commands:

declare (statement form), end, exercise, function, handle, program

For More Information:

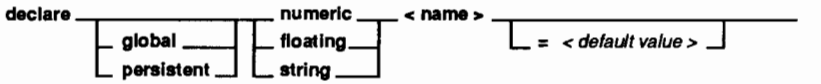
- The "Overview of TL/1" section of the *Programmer's Manual.*

declare (statement form)
statement

Syntax:

```
declare [global|persistent] numeric|string|floating
     <name> [=<default value>]

declare [global|persistent] numeric|string|floating
     array [<array dimensions>] <name>

<array dimension>
     [<number>:<number> {,<number>:<number>}]
```

Syntax Diagram:



Description:

Declares the scope, data type, and name of a single variable. Also defines the dimensions of an array.

Arguments:

name	Name of the variable being declared.
default value	An explicit value initially assigned to the variable. (Optional.)

array dimensions	Specification of the dimensions of the array. Each dimension gives the first and last permitted value of the corresponding subscript expression.

Example 1:

```
declare numeric array [1:10,1:10] x
     ! variable x is a 10x10 array with 100 cells
     ! and has the numeric data type.
```

Example 2:

```
declare numeric nano
     ! nano is declared as a numeric variable
```

Example 3:

```
declare global string hi_all
     ! the name hi_all is a string variable with
     ! global scope
```

Example 4:

```
declare floating f = 7.99
     ! f is declared as a floating-point variable
     ! with a default valve of 7.99
```

Example 5:

```
declare persistent numeric perseus
     ! perseus is declared as a
     ! persistent numeric variable
```

Remarks:

Declarations take effect for the entire block that encloses them. All declarations must appear before any executable statements in the enclosing block.

Persistent variables may not be declared as arrays.

Arguments to the enclosing block may not be declared as global or persistent variables.

Related Commands:

declare (block form), exercise, function, handle, program

For More Information:

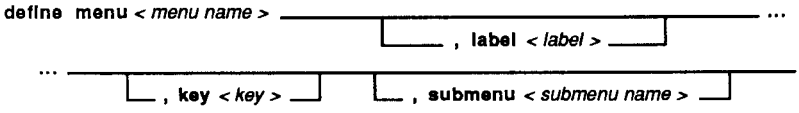
- The "Overview of TL/1" section of the *Programmer's Manual.*

define menu
special function

Syntax:

```
define menu <menu name>, [, label <label>] [, key <key>]
     [, submenu <submenu name>]
```

Syntax Diagram:



Description:

Defines a menu or menu item.

Arguments:

menu name	Name of the menu or menu item being defined. If the menu already exists, the arguments included in the define menu command are modified. If the menu does not exist, a new one is created.
label	Label to be displayed for the menu item. If not specified, and a new menu item is being created, it will default to the menu identifier.
key	The key to be associated with the menu item. Only the first character is significant. If there is no key code, that menu item can only be selected using cursor or button controls.

submenu name	The submenu points to another menu (the MMMM part of the identifier). If this item is selected, the menu defined becomes the new menu.

Example:

```
        ! define two menus, one called M1 and one
        ! called M2.  M1 allows the keys 1, 2, and 3
        ! to be used to select menu items.  Menu M2
        ! does not use key entry.  If item M1-C is
        ! selected, menu M2 becomes active.

define menu "M1-A", label "RAM test", key "1"
define menu "M1-B", label "ROM test", key "2"
define menu "M1-C", submenu "M2", label "other
  tests", key "3"
define menu "M2-A", label "BUS test"
define menu "M2-B", label "I/O test"
```

Remarks:

Menu definitions are used by the *readmenu* command when displaying to or reading from a menu on the monitor.

Menu entry identifiers are strings of the form "MMMM-IIII". The characters in front of the first "-" are considered to be the menu name. All menu entries with the same menu name are collected together into the same menu. The menu items are listed in the order they are defined for that menu. The entry IIII is the item name for the menu. This is significant when using *readmenu*. When *readmenu* returns a menu selection, it returns "MMMM-IIII" as it was defined by the *define menu* command. That is, the identifier is used to indicate which selection was made on the menu.

Passing an identifier such as "MMMM" with no item causes a menu to be created with no items if none exists, and does nothing if the menu already exists.

If a menu item already exists, specifying it will cause the elements of the menu item to be modified. Only the items included in the *define menu* command will be modified.

Once a TL/1 invocation has started, the menu definitions are global and remain in force until explicitly removed using the *remove* command, or until TL/1 is restarted. TL/1 is restarted whenever you press the REPEAT or EXEC keys on the operator's keypad, by pressing the INIT softkey when in the debugger, or by pressing the EXECUTE softkey when in the debugger if a program is not currently executing.

Note that each definition uses a small increment of memory. That increment is typically 10 to 30 bytes of memory for book keeping plus one byte for each character in the string or strings associated with this definition. If many menu definitions are to be made (and the program is intended to be run indefinitely), it is good practice to remove menu definitions using the *remove* command. This practice will prevent memory from being consumed by unused or obsolete definitions.

Related Commands:

readmenu, remove

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

define mode
special function

Syntax:

```
define mode <mode name>, attribute <attrlist>
```

Syntax Diagram:

Description:

Used to define the way a reference designator mode is displayed in a window.

Arguments:

mode name	A name you select to refer to this mode definition. Recommended names are "testing", "passed", "failed", "untested", and "partial".
attrlist	"normal", "blink", "bold", and "inverse". Combinations are also allowed, if separated by commas but including no spaces. If "normal" appears in the comma-separated list, the attributes preceding "normal" in attrlist are ignored.

Example:

```
define mode "testing", attribute "blink"
define mode "passed", attribute "blink,bold"
define mode "failed", attribute "inverse"

     ! Drawing a ref in testing mode results in a
     ! blinking part, passed mode results in
     ! blinking bold, and failed mode results in
     ! inverse video.
```

Remarks:

Mode definitions are used by the *draw ref* command to draw the parts of a UUT and to display the status of testing for that UUT in a window on the monitor.

The modes "testing", "passed", "failed", "partial" (partially passed, not completely tested) and "untested" are the set of recommended modes. There is however no enforced restriction on mode names.

Once a TL/1 invocation has started, the mode definitions are global and remain in force until explicitly removed using the *remove* command, or until TL/1 is restarted. TL/1 is restarted whenever you press the REPEAT or EXEC keys on the operator's keypad, by pressing the INIT softkey when in the debugger, or by pressing the EXECUTE softkey when in the debugger if a program is not currently executing.

Note that each definition uses a small increment of memory. That increment is typically 10 to 30 bytes of memory for book keeping plus one byte for each character in the string or strings associated with this definition. If many mode definitions are to be made (and the program is intended to be run indefinitely), it is good practice to remove mode definitions using the *remove* command. This practice will prevent memory from being consumed by unused or obsolete definitions.

Related Commands:

draw, draw ref, remove

For More Information:

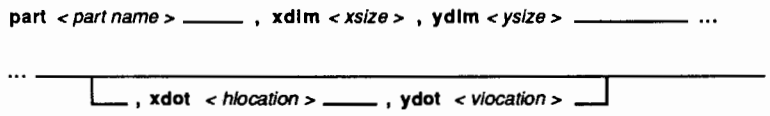
- The "Overview of TL/1" section of the *Programmer's Manual.*

define part
special function

Syntax:

```
define part <part name>, xdim <xsize>, ydim <ysize>
     [, xdot  <hlocation>, ydot <vlocation>]
```

Syntax Diagram:



Description:

Used to define a part shape and size for the *draw* commands. Defining a part causes no action to take place on the monitor.

Arguments:

part name	A name you select to refer to this part definition.
xsize	Horizontal size of the part to be displayed in scaled window coordinates (see the open command).
ysize	Vertical size of the part to be displayed in scaled window coordinates (see the open command).
hlocation	Horizontal location of the alignment dot within the part in scaled window coordinates. If xdot and ydot are not specified, there is no dot displayed.
vlocation	Vertical location of the alignment dot within the part in scaled window coordinates. If xdot and ydot are not specified, there is no dot displayed.

Example 1:

```
define part "dip", xdim 3, ydim 10, xdot 0, ydot 0

     ! A part name "dip" will be created with
     ! scaled window dimensions of (3,10) and a
     ! dot located in the upper left-hand corner.
```

Example 2:

```
define part "box", xdim 10, ydim 10

     ! A part named box with no dot will be
     ! created with scaled window dimensions of
     ! (10,10).
```

Remarks:

Part definitions are used by the *draw ref* command to draw a UUT in a window on the monitor. To display a part, you first define its shape and size with the *define part* command, then you give it a reference designator name and a location with the *define ref* command, and finally draw the part on a window of the monitor with the *drawref* command. Doing a *define part* on an existing part causes the information to be replaced with the new information.

Once a TL/1 invocation has started, the part definitions are global and remain in force until explicitly removed using the *remove* command, or until TL/1 is restarted. TL/1 is restarted whenever you press the REPEAT or EXEC keys on the operator's keypad, by pressing the INIT softkey when in the debugger, or by pressing the EXECUTE softkey when in the debugger if a program is not currently executing.

Note that each definition uses a small increment of memory. That increment is typically 10 to 30 bytes of memory for book keeping plus one byte for each character in the string or strings associated with this definition. If many definitions are to be made (and the program is intended to be run indefinitely), it is good practice to remove part definitions using the *remove* command. This practice will prevent memory from being consumed by unused or obsolete definitions.

Related Commands:

define ref, draw, draw ref, open, remove

For More Information:

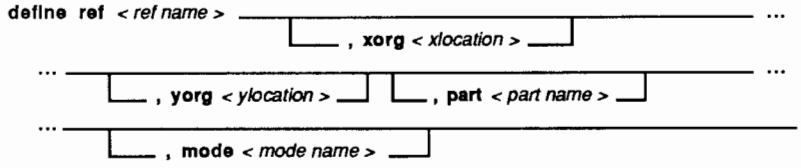
- "The Overview of TL/1" section of the *Programmer's Manual.*

define ref
special function

Syntax:

```
define ref <ref name> [, xorg <xlocation>] [, yorg
     <ylocation>]  [, part <part name>] [, mode
     <mode name>]
```

Syntax Diagram:



Description:

Used to define a reference designator. It associates a reference designator name with a location in a window and a part shape previously defined by the *define part* command. To display the reference designator, use a *draw* command.

Arguments:

ref name	A reference designator name you select.
xlocation	A numeric expression for the horizontal location (in scaled window coordinates) of the upper left-hand corner of the ref. If xorg is not specified, the old horizontal location is unchanged. If xorg was never specified, it defaults to 0.
ylocation	A numeric expression for the vertical location (in scaled window coordinates) of the upper left-hand corner of the ref.

	If yorg is not specified, the old vertical location is unchanged. If yorg was never specified, it defaults to 0.
part name	Part name of a *define part* definition. (If not specified, the part shape is unchanged).
mode name	The name of the current display mode for the component. (If not specified, the mode is unchanged).

Example:

```
define ref "U3", xorg 1, yorg 1, part "dip", mode
 "untested"

     ! A ref defined as U3 will be at location 1,1
     ! drawn with the part shape and size
     ! specified in the part definition named
     ! "dip".  The display mode will be that
     ! specified for untested components.
```

Remarks:

Reference designator definitions are used (along with *define part* commands) by a *draw* command to draw UUT components in a window on the monitor.

Once a TL/1 invocation has started, the ref definitions are global and remain in force until explicitly removed using the *remove* command, or until TL/1 is restarted. TL/1 is restarted whenever you press the REPEAT or EXEC keys on the operator's keypad, by pressing the INIT softkey when in the debugger, or by pressing the EXECUTE softkey when in the debugger if a program is not currently executing.

Note that each definition uses a small increment of memory. That increment is typically 10 to 30 bytes of memory for book keeping plus one byte for each character in the string or strings associated with this definition. If many definitions are to be made (and the program is intended to be run indefinitely), it is good practice to remove reference designator definitions using

the *remove* command. This practice will prevent memory from being consumed by unused or obsolete definitions.

A run-time error is caused by attempting to draw a ref with no part defined or with no mode defined for the part.

Related Commands:

define part, draw, draw ref, open, remove

For More Information:

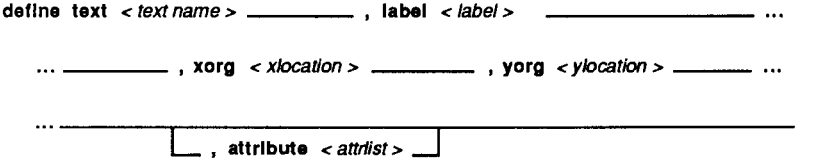
- The "Overview of TL/1" section of the *Programmer's Manual.*

define text
special function

Syntax:

```
define text <text name>, label <label>, xorg
     <xlocation>,  yorg <ylocation> [, attribute
     <attrlist>]
```

Syntax Diagram:



Description:

Used to define text to be displayed in a window using the scaled location coordinates of a *draw* command. A piece of text is associated with attributes and a location in the window.

Arguments:

text name	A string expression for the name of the text to be defined.
label	The text to be displayed.
xlocation	A numeric expression for the horizontal location (in scaled window coordinates) of the upper left-hand corner of the text.
ylocation	A numeric expression for the vertical location (in scaled window coordinates) of the upper left hand corner of the text.

attrlist — "normal", "blink", "bold", and "inverse". Combinations are also allowed, if separated by commas but including no spaces.

If "normal" appears in the comma-separated list, the attributes preceding "normal" in attrlist are ignored.

Example:

```
        ! Create a label named U1 starting at
        ! location 10,10 to identify U1 as a 68000
        ! chip.  Note that the text name and ref name
        ! do not need to be the same.  It is done in
        ! this example for convenience.

define text "U1", label "68000", xorg 10, yorg 10,
  attribute "bold"
```

Remarks:

Text definitions are used by a *draw* command to draw labels in a window on the monitor.

Once a TL/1 invocation has started, the text definitions are global and remain in force until explicitly removed using the *remove* command, or until TL/1 is restarted. TL/1 is restarted whenever you press the REPEAT or EXEC keys on the operator's keypad, by pressing the INIT softkey when in the debugger, or by pressing the EXECUTE softkey when in the debugger if a program is not currently executing.

Note that each definition uses a small increment of memory. That increment is typically 10 to 30 bytes of memory for book keeping plus one byte for each character in the string or strings associated with this definition. If many definitions are to be made (and the program is intended to be run indefinitely), it is good practice to remove reference designator definitions using the *remove* command. This practice will prevent memory from being consumed by unused or obsolete definitions.

Related Commands:

draw, draw ref, open, remove

For More Information:

- "The Overview of TL/1" section of the *Programmer's Manual.*

delete function

IEEE-488

Syntax:

```
delete file <file name>

delete (<file name>)
```

Syntax Diagram:

Description:

Deletes a text file from the userdisk or from a UUT directory.

Arguments:

file name	A string which specifies the path of the text file to be deleted. If a full pathname is not used, the text file will be deleted from the current UUT directory.

Example 1:

```
delete file "test1"  ! deletes a UUT text file
```

Example 2:

```
testdata = "/HDR/test"
delete testdata      ! deletes a userdisk text file
```

Remarks:

The *delete* command performs the same function as the REMOVE softkey in the editor, with the exception that the *delete* command removes only files of the type TEXT.

Related Commands:

open, print

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

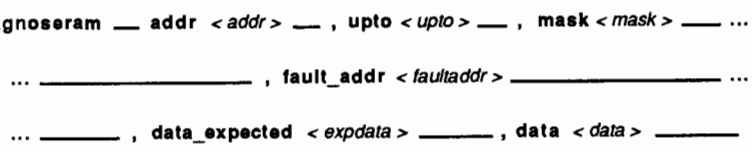
diagnoseram function

IEEE-488

Syntax:

```
diagnoseram addr <addr>, upto <upto>, mask <mask>,
     fault_addr <faultaddr>, data_expected
     <expdata>,  data <data>

diagnoseram (<addr>, <upto>, <mask>, <faultaddr>,
     <expdata>,  <data>
```

Syntax Diagram:



Description:

Used with your customized RAM tests or with Pod Quick Tests to provide diagnostics and fault conditions which are consistent with those of the *testramfast* and *testramfull* RAM tests.

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask for data bits to test.
faultaddr	Address of detected fault.
expdata	Data expected from fault address.
data	Data actually read from fault address.

Example:

```
diagnoseram addr 0, upto $FFFE, mask $7F,
  fault_addr $10DA, data_expected $AA, data $A8
```

Remarks:

The *diagnoseram* command is an aid for using Pod Quick tests, or for using customer-designed tests such as a downloaded RUN UUT test. The *diagnoseram* command supplies the same 9100A/9105A diagnostics and fault conditions that would result from the *testramfast* and *testramfull* commands.

Related Commands:

pretestram, testramfast, testramfull

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual* .

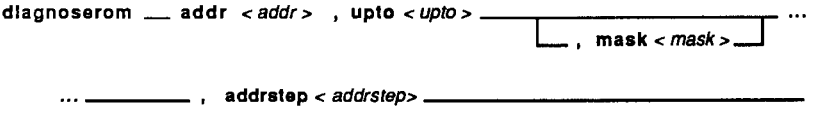
diagnoserom function

IEEE-488

Syntax:

```
diagnoserom addr <addr>, upto <upto> [, mask
     <mask>], addrstep <addrstep>

diagnoserom (<addr>, <upto>, <mask>, <addrstep>)
```

Syntax Diagram:



Description:

Used with your customized ROM tests or with Pod Quick Tests to preform data and address diagnostics on a range of ROM. The diagnostics and fault conditions are consistent with those of the *testromfull* ROM test.

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask of valid data bits. (Default = $FFFFFFFF)
addrstep	Address increment.

Example 1:

```
diagnoserom addr $E000, upto $EFFF, mask $F0F0,
  addrstep 2
```

Example 2:

```
diagnoserom addr adstart, upto (adstart + $0FFF),
  addrstep 1
```

Remarks:

The *diagnoserom* command is similar to *testromfull* in functionality and results. The major difference is *testromfull* performs both a test of the ROM (by comparing the ROM signature with an expected signature), and diagnostics if the signatures do not match. The *diagnoserom* command does not test the ROM (so it does not require a valid ROM signature) and goes right into the diagnostics.

Another difference with *diagnoserom* is it tests for address line faults first, while *testromfull* tests for data line faults first. If there are both address faults and data faults, the two tests will report different results.

The *diagnoserom* command is intended for use with your customized ROM tests, or when a Pod Quick Test indicates that the ROM signature is incorrect.

Related Commands:

testromfull

For More Information:

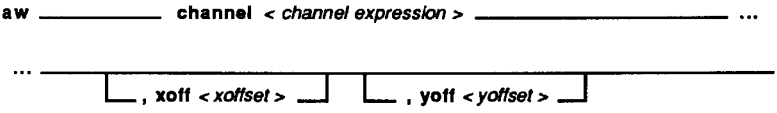
- The "Overview of TL/1" section of the *Programmer's Manual*.

draw
special function

Syntax:

```
draw channel <channel expression> [, xoff <xoffset>
     [, yoff <yoffset>]
```

Syntax Diagram:



Description:

Draws all of the previously defined UUT components and then all of the defined text on a window in the monitor.

Arguments:

channel expression	A numeric expression for a channel opened to write on the desired window. Remember that /term1 and /term2 are also considered windows.
xoffset	A numeric expression to define the horizontal offset used when drawing the ref and text. This value is subtracted from the value of xorg defined by the *define ref* and *define text* commands.
yoffset	A numeric expression to define the vertical offset used when drawing the ref and text. This value is subtracted from the value of yorg defined by the *define ref* and *define text* commands.

Example:

```
draw channel ch3

     ! Display all defined UUT components and all
     ! defined text on the window opened for
     ! writing through a channel named ch3.
```

Related Commands:

define mode, define part, define ref, define text, open

For More Information:

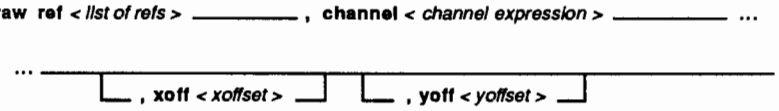
- The "Overview of TL/1" section of the *Programmer's Manual.*

draw ref
special function

Syntax:

```
draw ref <list of refs>, channel <channel
     expression>, [, xoff  <xoffset>] [, yoff
     <yoffset>]
```

Syntax Diagram:



Description:

Draw previously defined UUT components on a window in the monitor. This command can also be used to change the display mode of a component or list of components.

Arguments:

list of refs	A list of reference designators defined by *define ref* commands. When more than one reference designator is included in the list, the reference designators are separated by commas (but no spaces are used). If this argument is an empty string, all defined refs are drawn.
channel expression	A numeric expression to define a channel opened to write on the desired window. Remember that /term1 and /term2 are also considered windows.

xoffset	A numeric expression to define the horizontal offset used when drawing the ref. This value is subtracted from the value of xorg defined by the *define ref* command.
yoffset	A numeric expression to define the vertical offset used when drawing the ref. This value is subtracted from the value of yorg defined by the *define ref* command.

Example:

```
draw ref "U3,U5,U22,U1", channel ch2
     ! Display the specified group of UUT
     ! components on a window written to by a
     ! channel named ch2
```

Related Commands:

define mode, define part, define ref, draw, open

For More Information:

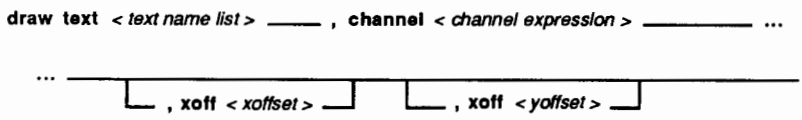
- "The Overview of TL/1" section of the *Programmer's Manual.*

draw text
special function

Syntax:

```
draw text <text name list>, channel <channel
     expression> [, xoff <xoffset>] [, yoff
     <yoffset>]
```

Syntax Diagram:



Description:

Displays the text at the scaled location indicated.

Arguments:

text name list	A list of text names created by *define text* commands. When more than one text name is included in the list, the names are separated with commas, but no spaces are used. If this argument is the null string (""), all defined text is shown.
channel expression	A numeric expression to define a channel opened to write on the desired window. Remember that /term1 and /term2 are also considered windows.
xoffset	A numeric expression to define the horizontal offset used when drawing the text. This value is subtracted from the value of xorg defined by the *define text* command.

yoffset	A numeric expression to define the vertical offset used when drawing the text. This value is subtracted from the value of yorg defined by the *define text* command.

Example 1:

```
draw text "lab1,lab2", channel chan1
     ! draw only lab1 and lab2 on a window opened
     ! for writing through a channel named chan1
```

Example 2:

```
draw text "", channel chan1
     ! draw all defined text on a window opened
     ! for writing through a channel named chan1
```

Related Commands:

define text, draw, open

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

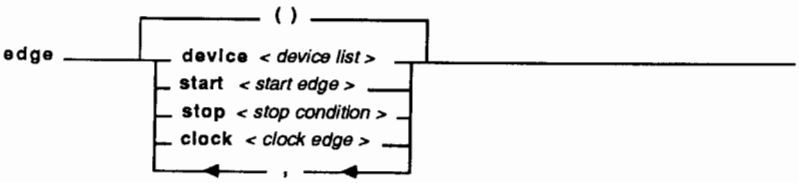
edge
function

IEEE-488

Syntax:

```
edge [device <device list>] [, start <start edge>]
     [, stop <stop condition>][, clock <clock
     edge>]

edge (<device list>, <start edge>, <stop
     condition>, <clock edge>)

edge ( )
```

Syntax Diagram:



Description:

Specifies the active edge of the external START, STOP, and CLOCK lines for the probe or an I/O module. Allowable edges are rising ("+") or falling ("-"); "count" is also allowed for the stop condition.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
start edge	I/O module external start edge "+", "-", or "at_arm". (Default = "+")

	Probe external start edge "+" or "-". (Default = "+")
stop condition	External stop edge "+" or "-", or "count" condition (value determined by the TL/1 stopcount function). (Default = "+")
clock edge	External clock edge "+" or "-". (Default = "+")

Example:

```
mod = clip ref "u54", pins 16
edge device mod, start "+", stop "-", clock "+"
```

Related Commands:

arm, readout, stopcount

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

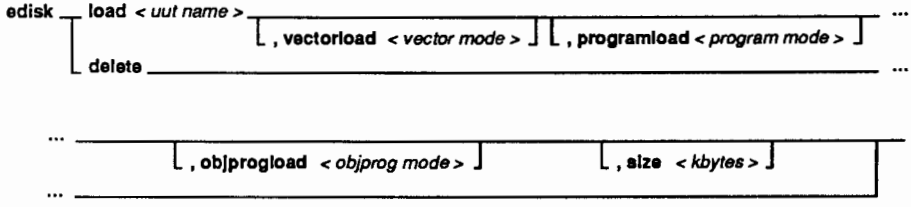
edisk
special function

IEEE-488

Syntax:

```
edisk load <uut name> [, vectorload <vector mode>]
[, programload <program mode>]
[, objprogload <objprog mode>] [, size <kbytes>]

edisk delete
```

Syntax Diagram:



Description:

Creates or deletes the E-disk, a temporary RAM cache. TL/1 programs and TL/1 compiled programs are optionally loaded into the E-disk to increase their execution speed. The load operation loads all the GFI database and optionally, the test vectors (VECTOR), the compiled programs (OBJPROG), and the source programs (PROGRAM), in the specified UUT directory into the E-disk. If a pod is specified by SETUP POD NAME, all the TL/1 programs in the specified POD directory are also loaded.

Arguments:

uut name	A string which specifies the path of the UUT to be loaded. If a full path is not used, the UUT is loaded from the current user disk.
vector mode	"ON" - test vectors are to be loaded. "OFF" - test vectors are not to be loaded. (Default = "OFF")

program mode	"ON" - programs (PROGRAM) are to be loaded. "OFF" - programs are not to be loaded. (Default = "ON")
objprog mode	"ON" - compiled programs (OBJPROG) are to be loaded. "OFF" - compiled programs are not to be loaded. (Default = "ON")
kbytes	Size of the E-disk in kilobytes. (Default = 500K bytes)

Example 1:

```
! create an E-disk of size 1000 kbytes and
! load it from user disk DR1, uut DEMO,
! including the test vectors, PROGRAMs, and
! OBJPROGs

edisk load "/DR1/DEMO", vectorload "ON", size 1000
```

Example 2:

```
! create an E-disk of size 500 kbytes and load
! it from user disk HDR, uut TK80 without the
! test vectors, without PROGRAMs, but
! including OBJPROGs.

edisk load "/HDR/tk80", programload "OFF"
```

Example 3:

```
! delete the E-disk

edisk delete
```

Remarks:

To delete the E-disk, use the delete form of the E-disk function. The E-disk is also automatically deleted when the EDIT key is pressed on the 9100A. Either procedure releases the previously allocated E-disk memory for use with other operations.

To create the E-disk, you must ascertain that sufficient memory is available in RAM. Once the E-disk is created, you may also have to adjust RAM allocations so that there is enough left for other functions. Insufficient RAM before and after the creation of the E-disk results in error messages. Adjustments to RAM to create a balance of available memory are made with the kbytes argument.

To adjust the size of the E-disk, reduce the number of kbytes until an error message no longer occurs. Another way to make memory available is to turn off VECTORs, PROGRAMs, or OBJPROGs.

If an E-disk currently exists when this statement is issued, one of the following occurs:

- If the UUT directory name of the current E-disk does not match the specified UUT directory name, the original files are deleted and the new files loaded.
- If the UUT directory name of the current E-disk matches the newly specified UUT directory, a merge copy takes place.

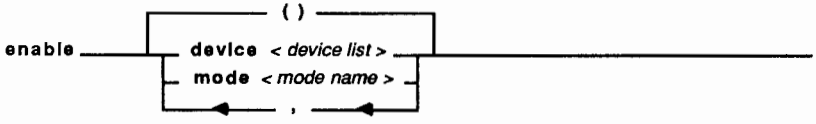
enable function

IEEE-488

Syntax:

```
enable [device <device list>] [, mode <mode name>]

enable (<device list>, <mode name>)

enable ()
```

Syntax Diagram:



Description:

Sets the enable mode of the response gathering hardware for the probe or an I/O module. Used only when the sync mode is "ext".

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
mode name	I/O module enable modes: "always", "high", "low", or "pod". Probe enable modes: "always", "high", "low", "pod", "pod*en0", "pod*en1". (Default = "always")

Example 1:

```
mod = clip ref "U22", pins 28
enable device mod, mode "high"
```

Example 2:

```
enable device "/probe", mode "pod"
```

Remarks:

The enable mode is one of the following:

"high"	Enable condition is true when the external enable line is high.
"low"	Enable condition is true when the external enable line is low.
"always"	Enable condition is always true.
"pod"	Enable condition is generated by the pod sync pulse.
"pod*en0"	Enable condition is generated by the pod sync pulse ANDed with the enable signal. The enable condition is true if the pod's PodSync line is active and the clock module enable line is low. This mode is valid only for the probe.
"pod*en1"	Enable condition is generated by the pod sync pulse ANDed with the enable signal. The enable condition is true if the pod's PodSync line is active and the clock module enable line is high. This mode is valid only for the probe.

Related Commands:

arm, readout, sync

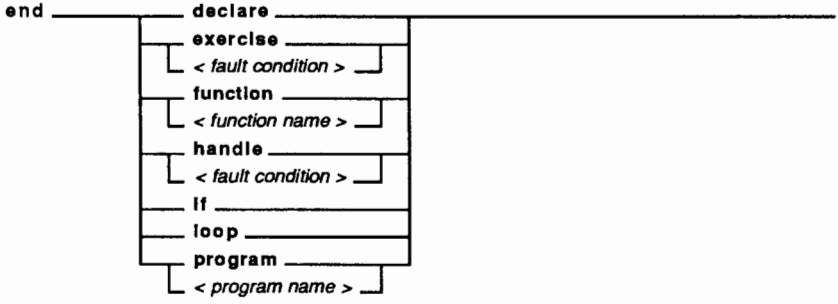
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
end declare
end exercise
end <fault condition>
end function
end <function name>
end handle
end <fault condition>
end if
end loop
end program
end <program name>
```

Syntax Diagram:



Description:

Defines the end of a block.

Arguments:

fault condition	The fault condition used to call an exerciser block or a handler block.
function name	The function name used to call the function block.

program name	The program name used to call the program block.

Examples:

```
end declare   ! ends a variable declaration block

end if        ! ends an if . . . then or an
              ! if . . . then . . . else
              ! block

end loop      ! ends a loop block

end my_exer   ! ends an exerciser block, where the
              ! fault name given in the
              ! exercise statement is my_exer

end my_func   ! ends a function, where the function
              ! name given in the function
              ! statement is my_func

end my_handl  ! ends a fault-handler block, where
              ! the fault name
              ! given in the handle statement
              ! is my_handl

end my_prog   ! ends a program, where the program
              ! name given in the program statement
              ! is my_prog
```

Related Commands:

declare (block form), exercise, function, handle, if, loop, program

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

endif
statement

Syntax:

```
endif
```

Syntax Diagram:

Description:

Specifies the end of an if block.

Example:

```
if a < 4 then
     b = 0
     c = 1
endif
```

Remarks:

In TL/1 *endif* is functionally equivalent to *end if*. This form of the *end if* statement is provided for ANSI compatibility, but is not the recommended TL/1 syntax.

Related Commands:

if, end

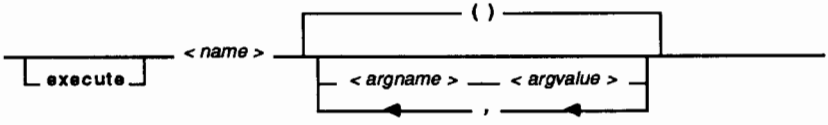
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
[execute] <name> <argname> <argvalue> {, <argname>
     <argvalue>}

[execute] <name> (<argvalue> {, <argvalue>})

[execute] <name> ()
```

Syntax Diagram:



Description:

Causes the named program or function to be executed. The execute keyword is optional.

Arguments:

name	Valid program or function name.
argname	Name of an argument.
argvalue	An expression which provides the value for an argument.

Example 1:

```
! Suppose you have written a function called
! send which requires two arguments:
! addr and data. This is an example of
! calling send using keyword notation.

execute send addr a, data d
```

Example 2:

```
        ! This is equivalent to Example 1, but the
        ! argument list is written in positional
        ! notation.

execute send (a, d)
```

Example 3:

```
program test4
     function f1     ! defines the function f1
          ch1 = open device "/term2", as "output"
          print on ch1 "Executed function f1"
          close channel ch1
     end f1
     .
     .
     execute f1 ()   ! calls the function f1
     .
     .
     execute f1 ()   ! calls the function f1
     .
     .
     execute test3 ()   ! calls another program
                        ! since this name is not
                        ! defined as a function in
                        ! this program
end program
```

Remarks:

The argument list for the *execute* command has two forms:

- Keyword notation - where argument name and argument value pairs are listed with each pair separated by commas.

- Positional notation - where only the argument values are listed in order separated by commas. The order of argument values must be the same as the order in the *program* or *function* command called by the *execute* command.

The value returned by a function may be used as the argument to another function, as in the statement z = f(g(x),y). When the functions use keyword notation, it may be difficult to interpret the resulting statement. In the following invocation:

```
z = f f_arg1 g g_arg1 x, arg2 y
```

it is not obvious whether arg2 belongs to g's argument list or f's argument list. Therefore, when a function invocation using keyword notation is an argument to another function, the argument function invocation must be surrounded by parentheses to remove ambiguity as shown in the following:

```
execute f f_arg1 (g g_arg1 x), arg2 y
```

Related Commands:

function, program, return

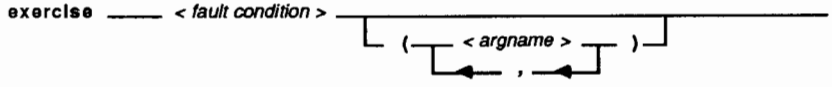
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
exercise <fault condition> [(<argname>
     {, <argname>})]
```

Syntax Diagram:



Description:

Specifies the beginning of a fault condition exerciser.

Arguments:

fault condition	Name of the fault condition to be exercised. (See Appendix G, "Handling Built-in Fault Messages in TL/1 Programs," in this manual.)
argname	Name of an argument for this exerciser block.

Example:

```
exercise my_exerciser (a,b,c)

     declare
          numeric e,f
     end declare

     .
     .
     .

end my_exerciser
```

Remarks:

A fault condition exerciser is a sequence of statements that detects a particular, previously discovered fault condition. When a fault condition occurs, an operator can execute the exerciser continually (controlled through the operator's display) while attempting to repair the UUT fault. A program or function can provide exercisers for fault conditions that are raised during execution.

You may write your own exercisers in TL/1 or use the built-in exercisers provided by the built-in tests.

The fault name must be the same in both the *exercise* command and the *end* command. A fault exerciser has the name of the fault condition it is intended to exercise.

The argument list consists of argument names separated by commas. If any arguments have default values, these values are assigned in the declarations. The *exercise* command must include all arguments named in the corresponding *fault* command, but it can include additional arguments as well.

Unlike the scope of a variable name, which is static based upon the block structure of a TL/1 program, the scope of an exerciser is dynamic. The scope of an exerciser extends from the time at which the containing block is entered until the containing block is exited. If the block invokes another function, the exerciser remains "active", unless another exerciser with the same name is activated in the invoked block.

Variables declared inside an exerciser are local unless explicitly declared to be global.

Any fault condition that is exercised causes the program to indicate that the UUT fails if the last full iteration of the exerciser detected a fault and allows the program to indicate a "passes" if the last full iteration of the exerciser did not detect a fault.

Related Commands:

declare, end, execute, fails, handle, passes, refault, return

For More Information:

The "Overview of TL/1" section of the *Programmer's Manual.*

fabs
function

IEEE-488

Syntax:

```
fabs num <expression>

fabs (<expression>)
```

Syntax Diagram:

Description:

Returns the absolute value (floating-point) of the argument value.

Arguments:

expression	The floating-point argument value.

Returns:

A floating-point value.

Examples:

```
f = fabs (f)                ! Convert f to its absolute
                            ! value.
f = fabs num (sin angle f)
```

Remarks:

If	fabs returns
<expression> ≥ 0.0	<expression>
<expression> < 0.0	- <expression>

fails
operator

Syntax:

```
<invocation> fails
```

Syntax Diagram:

Description:

Tests the termination status of a called program or function. The *fails* operator evaluates as true if the called function or program ends with a fail status and as false otherwise.

Arguments:

invocation	Program or function call.

Example 1:

```
if testramfull ($1000,$1FFF,2) fails then x = 0
```

Example 2:

```
if testbus ($FFFF) fails then y = 0
```

Remarks:

Termination status indicates whether or not a UUT passes functional tests. Termination status is revised for every invoked program or function.

Termination status can be:

passes	represents completion of a test without any unhandled fault conditions. The UUT is free from any faults that the test can detect.
fails	represents the existence of one or more unrepaired faults at the end of test execution.

A program that runs to completion without detecting any faults indicates that the UUT passes. Detection of a fault by the program (or any programs it calls) affects the termination status of the program. Any unhandled, unexercised fault condition causes the program to indicate that the UUT fails. Any fault condition that is exercised causes the program to indicate that the UUT fails if the last full iteration of the exerciser detected a fault and allows the program to indicate a "passes" if the last full iteration of the exerciser did not detect a fault. The termination status of a program is accumulated in the program that called it, so that if any called programs indicated a failure, the calling program also indicates that the UUT fails.

A fault condition can be handled by a block of statements called a fault condition handler. The fault condition handler has access to the arguments of the fault and the global variables of the test program. When a fault condition handler encounters either a *return* statement or its last statement the handler terminates, and execution resumes at the statement following the *fault* command.

If the handler does not execute a *fault* command, the fault condition is handled and disappears. In this case, the termination status is "passes".

A *fault* command with no fault name or arguments unconditionally sets the termination status to "fails."

When a *refault* or a *fault* command with a fault name is executed, the termination status is affected by the presence of other handlers or exercisers for the fault condition.

Related Commands:

execute, exercise, fault, handle, if, passes, refault, while

For More Information:

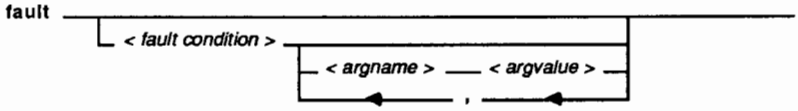
- The "Overview of TL/1" section of the *Programmer's Manual.*

fault
statement

Syntax:

```
fault [<fault condition> <argname> <argvalue>
     {, <argname> <argvalue>}]
```

Syntax Diagram:



Description:

Raises a fault condition and provides a list of arguments that describe details of the fault condition.

Arguments:

fault condition	Name of the fault condition to be raised.
argname	Name of an argument.
argvalue	An expression which provides the value for an argument.

Example 1:

```
if errors > 10 then fault      ! termination status
                               ! is set to "fails"
```

Example 2:

```
fault pod_data_tied mask mask, access_attempted
 "write", addr adr2, data d
```

Remarks:

A fault condition occurs during execution when the UUT does not respond as expected. When faulty UUT behavior is detected by a test program, a fault condition describing the behavior is raised with a *fault* command. The *fault* command consists of a fault name and a list of arguments that describe details of the fault condition.

When a *fault* command is executed, program execution is suspended. If the current invocation contains a handler for the raised fault condition, the handler is executed. If the current invocation does not contain a handler for the raised fault condition, each invocation in the calling chain is checked until a handler is found. If no invocation has a handler for the raised fault condition, the system issues the fault message on the operator's display.

A program that runs to completion without detecting any faults indicates that the UUT passes. Detection of a fault by the program (or any programs it calls) affects the termination status of the program. Any unhandled, unexercised fault condition causes the program to indicate that the UUT fails. Any fault condition that is exercised causes the program to indicate that the UUT fails if the last full iteration of the exerciser detected a fault and allows the program to indicate a "passes" if the last full iteration of the exerciser did not detect a fault. The termination status of a program is accumulated in the program that called it, so that if any called programs indicated a failure, the calling program also indicates that the UUT fails.

A fault condition can be handled by a block of statements called a fault condition handler. The fault condition handler has access to the arguments of the fault and the global variables of the test program. When a fault condition handler encounters either a *return* statement or its last statement the handler terminates, and execution resumes at the statement following the *fault* command.

If the handler does not execute a *fault* command, the fault condition is handled and disappears. In this case, the termination status is "passes".

A *fault* command with no fault name or arguments unconditionally sets the termination status to "fails."

When a *refault* or a *fault* command with a fault name is executed, the termination status is affected by the presence of other handlers or exercisers for the fault condition.

Related Commands:

execute, exercise, fault, handle, if, passes, refault, while

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- Appendix H, "Raising Built-in Fault Messages in TL/1 Programs," in this manual.

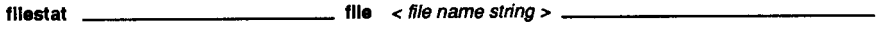
filestat function

IEEE-488

Syntax:

```
filestat file <file name string>

filestat (<file name string>)
```

Syntax Diagram:



Description:

Returns information about the existence, readability, and writeability of a text file.

Arguments:

file	A string containing a relative or absolute file path name.

Returns:

A three character string if the file exists, or the empty string if the file does not exist.

The first character of the string is "r" if the file is readable and "-" otherwise.

The second character of the string is "w" if the file is writable and is "-" otherwise.

The third character of the string is "-".

Example:

```
print "DEMO--", filestat ("DEMO")
```

The example prints "**DEMO--rw-**" if the text file DEMO in the current UUT exists and is not write protected.

Remarks:

Filestat returns information about the status of the file. It cannot tell if a floppy disk write-protect tab is in the protected position.

Related Commands:

cwd, open, close

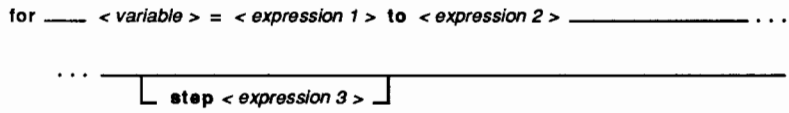
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
for <variable> = <expression 1> to <expression 2>
     [step <expression 3>]
```

Syntax Diagram:



Description:

Executes a series of statements repeatedly for each value of a control variable within a specified range.

Arguments:

variable	Any numeric variable; used as an index.
expression 1	An integer expression for the lowest value in the range.
expression 2	An integer expression for the highest value in the range.
expression 3	An integer expression which specifies the increment after each loop iteration. (Default = 1)

Example:

```
for n = 1 to 5 step 2

   .                    ! if n is 5 or less, perform
   .                    ! the statements within the
   .                    ! "for" block. Since the
   .                    ! "step" variable is 2, the
   .                    ! statements within the
   .                    ! block are executed three
                        ! times.
next
```

Remarks:

The *for . . . next* block repeats the controlled statements for each value of an index variable within a specified range, after which execution continues at the line following the end of the block. The *for . . . next* block ends with a *next* statement.

The optional step expression indicates how much to add to the block control variable after each iteration of the block.

The *for . . . next* block is provided for ANSI compatibility, but the recommended TL/1 structure is the *loop for . . . end loop* block.

Related Commands:

loop, next

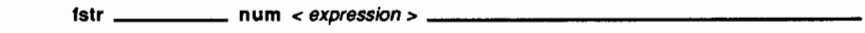
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
fstr num <expression>

fstr (<expression>)
```

Syntax Diagram:



Description:

Produces a string representation of the floating argument value.

Arguments:

expression	A floating-point argument value.

Returns:

The default string representation of the floating argument value.

Examples:

```
s = "The answer is" + fstr (f)
s = fstr num f
```

Remarks:

The returned string is in scientific format, with six digits of precision following the decimal point. For example, the following are strings produced by the *fstr* command:

1.110000E+02
9.900000E+101
-3.240000E-03

Related Commands:

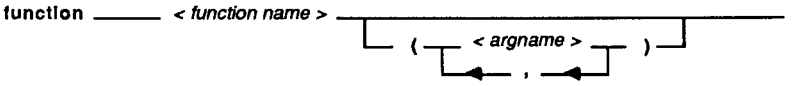
fval

function
statement block

Syntax:

```
function <function name> [(<argname>
     {, <argname>})]
```

Syntax Diagram:



Description:

Specifies the beginning of a function definition block.

Arguments:

function name	Name of the function that is defined in the lines between the *function* and *end* statements.
argname	Name of an argument for this function.

Example:

```
program test4

function max (x,y)

     ! Defines a function called max
     ! Suppose you wrote this function to take two
     ! numeric values as input and return the
     ! greater value (or the second value if they
     ! are equal).
```

(example is continued on the next page)

```
     declare
          numeric x
          numeric y
     end declare
     if (x>y) then
           return x
     else
           return y
     end if
end max
     .
     .
a = max (16, datawidth)   ! calls the function max
     .                    ! defined at the beginning
     .                    ! of the program
end program
```

Remarks:

A function is a sequence of TL/1 statements called by a single name. A function is syntactically identical to a program except that it begins with a *function* statement rather than a *program* statement.

The only difference between the program and function definition block is that the scope of a function name extends only within the block (usually a program block) that encloses it. The scope of a program name is every program in the same UUT and every program in the program library.

The function name used in the *function* statement, the *end* statement, and all invocations of the function must be the same. Function names are case-sensitive; "add" is not the same as "aDd." A function name cannot be the same as the name of a built-in function.

The argument list consists of one or more argument names separated by commas. The list is enclosed in parentheses. The order of the names in this list is the same order in which the values for these arguments must be listed in positional notation calls to this function. If any arguments have default values, these values are assigned in the subsequent declaration blocks.

Function arguments may not be declared as arrays, nor as global or persistent variables.

Function definition blocks may contain any or all of the following: declaration blocks, handler definition blocks, and exerciser definition blocks.

Related Commands:

declare, end, execute, exercise, handle, program, return

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

fval
function

IEEE-488

Syntax:

```
fval str <expression>

fval (<expression>)
```

Syntax Diagram:

Description:

Calculates the floating-point value of the string argument.

Argument:

expression	A valid string expression which represents a floating-point number.

Returns:

The floating-point value obtained by interpreting the string.

Examples:

```
f = fval ("1.0")          ! f is set to the value 1.0

f = fval str "-3E2"       ! f is set to the value
                          ! -300.0
f = fval ("2")            ! f is set to the value 2.0
                          ! Note that a decimal point
                          ! is not necessary here.
```

Remarks:

Any string which can be interpreted as a decimal floating-point number is acceptable. Decimal points are not required. In addition, the first character of the string can be a minus sign, indicating that the number is negative.

Related Commands:

fstr, isflt

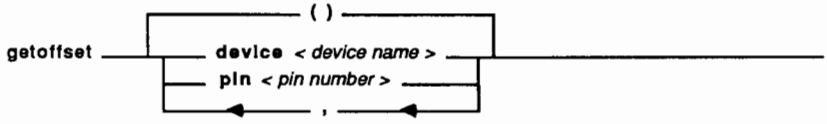
getoffset function

IEEE-488

Syntax:

```
getoffset [device <device name>] [, pin
     <pin number>]

getoffset (<device name>, <pin number>)

getoffset ()
```

Syntax Diagram:



Description:

Returns the current calibration delay offset for the specified device. The value returned is biased by 1000000 (decimal). This value can be changed via calibration, restoring caldata, or using the *setoffset* command. Each sync mode (such as pod addr, pod data, or ext) has a separate offset associated with it. Thus, changing sync modes also changes the offset.

Arguments:

device name	I/O module name or probe name. (Default = "/probe")
pin number	(Default = 1)

Returns:

The current calibration delay offset for the specified device.

Example 1:

```
        ! This example looks at the current offset for
        ! external sync in the probe

sync device "/probe", mode "ext"
offset = getoffset device "/probe"
```

Example 2:

```
        ! This example looks at the current offset for
        ! pod address sync in I/O module 2

sync device "/mod2", mode "pod"
sync device "/pod", mode "addr"
offset = getoffset device "/mod2"
```

Remarks:

The *getoffset* command is valid only when the sync mode is "pod" or "ext".

The value returned is biased by 1000000 (decimal). This means that a returned value of 1000000 represents an offset of 0 nanoseconds, a returned value of 1000020 represents an offset of 20 nanoseconds, and a returned value of 999970 represents an offset of -30 nanoseconds.

The offset value can be changed by calibration, by restoring caldata, or by using the *setoffset* command. In addition, each sync mode (such as pod addr, pod data, or ext) has a separate offset associated with it. Thus, changing sync modes also changes the offset.

The *getoffset* function will not necessarily return exactly the same value that *setoffset* passed. This is because the hardware delay line provides delay increments of about 15 nanoseconds.

Related Commands:

setoffset

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Offset Command" section of the *Programmer's Manual* for information on using offsets with GFI.

getpod function

IEEE-488

Syntax:

```
getpod podname
```

Syntax Diagram:

Description:

Returns information about the current pod.

Arguments:

podname	Return the name of the currently connected pod.

Returns:

The name of the current pod.

Example:

```
if instr ((getpod podname), "M") = 1 then
    print "9132A pod in use"
end if
```

Remarks:

There is a current pod name (called "32BIT") when no pod is plugged in. 9132A pod names all begin with "M", so getpod is useful to determine the type of pod.

Related Commands:

podinfo, podsetup

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

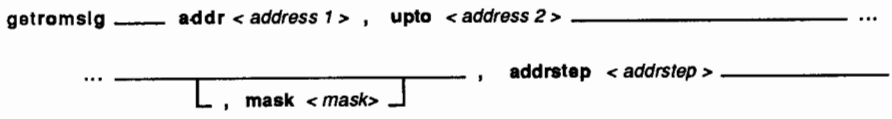
getromsig function

IEEE-488

Syntax:

```
getromsig addr <address 1>, upto <address 2>
     [, mask <mask>], addrstep <addrstep>

getromsig (<address 1>, <address 2>, <mask>,
     addrstep>)
```

Syntax Diagram:



Description:

Returns the signature gathered from one or more ROMs using a mask to select which data bits will be used to form the signature.

Arguments:

address 1	Starting address.
address 2	Ending address.
mask	Bit mask of data bits to be used to form the signature. (Default = $FFFFFFFF)
addrstep	Address increment.

Returns:

The signature measured.

Example 1:

```
measured_sig = getromsig addr 0, upto $7FF,
 addrstep 2
```

Example 2:

```
measured_sig = getromsig (first,last,$7C,4)
```

Remarks:

The ROM signature is a CRC word calculated from the data contained in the ROM. The ROM data is considered to be composed of bit streams consisting of data bit *k* of addresses (*addr, addr + addrstep, addr + 2*addrstep; addr + 3*addrestep,* etc.). These bit streams are concatenated with the most significant set bit in *mask* first, followed by less significant set bits, with the CRC result being taken from the concatenated bit stream. Therefore, if *mask* has only one set bit, the CRC returned by *getromsig* is the same as the signature that would be returned if the probe were placed on the same data bit, and reads were done at addresses (*addr, addr + addrstep, addr + 2*addrstep, addr + 3*addrstep etc.*).

Related Commands:

testromfull

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- *Supplemental Pod Information for 9100A/9105A Users Manual.*
- The Fluke pod manual for the microprocessor you are using.

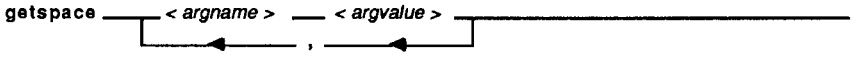
getspace function

IEEE-488

Syntax:

```
getspace <argname> <argvalue> {  <argname>
     <argvalue>}

getspace (<argvalue> {, <argvalue>})
```

Syntax Diagram:



Description:

Converts a list of specified address parameter values to a number suitable as an argument for the *setspace* command. This number depends on the specified parameter values and on the pod which is currently connected.

Arguments:

argname	Address parameter name (refer to the *Supplemental Pod Information for 9100A/9105A Users Manual* for the microprocessor you are using).
argvalue	The address parameter string value you select (refer to *Supplemental Pod Information for 9100A/9105A Users Manual* for the microprocessor you are using).

Returns:

The selected address space as a number.

Example:

```
program test6
     s = getspace space "memory", size "word"
          ! an 80286 address space
     setspace space s
     .
     .
end program
```

Remarks:

Microprocessor address lines usually have multiple addressing modes. These are often referred to as address spaces. For example, in the Z80 microprocessor, two address spaces are available: memory space and I/O space. More powerful microprocessors have a correspondingly larger number of address spaces that can be used. When testing a microprocessor-based UUT, you must select an address space and any other addressing parameters that are needed to test the UUT in the desired address space.

The 9100A/9105A provides a convenient means to select the address space and other addressing parameters. When a pod is connected to the 9100A/9105A and the power is turned on (or the RESET key is pressed), the addressing parameters possible for this pod are made available to the 9100A/9105A in the form of a list of strings representing all legal combinations of addressing options. These options are shown on the operator's display when appropriate commands are selected from the operator's keypad.

These options are also selectable by TL/1 through the use of the *getspace* function used along with the *setspace* command. With the *getspace* function, you specify both a name and a selected string value for each address parameter. No parameters may be left out. The *getspace* function then checks the list of address parameter combinations and returns a number related to the position in the list where the specified combination was found. *The Supplemental Pod Information for 9100A/9105A Users Manual* shows the appropriate parameter names and all legal combinations of parameter values that can be used for each supported microprocessor.

For example, when you refer to *The Supplemental Pod Information for 9100A/9105A Users Manual,* you would find the information on the 80186 microprocessor to write the following getspace command. (A summary of this information for the 80186 microprocessor is also shown in Appendix I of this manual.)

```
s = getspace mode "normal", space "memory", size
 "word"
```

This command would request the number of the address parameter string which has "normal" as the choice for the mode parameter, "memory" as the choice for the space parameter, and "word" as the choice for the size parameter.

The number returned by *getspace* is then used with the *setspace* command to set the pod to operate with the requested addressing parameters. For example, you might use the following command to set the pod to use the number returned by the *getspace* command above:

```
setspace space s
```

Related Commands:

setspace, sysspace

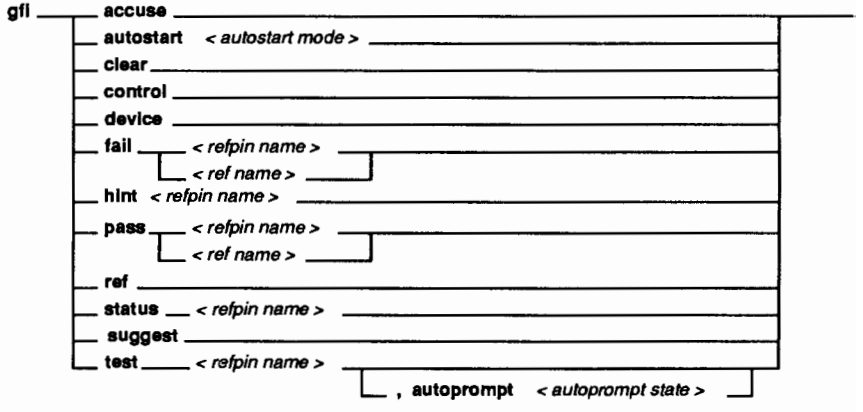
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- Appendix I, "Pod-Related Information," in this manual.
- *The Supplemental Pod Information for 9100A/9105A Users Manual.*
- The Fluke pod manual for the microprocessor you are using.

Syntax:

```
gfi accuse
gfi autostart <autostart mode>
gfi clear
gfi control
gfi device
gfi fail <refpin name>
gfi fail <ref name>
gfi hint <refpin name>
gfi pass <refpin name>
gfi pass <ref name>
gfi ref
gfi status <refpin name>
gfi suggest
gfi test <refpin name> [, autoprompt <autoprompt
     state>]
```

Syntax Diagram:



Description:

The gfi commands allow TL/1 to interact with the resident GFI software. The *gfi status* and *gfi test* commands can be used as the heart of functional testing using the probe and I/O module. Using these commands to perform functional tests has the advantage of using the learned responses (response files) which characterize the known-good UUT rather than including response information in TL/1 functional test programs.

Options:

accuse — (Has no argument value.)

This option returns a string that describes the current GFI accusation, or conclusion ("U1 is BAD or OUTPUT U1-24 is LOADED", for example). If GFI currently has no accusations, an empty string is returned.

autostart — <autostart mode>

This option enables or disables an automatic transition from a TL/1 functional test to GFI.

The autostart mode can be "enable" or "disable". If the mode is "enable", GFI automatically starts after a TL/1 program that generates GFI hints has finished running. If the mode is "disable", a message is displayed informing the operator that GFI hints are available.

clear — (Has no argument value.)

This option erases the GFI summary and GFI suggestion list. It also forces the UUT's GFI database out of memory. An empty string is returned. An error will be reported if a stimulus program attempts to clear GFI. This command should be executed before troubleshooting of a new UUT is begun.

control

(Has no argument value.)

This option determines whether the program is being executed under GFI control. A value of "yes" or "no" is returned.

device

(Has no argument value.)

This option returns a string containing the name of the measurement device ("/probe", "/mod1", "/mod2", "/mod3", "/mod4", etc.) that is being used. Typically, the device list returned is passed to other TL/1 functions which accept device lists and affect the I/O module and probe.

This option is used only in stimulus programs, which are executed under control of the GFI program. If the program is not being executed under GFI control, an error is reported.

fail

<refpin name>

This option is used in a GFI stimulus program to force GFI to fail the specified pin, independent of the actual measured response.

The FAIL applies only to the single program that this statement appears in.

The specified pin must be a pin that is currently being tested by GFI. The name of the current pin or ref being tested can be obtained using the 'gfi ref' option.

<ref name>

If a reference designator name is specified, it will force GFI to fail all the pins on that ref that are tested by the stimulus program.

hint <refpin name>

This option adds the specified pin name to the end of the GFI suggestion list. This option is used in functional tests to identify nodes that are suspected to be faulty.

pass <refpin name>

This option is used in a GFI stimulus program to force GFI to pass the specified pin, independent of the actual measured response.

The PASS applies only to the single program that this statement appears in. (If a pin is tested by several stimulus programs, the cumulative status of the pin can still be BAD if the pin fails one or more of those other programs).

The specified pin must be a pin that is currently being tested by GFI. The name of the current pin or ref being tested can be obtained using the 'gfi ref' option.

<ref name>

If a reference designator name is specified, it will force GFI to pass all the pins on that ref that are tested by the stimulus program.

ref

(Has no argument value.)

This option returns a string containing the name of the reference designator or pin that is being tested by GFI.

This option is used only in stimulus programs, which are executed under control of the GFI program. If the program is not being executed under GFI control, an error is reported.

status

<refpin name>

This options returns a string describing the status of the named pin. It returns: "good" if the pin has been tested and was good, "bad" if the pin has been tested and was bad, and "untested" if the pin has not been tested or no such pin exists.

suggest

(Has no argument value.)

This option returns a string containing the first (highest priority) suggestion on the GFI suggestion list. The string has the form "ref-pin". If the suggestion list is empty, an empty string is returned.

test

<refpin name>, [autoprompt <autoprompt state>]

This option tests the named pin by executing all stimulus programs associated with it. If the named pin is tested with the I/O module, GFI will test all the pins on the component.

The TL/1 "passes" or "fails" condition will be set according to the status of the component (not just the named pin). It will be set to "fails" if any pin on the component is bad. It will be set to "passes" if all the pins on the component are good or untested.

The autoprompt state can be "yes" or "no". (Default = "yes".) If it is "yes", then the necessary operator prompts to clip or probe the component will be automatically generated. If it is "no", the system assumes that the programmer has already prompted the operator. Prompting for the I/O module must be done with the TL/1 *clip* command. This function updates the system connection data for the module, which is used by GFI.

This option cannot be used in a GFI stimulus program.

Example 1:

```
! This program performs the equivalent of RUN GFI
! from the operator's keypad and display.  The only
! difference is that no graphics will be generated
! for the part.

program auto_gfi
     loop while (((gfi accuse) = "") and
      ((gfi suggest) <> ""))
          pin = gfi suggest
          gfi test pin
     end loop
end auto_gfi
```

Example 2:

```
gfi autostart "enable"
gfi hint "U25-15"
     ! enable autostart and add U25-15 to the GFI
     ! suggestion list
```

Example 3:

```
 ! test all pins on U1(if U1 is tested with
 ! the I/O module)
if gfi test "U1-1" passes then
     print "U1 is good"
else
     print "U1 is bad"
     print "U1-1 is ", (gfi status "U1-1")
     print "U1-2 is ", (gfi status "U1-2")
     print "U1-3 is ", (gfi status "U1-3")
          .
          .
end if
```

Example 4:

```
! This stimulus routine wiggles the data pins out
! to the inputs of all components directly
! connected to the microprocessor bus buffer.

program micro_data

               ! If testing with GFI then
               !      get device name from GFI
               ! else not testing with GFI
               !      so specify device name
     if (gfi control) = "yes" then
          devlist = gfi device
     else
          devlist = "/mod1"
     end if
```

```
                ! Setup measurement and
                ! stimulus devices.
        x = getspace space "memory", size "word"
        setspace (x)
                ! Reset device to known state
                ! then configure devices as desired
        reset device devlist
        sync device devlist, mode "pod"
        sync device "/pod", mode "data"
        threshold device devlist, mode "ttl"
        counter device devlist, mode "transition"
        arm device devlist          ! Begin measurement
                              ! Perform stimulus to UUT
             rampdata addr $20000, data 0,mask $1F
             rampdata addr $20000, data 0,mask $1F0
             rampdata addr $20000, data 0,mask $1F00
             rampdata addr $20000, data 0,mask $1F000
        readout device devlist    ! End measurement

end micro_data
```

Remarks:

The *gfi accuse*, *gfi suggest*, and *gfi test* commands can be combined to perform the equivalent RUN GFI from the operator's keypad and display (see example 1). You can also drive an autoprober by modifying example 1; position the autoprober before performing the *gfi test* command.

The *gfi test* and *gfi status* commands can be used as the heart of functional testing. Once the UUT has been divided into sections, functional test programs can be written for each section. These functional tests can use the *gfi test* command to test the components generating output signals for each section.

For example, assume a section has two ICs, U21 and U33, that provide the output signals for that section. In this case, *gfi test* "U21-1" and *gfi test* "U33-1" can be used to test these two components and therefore, the output of the section. (There is an additional assumption that the I/O module is being used to test

U21 and U33. When the I/O module is used, *gfi test* will test all pins on the IC, not just the specified pin.) If one or more of the *gfi test* commands fails, the *gfi status* command can be used to find out which pins on the components failed. The failing pins can be entered in the suggestion list using the *gfi hint* command.

Stimulus programs can be executed from the operator's keypad and display, from the debugger, and from GFI. The *gfi control* command and *gfi device* command are used to identify when GFI is running, and if it is, to get the testing device from GFI (see example 4).

Refer to the *Programmer's Manual* for a description of the GFI program and information on how to create a UUT database for GFI.

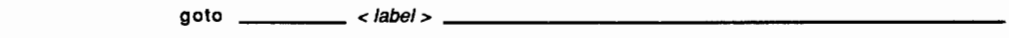
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Guided Fault Isolation (GFI)" section of the *Programmer's Manual.*

Syntax:

```
goto <label>
```

Syntax Diagram:



Description:

Causes program execution to jump to the beginning of the line labeled by <label>.

The *goto* command should be avoided where possible (see the Remarks section for this command).

Arguments:

label	A statement label which exists in the program. A label is always followed by a colon (":").

Example:

```
program test7
      .
      .
      if y <> 24 then goto finish ! skips the lines
                                  ! from here to
      a = a + 1                   ! the label
      b = b + 1                   ! finish unless y
      c = c + 1                   ! is equal to
      finish:                     ! decimal 24
      .
      .
end program
```

Remarks:

Normal execution of statements in a program proceeds in order from one statement to the following statement. But the *goto* command causes program execution to jump to the line with the specified label.

In the preceding example, if y is not equal to 24, the statements between the *if* command and the label "finish" are not executed; the *goto* command transfers program execution to the line beginning with the *finish* label.

No more than one line can be labelled with a particular name. Labels must meet the requirements for variable names as outlined in Section 2 of this manual, "Name Conventions." A colon separates the label from the rest of the line, but is not part of the label name.

The *goto* command should be avoided where possible; it is provided as a last-choice alternative to other control statements. A more orderly and logical flow of instructions can be achieved by using *loop . . . end loop* blocks and *if . . . end if* blocks.

Several restrictions and caveats apply to the *goto* command:

- The label specified by a *goto* command must mark an executable command line somewhere in the current definition block (program, function, handler, or exerciser). A *goto* command cannot jump from one definition block to another or from a program into an enclosed definition block.
- A *goto* command cannot jump to a line contained in a *loop for . . . end loop* block.
- A *goto* command cannot jump out of a *loop for . . . end loop* block.

Related Commands:

if, loop

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

haltuut function

IEEE-488

Syntax:

```
haltuut ()
```

Syntax Diagram:

Description:

Terminates normal runuut operation, if it is active, and displays any fault conditions that occurred during the runuut execution.

Example:

```
haltuut ()
```

Remarks:

After executing *runuut,* you must invoke either *haltuut* or *waituut* to regain control of the pod before executing other statements that send commands to the pod. A *haltuut* command is equivalent to *waituut (0).*

Related Commands:

runuut, waituut

For More Information:

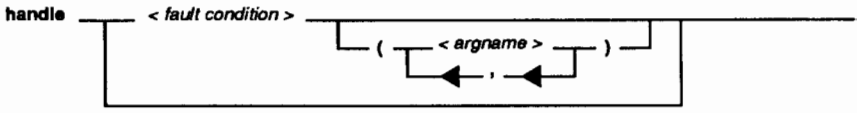
- The "Overview of TL/1" section of the *Programmer's Manual.*

handle
statement block

Syntax:

```
handle <fault condition> [(<argname>
     {, <argname>})]
```

Syntax Diagram:



Description:

Specifies the beginning of a fault condition handler block.

Arguments:

fault condition	Name of the fault condition to be handled. (See Appendix G, "Handling Built-in Fault Messages in TL/1 Programs," in this manual for a list of built-in fault messages.)
argname	Name of an argument for this handler block.

Example:

```
handle my_handler (a,b,c)
          .
          .
          .
end my_handler
          ! or this line could be simply end handle
```

Remarks:

A fault condition handler is a sequence of statements which is executed when a fault condition occurs. Fault condition handlers localize the statements that deal with or respond to the occurrence of a particular fault condition; without handlers, these same statements would require duplication anywhere the fault condition could occur (in some cases, almost anywhere in a program). Handlers are used to acknowledge the presence of fault conditions, inspect data about fault conditions, and make decisions regarding fault conditions. A program or function can provide handlers for fault conditions that may be raised during execution. When a handler exists for a raised fault condition, the normal execution of the program is temporarily suspended and the handler is invoked.

You may write your own handlers in TL/1 for any fault conditions raised by built-in tests or TL/1 *fault* commands.

The fault name must be the same in both the *handle* statement and the *end* statement. A fault handler has the name of the fault condition it is intended to handle.

The argument list consists of argument names separated by commas. If any arguments have default values, these values are assigned in the declarations. The *handle* statement must include all arguments named in the corresponding *fault* command, but it can include additional arguments as well.

Unlike the scope of a variable name, which is static based upon the block structure of a TL/1 program, the scope of a handler is dynamic. The scope of a handler extends from the time at which the containing block is entered until the containing block is exited. If the block invokes another function, the handler remains "active."

Variables declared inside a handler are local unless explicitly declared to be global.

When a fault condition handler encounters either a *return* statement or its last statement the handler terminates, and execution resumes at the statement following the *fault* command. If the handler does not execute a *fault* command, the fault condition is handled and disappears. In this case, the termination status is "passes".

Related Commands:

abort, declare, end, execute, exercise, fails, fault, passes, refault, return

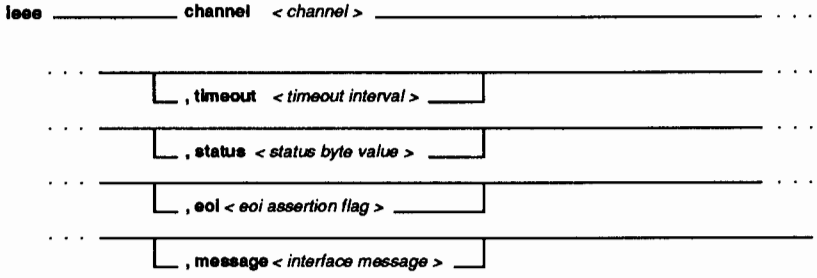
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- Appendix G, "Handling Built-in Fault Messages in TL/1 Programs," in this manual.

Syntax:

```
ieee channel <channel>
     [ , timeout <timeout interval> ]
     [ , status <status byte value> ]
     [ , eoi <eoi assertion flag> ]
     [ , message <interface message> ]
```

Syntax Diagram:



Description:

When the 9100A/9105A is configured as an IEEE-488 talker/listener, the *ieee* command is used to:

- Set/clear the most significant bit of the serial poll status byte.

When the 9100A/9105A is configured as an IEEE-488 controller, the *ieee* command is used to:

- Issue IFC (Interface Clear).
- Send DCL (Device Clear) to all bus devices or SDC (Selected Device Clear) to all devices in a group.
- Assert REN (Remote Enable), optionally addressing a group of devices to listen, putting them in Remote State.
- De-assert REN.
- Send GTL (Go To Local) to a group of devices to put them in Local State.
- Send GET (Group Execute Trigger) to a group of devices to trigger them.

For both configurations of the 9100A/9105A, the *ieee* command is used to:

- set the EOI Enable flag associated with an IEEE-488 channel, controlling EOI assertion on the last output byte of any print command on the channel.
- set the timeout interval associated with a channel.

If an I/O error occurs while attempting to process an *ieee* command (for example, a timeout error occurs), then the io_error fault is raised, with numeric argument err_num containing the error number and string argument err_msg. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error (err_num, err_msg)
  declare numeric err_num
  declare string err_msg
  print using "Error ?#: ?#", err_num, err_msg
end handle
```

Some of the operations described below are susceptible to termination by timeout, while others are not. The former are explicitly noted as being susceptible to timeout. If no mention is made of timeout susceptibility, the operation is not susceptible. When a timeout error occurs, the io_error fault is raised with the err_num and err_msg fault arguments set to indicate a timeout.

Arguments:

channel — A mandatory numeric argument, which must be a channel number returned by the *open* command. The channel will optionally have an IEEE-488 address list associated with it. (See the description of the open function for details.) If so, the channel is said to be open on a group of bus devices. If not, the channel is said to be open on the IEEE-488 interface.

timeout

An optional numeric argument, which specifies the timeout interval to use for any IEEE-488 bus operation on this channel, in milliseconds. A value of zero specifies an infinite timeout interval; i.e., timeouts are disabled.

status

An optional numeric argument, which specifies the value to assign to the serial poll status byte. Only the most significant bit (bit 7) can be set from TL/1. If bit 7 is set/cleared in this argument, bit 7 in the serial poll status byte is set/cleared. All other bits in the argument are ignored.

This argument is applicable only when the 9100A/9105A is configured as a talker/listener. If it is configured as a controller, the status argument is ignored.

eoi

An optional string argument, which specifies whether EOI (End or Identify) assertion on the last output byte of any print command on this channel is enabled (the argument value is "on") or disabled (the argument value is "off"). The argument value is case-insensitive.

Even if last byte EOI assertion is disabled, EOI is still asserted on any termination character associated with the channel when it was opened (see the *open* command, described in this manual for details).

message

An optional string argument, which specfies a special-purpose interface message or operation. This argument is only applicable if the 9100A/9105A is configured as a controller; if it is configured as a talker/listener, use of this argument causes an error.

The message argument takes one of the following values. These are interpreted case-insensitively:

ifc	The IFC line (Interface Clear) is momentarily asserted for at least 100 microseconds.
clear	If the channel is opened on the IEEE-488 interface, UNT (Untalk), UNL (Unlisten), and DCL (Device Clear) are issued. If the channel is opened on a group of devices, UNT and UNL are issued first, then each device in the group is addressed to listen, then SDC (Selected Device Clear) is issued. Both forms of this operation are susceptible to timeout.
local	If the channel is opened on the IEEE-488 interface, REN (Remote Enable) is de-asserted. If the channel is opened on a group of devices, UNT and UNL are issued first, then each device in the group is addressed to listen, then GTL (Go To Local) is issued. Only the latter form of this operation is susceptible to timeout.
remote	First, REN is asserted. If the channel is opened on a group of devices, UNT and UNL are issued and each device in the group is addressed to listen. If the latter step is performed, this operation is susceptible to timeout.

trigger — If the channel is opened on a group of devices, first UNT and UNL are issued, then each device in the group is addressed to listen, then GET (Group Execute Trigger) is issued.

This operation is susceptible to timeout.

If more than one of the optional arguments are supplied, they all take effect in the following order: eoi, timeout, status, and message. If none of the optional arguments are supplied, the *ieee* command does nothing.

The default initial values for a channel are zero for timeout and off for eoi. The default initial value for the serial poll status byte is the current value of the byte; that is, changes to the serial poll status byte survive the execution of a TL/1 program.

Example 1:

These examples illustrate operation as a talker/listener.

The following TL/1 fragment sets, then clears, bit 7 in the serial poll status byte:

```
ieee_chan = open device "/ieee"
ieee channel ieee_chan, status $80
ieee channel ieee_chan, status 0
```

The following TL/1 fragment opens the IEEE-488 interface with the termination character set to "nothing", then prints a single record to the interface with three separate *print using* commands. EOI assertion is disabled for the first two *print using* commands, and turned on for the last one; as a result, EOI is asserted on the last byte of the last *print using* command.

```
ieee_chan = open device "/ieee", term ""
ieee channel ieee_chan, eoi "off"
print on ieee_chan, using "?#", v1
print on ieee_chan, using "?#", v2
ieee channel ieee_chan, eoi "on"
print on ieee_chan, using "?#", v3
```

The following TL/1 fragment opens the IEEE-488 interface with the termination character set to linefeed, then perpetually inputs a single line from the interface and echoes it back:

```
declare string s
ieee_chan = open device "/ieee", term "\0A"
loop
      input on ieee_chan, s
      print on ieee_chan, s
end loop
```

Example 2:

These examples illustrate operation as a controller.

The following TL/1 fragment momentarily asserts IFC for at least 100 microseconds:

```
ieee_chan = open device "/ieee"
ieee channel ieee_chan, message "ifc"
```

The following TL/1 fragments illustrate the two uses of the "clear" message. The first one sends DCL to all devices on the bus, while the second one sends SDC to a particular group of devices:

```
! send DCL
ieee_chan = open device "/ieee"
ieee channel ieee_chan, message "clear"

! send SDC to addresses 2, 3, and 4:10
! (primary address 4, secondary address 10)
ieee_chan = open device "/ieee/2,3,4:10"
ieee channel ieee_chan, message "clear"
```

The following TL/1 fragment asserts REN:

```
ieee_chan = open device "/ieee"
ieee channel ieee_chan, message "remote"
```

The following TL/1 function sends a hypothetical voltmeter a command requesting a measurement, inputs the measurement, strips the measurement of whitespace, verifies that the measurement string represents a floating-point value, and returns

the value. If any I/O errors which result in the io_error fault occur, they are handled locally.

Note that with this fragment a global variable is used to flag whether I/O failed. In an application that uses this function, the caller would examine this flag before attempting to use the return value from get_rdg for anything. Also, note that the io_error handler doesn't do much about the I/O error. In a real application, the I/O error handling would be more sophisticated, and have provision for retries, clearing the IEEE-488 bus, etc.

```
function get_rdg

   handle io_error (err_num, err_msg)
       declare
       numeric err_num   ! I/O error number
       string err_msg    ! I/O error message

       global numeric term_chan   ! terminal channel
       global numeric io_failed   ! I/O failed flag
       end declare
     io_failed=1
     print on term_chan, using "Error ?#: ?#",
     err_num, err_msg
   end handle

   declare
       global numeric io_failed
       global numeric term_chan
       numeric dvm_chan
       string dvm_meas_string
   end declare

   io_failed = 0
   ! clear the "I/O failed" flag
   term_chan = open device "/term1"
   ! open the error message channel

   ! open a channel to the DVM, which has IEEE-488
   ! address 1,
   ! using a linefeed character as the terminator
   dvm_chan = open device "/ieee/1", term "\0A"
   if (io_failed) then return (0.0)
```

```
        ! set the timeout on the DVM channel to 4
        ! seconds
        ieee channel dvm_chan, timeout 4000
        if (io_failed) then return (0.0)

        ! send the reading request to the DVM
        print on dvm_chan, "rdg?"
        if (io_failed) then return (0.0)

        ! input the measurement string
        input on dvm_chan, dvm_meas string
        if (io_failed) then return (0.0)

        ! this hypothetical DVM terminates readings with a
        ! carriage return, linefeed combination.  The
        ! linefeed will have been removed, since it was
        ! specified as the termination character; however,
        ! dvm_meas_string will still have the carriage
        ! return. Strip it and any other whitespace via
        ! the 'token' function.
        dvm_meas_string = token str dvm_meas_string, from 1
        ! verify that the measurement string is a
        ! floating-point number
        if (isflt (dvm_meas_string)) then
          return(fval(dvm_meas_string))
        else
          io_failed = 1
          return (0.0)
        end if

end function
```

Related Commands:

open, print, input, poll

For More Information:

- The "IEEE-488, 9100A-015" section of the *Technical User's Manual*".
- The "9100 Series Error Numbers" appendix in this manual.

Syntax:

```
if <condition> then
```

Syntax Diagram:

Description:

Specifies the beginning of an if block.

Arguments:

condition	A logical expression.

Example 1:

```
     ! if the variable "initialized" has the
     ! value 0, then...

if not initialized then
     i = 0
     start = 0
     stop = 255
     incr = address_increment
end if
```

Example 2:

```
if b <> 0 then
     if a/b > 10 then        ! bigtest is executed
          execute bigtest    ! only if both b <> 0
                             ! and a/b > 10
     end if
end if
```

Example 3:

```
      ! if the value of the variable "lower" is
      ! less than the value of the variable
      ! "upper", then perform the program/
      ! function "rangetest", using "lower" and
      ! "upper" as arguments.

if ( lower <= upper ) then
     range_test (lower , upper)

     ! else perform the program/function
     ! "range_error" using "lower" and "upper"
     ! as arguments.

else
     range_error (lower , upper)
end if
```

Example 4:

```
if lbit < 4 then
     b = 0
else if lbit = 4 then
     b = 1
else if lbit = 5 then
     b = 2
else                ! (if lbit > 5 then)
     b = 3
end if
```

Remarks:

An if block executes a list of statement lines if a condition in the *if* statement is true (non-zero). If the condition is true, execution begins on the line immediately following the *if* statement. When the controlled statements have been completed, execution continues at the line following the *end if* statement.

If the condition is false, the condition of the first *else if* statement (if one exists) is evaluated; if the condition is true, the statements controlled by the *else if* block are executed, then execution continues at the line after the *end if* statement. If the condition of the *else if* statement is false, the condition of each following *else if* statement is evaluated in the same way.

If the condition of the *if* statement and all of the *else if* statements are false, the statements controlled by the *else* block (if one exists) are executed. If no *else* block exists, execution continues at the statement following the *end if*.

NOTE

You may nest if statements within other if statements. TL/1 does not limit the number of if levels used.

Related Commands:

end, if (statement form)

For More Information:

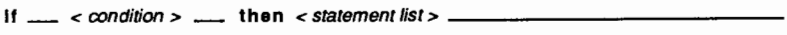
- The "Conditional Expressions" section in Section 2 of this manual.
- The "Overview of TL/1" section of the *Programmer's Manual*.

if (statement form)
statement

Syntax:

```
if <condition> then <statement list>
```

Syntax Diagram:



Description:

Executes statement(s) under control of a condition.

Arguments:

condition	A logical expression.
statement list	One or more TL/1 statements, separated by backslashes (\), all appearing on one line.

Example 1:

```
if x = 10 then a = a + 1
```

Example 2:

```
if x = 10 then a = a + 1 \ b = b + 1
```

Example 3:

```
if x = 10 then a = a + 1 \ if a = 11 then b = 3
```

Remarks:

A single-line *if* statement controls only the statements which appear on a single line. The statements on the line are executed if the condition is true; after the controlled statements have been executed, execution continues on the following line. If the condition is false, the controlled statements are not executed.

An *if* statement may appear within another *if* statement. In this case, execution continues throughout the entire statement line, until either the last statement on the line is executed, or one of the conditions evaluates to false.

You may structure conditions so that under certain circumstances they are not fully evaluated. For example, in the statement:

```
if (b <> 0) and (a/b > 10) then x=1
```

the second condition must not be evaluated when b is equal to zero, because it would cause a divide-by-zero error. To avoid the potential error, you write:

```
if (b <> 0) then if (a/b > 10) then x=1
```

The first *if* statement controls all statements to its right. If the first condition is false, the second condition is not evaluated.

Related commands:

if (block form)

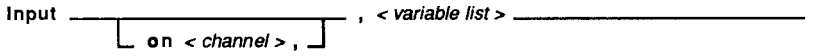
For More Information:

- The "Conditional Expressions" section in Section 2 of this manual.
- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
input [on <channel>,] <variable list>
```

Syntax Diagram:



Description:

Reads data from the text file, serial port, keyboard, or other interface associated with a specified channel, and stores it into specified variables. The *input* command waits for characters to be input.

Arguments:

channel	A numeric expression that identifies a device open for input. (Default = the first channel opened for "input" or "update")
variable list	List of variables in which to store input data values.

Example 1:

```
input on ichan, a, b, c
     ! reads in three values from device on ichan
     ! when entered, these values must be separated
     ! by space characters
```

Example 2:

```
input startaddr, endaddr, addrincr
      ! reads in three values from
      ! operator's keypad
      ! provided a channel has been opened
```

Remarks:

Before using the *input* statement, the input variables must be declared and the input device must be opened for input. The *input* statement performs the following actions:

- Suspends program execution until data appears on the specified channel.
- Reads data from the specified channel.
- Stores the data in the specified variables.

If the channel is buffered, the *input* statement reads data from the channel until it encounters the channel's termination character, which usually defaults to a new-line character (0D hexadecimal on most keyboards, the ENTER key on the operator's keypad). If the channel is not buffered, the *input* statement suspends execution until it reads a single character (any key code).

Space characters must be used to separate multiple input values entered with an *input* command.

To input hexadecimal numeric data, use the *input using* command.

If an I/O error occurs while attempting to process an *input* command (for example, a timeout error while attempting to input from an IEEE-488 device), the 'io_error' fault is raised, with numeric argument 'err_num" containing the 9100A/9105A error number and string argument 'err_msg' containing a description of the error. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error (err_num, err_msg)
     declare numeric err_num
     declare string err_msg
     print using "Error ?#: ?#", err_num, err_msg
end handle
```

If the input is obtained from a channel opened on either the IEEE-488 interface or a device connected to the IEEE-488 interface, then buffered input can be terminated by either the termination character associated with the channel or by the assertion of EOI on the last input character. If input is terminated by EOI assertion and the associated character is not the termination character, then that character is included in the input value.

Related Commands:

input using, open, print, poll, ieee

For More Information:

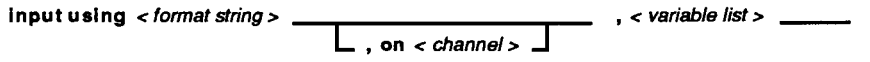
- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Operator's Keypad Mapping to TL/1 Input," appendix in this manual.
- The "Programmer's Keyboard Mapping to TL/1 Input," appendix in this manual.
- The"9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488, 9100A-015" section of the *Technical User's Manual*".

input using statement

Syntax:

```
input using <format string> [, on <channel>],
     <variable list>
```

Syntax Diagram:



Description:

Reads data from a specified channel, compares it with an expected format, and stores it in specified variables.

Arguments:

format string	The string that defines the input format. ("buffered" mode only.)
channel	A numeric expression that identifies a device open for input. (Default = the first channel opened for "input" or "update".)
variable list	List of variables in which to store input data values.

Example 1:

```
input using "TITLE: ####### WR.PROT: ###",t,wp

     ! the input data:
     !    TITLE: prog 1  WR.PROT: YES
     ! produces the values:
     !    t = "prog 1 ", wp = "YES"
```

Example 2:

```
input using "& @ % #", a, b, c, d
     ! The input data:
     !    1 2 3 4
     ! produces the values:
     !    a = 1, b = 2, c = 3, d = 4
     ! The following input data produces an error:
     !    4 3 2 1
     ! because "4" is not matched by the "&"
     ! picture, which expects only "1" or "0"
```

Example 3:

```
input using ":##%##%##%##%##%##%##%", addr,a,b,
c,d,e,f,g,h,chksum

     ! The input data:
     !    :10000001020304050607 8F
     ! produces the values
     !    addr = 1000, a = 0, b = 1, c = 2,
     !    d = 3, e = 4, f = 5, g = 6, h = 7,
     !    chksum = 8F
```

Example 4:

```
input using "?& ?@ ?% ?# ?^", a, b, c, d, f

     ! The input data:
     !    0101 100 100 32 1.97
     ! produces the decimal values
     !    a = 5, b = 100, c = 256
     !    d = 32, f = (floating) 1.97
     ! The input data:
     !    011 25 11 9999 2.0E30
     ! produces the decimal values
     !    a = 3, b = 25, c = 17,
     !    d = 9999, f = (floating) 2.0E+30
```

Example 5:

```
input using "##^^^ ###^^EEEEE", f1, f2

     ! The input data:
     !   3.2  32.3E+100
     ! produces the floating-point values
     ! f1 = 3.2, f2 = 3.23E+101
```

Remarks:

The *input using* statement requires that the input variables be previously declared and that the input device has already opened for input. The *input using* statement performs the following actions:

- Suspends program execution until data appears on the specified channel.
- Reads data from the specified channel.
- Stores the data in the specified variables.

If an I/O error occurs while attempting to process an *input using* command (for example, a timeout error while attempting to input from an IEEE-488 device), then the io_error fault is raised, with numeric argument err_num containing the 9100A/9105A error number and string argument err_msg containing a description of the error. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error(err_num, err_msg)
  declare numeric err_num
  declare string err_msg
  print using "Error ?#: ?#", err_num, err_msg
end handle
```

If the input is obtained from a channel opened on either the IEEE-488 interface or a device connected to the IEEE-488 interface, buffered input can be terminated by either the termination character associated with the channel or by the assertion of EOI on the last input character. If input is terminated by EOI assertion and the associated character is not the termination character, then that character is included in the input value.

The *input using* statement is an extended form of the *input* statement which uses format specifications which are provided in format strings. Through format specifications, you can set the expected number of characters of digits for each data value, and the radix (hexadecimal, decimal, or binary) for numeric data. You can also specify other data that must appear in the input but is not actually required for storage into variables. An error occurs when the number of input values does not match the number of format pictures in the format specification.

A format string contains zero or more format pictures. A format picture is a string that describes the format for a single data value. Format pictures are one of two types: fixed-width and variable-width. Fixed-width format pictures match each input character with a format picture character, while variable-width format pictures match as many input characters as possible in the context of the data type for the format picture.

Fixed-width Formatted Input

A fixed-width format picture consists of zero or more "#" characters (floating digit places), followed by one or more "%", "@", "&", "#", or "^" characters (fixed digit places,) which also describe the radix. The format picture characters are defined below:

Symbol	*Description*
#	A floating digit or character place. If this symbol is in the right-most place in a numeric input format picture, the input is interpreted in decimal radix. Otherwise, this symbol is interpreted as a floating digit, matching digit, or a space. For string input, this symbol matches with any character.
%	Required digit place for hexadecimal numbers. This symbol, and any "#" symbols to its left matches with a hexadecimal digit.
@	Required digit place for decimal numbers. This symbol, and any "#" symbols to its left matches with a decimal digit.
&	Required digit place for binary numbers. This character and any "#" symbols to its left matches with a binary digit.
^	Required digit place for floating-point numbers. A sequence of "^" characters may optionally be followed by a fixed sequence of five "E" characters. For output format pictures, this represents the exponent field; here, it is provided merely for symmetry with output format pictures.

If the value is to contain more than one digit or character, the picture must be extended to the left by "#" characters, one for each additional digit or character. For example, the picture "@" represents a single-digit decimal number; the picture "###@" represents a four-digit decimal number. The picture "#######" represents a seven-character string or a seven-digit decimal number. The picture "%%%%" represents a four-digit hexadecimal number with required leading zeros (00FF).

If numbers in the input string are not separated by non-numeric characters, one of the required digit characters must appear as the last character of the format picture. The two format pictures, "####" and "###@", when combined, yield a format picture, "########@", which is interpreted as an eight-digit decimal picture, not two four-digit pictures. If two pictures are needed, use a space ("#### ####"), use the fixed digit characters ("###%###%"), or use two consecutive *input* statements.

Fixed-width input format checking for floating-point numbers is less strict then it is for other data types. The number of input characters implied by the width of the format pictures are collected and checked to make sure that they represent a valid string representation of a floating-point number.

Variable-Width Formatted Input

Like a fixed-width format picture, a variable-width format picture describes the format for a single data value. The difference is that a variable-width format picture accepts as many input characters as it can use to match the picture's data type. A variable-width format picture consists of a single "?" character, followed by a single character denoting the picture type. The following special picture type characters are defined:

#	Match a variable-width string or decimal number.
%	Match a variable-width hexadecimal number.
@	Match a variable-width decimal number.

& Match a variable-width binary number.

^ Match a variable-width floating-point number

? Match a single "?" character.

If the picture type character is not one of the above, then the format picture means "match one or more of the literal character". For example, the input picture"?X" will match one or more "X" characters in the input. For variable-width literal character matches, no assignment to input variables is performed.

Note that it is not possible to match a variable number of "?" characters; that is, the picture "??" implies a match of exactly one "?" character.

Related Commands:

input, open, print using

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- Appendix J "9100A/9105A Error Numbers".
- The"9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488, 9100A-015" section of the *Technical User's Manual*".

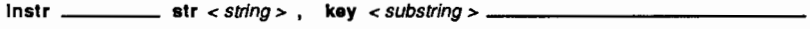
instr function

IEEE-488

Syntax:

```
instr (<string>, <substring>)

instr str <string>, key <substring>
```

Syntax Diagram:



Description:

Returns the index number at which a sub-string appears in a string; returns zero if the sub-string does not appear in the string.

Arguments:

string	String to be searched.
substring	Sub-string to be searched for.

Returns:

The index number if the substring is found; zero if the sub-string is not found.

Example:

```
x = instr ("enter data and address", "data")
     ! the variable x is set to 7.
```

Remarks:

An index number is the number corresponding to a particular character position within a character string. For example, the index of "r" in "America" is 4.

isflt function

IEEE-488

Syntax:

```
isflt str <expression>

isflt (<expression>)
```

Syntax Diagram:

Description:

The *isflt* command is used to pre-test an expression for validity as an argument to the *fval* command.

Arguments:

expression	The string expression that is to be tested.

Returns:

1 if the expression is a valid argument to *fval*, implying that *fval* will not report an error with the expression as its argument

0 if the expression is not a valid argument to *fval*

Examples:

```
isflt ("foo")          ! returns 0
isflt ("1.0")          ! returns 1
```

Remarks:

The *isflt* command is particularly useful for strings obtained by the *input* command, which may be expected to be strings representing valid floating-point numbers, but are not guaranteed to be.

Related Commands:

fval, isval, token

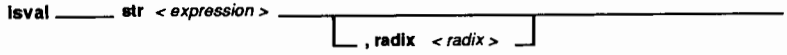
isval
function

IEEE-488

Syntax:

```
isval str <expression> [ , radix <radix> ]
isval (<expression>, <radix>)
```

Syntax Diagram:



Description:

The *isval* command is used to pre-test a set of arguments for validity as arguments to the *val* command.

Arguments:

expression	The string expression which is to be tested for representation of a valid number in the indicated radix.
radix	A numeric expression for the radix of the interpreted number. Allowable values for radix are 2, 8, 10 (default), and 16.

Returns:

1	if the arguments are a valid set to *val* (implying that *val* will not report an error with the given argument set).
0	if the arguments are not a valid argument set to *val*.

Remarks:

The *isval* command is useful for strings obtained by the *input* command, which may be expected to be strings representing valid numbers, but are not guaranteed to be.

Examples:

```
isval ("foo")          ! returns 0
isval ("1", 2)         ! returns 1
isval ("3", 2)         ! returns 0
isval ("3", 10)        ! returns 1
```

Related Commands:

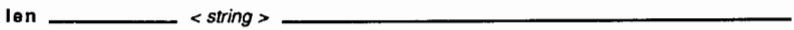
val, isflt, token

len
operator

Syntax:

```
len <string>
```

Syntax Diagram:



Description:

Counts the number of characters in the string operand.

Arguments:

string	String or string expression whose length is to be determined.

Returns:

The number of characters in the string.

Example 1:

```
x = len "I/O finished"          ! the variable x is
                                ! set to C (hex)
```

Example 2:

```
a = "Hello"
x = len a                       ! the variable x is
                                ! set to 5
```

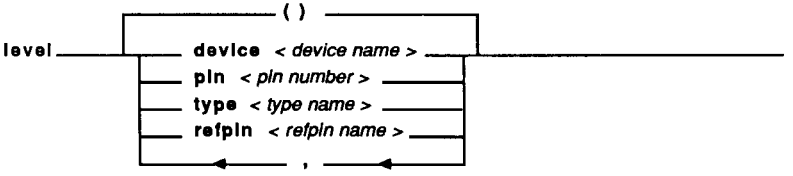
level function

IEEE-488

Syntax:

```
level [device <device name>] [, pin <pin number>]
     [, type <type name>] [, refpin <refpin name>]

level (<device name>, <pin number>, <type name>,
     <refpin name>)

level ()
```

Syntax Diagram:



Description:

Returns the synchronous level history or asynchronous level history for one pin. The data can be requested either in terms of an I/O module pin, a component pin, or the probe. This command will return useful information only after an *arm . . . readout* block has taken a measurement.

Arguments:

device name	I/O module name, clip module name, probe name, or reference designator. (Default = "/probe")
pin number	Pin number. (Default = 1)
type name	"clocked" or "async". (Default = "clocked")

refpin name	Specify the device and pin in string format. The refpin argument is used to override the device and pin values. (Default = "")

Returns:

A number that represents the level history:

0 - No levels clocked.
1 - Low.
2 - Invalid.
3 - Invalid, low.
4 - High.
5 - High, low.
6 - High, invalid.
7 - High, invalid, low.

Example 1:

```
mod = clip ref "u3", pins 40

arm device mod
     execute stim_prog
     loop while ((checkstatus(mod) <> $F))
     end loop
readout device mod

modlevel = level device "u3", pin 3, type
 "clocked"
```

Example 2:

```
arm device "/probe"
     .
     .
     .
readout device "/probe"
probelevel = level device "/probe", type "clocked"
```

Remarks:

The *level* function returns the synchronous level history or asynchronous level history for one pin. The data can be requested either in terms of an I/O module pin or a component pin.

The level can be requested for a specific pin of an I/O module by specifying the module name ("/mod1", "/mod2", etc.) as the device argument. The pin argument is interpreted as an I/O module pin. Refer to Appendix E for tables that show what I/O module pin numbers to use for every possible clip module.

If a component name ("U1", "U2", etc.) is specified as the device argument, the pin argument is interpreted as a component pin. The *level* function determines the I/O module and pin number that corresponds to the specified component pin. The named component must have been previously named in a *clip* command.

If the string value for refpin is not a null string (""), the values of the device and pin arguments are ignored.

The *level* function should be called only after the execution of an *arm . . . readout* block.

Related Commands:

arm, count, readout, sig

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

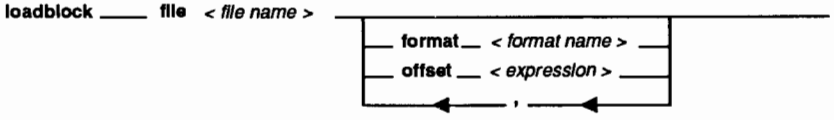
loadblock function

IEEE-488

Syntax:

```
loadblock file <file name> [, format <format name>,
     offset <exp>]

loadblock (<file name>, <format name> , <offset
     expression>)
```

Syntax Diagram:



Description:

Loads the contents of a file in a standard ASCII form (Motorola S-Record format or Intel Hex format) into UUT or pod overlay RAM. The file contains information about the starting address and number of data bytes.

The value of the offset expression is interpreted as a 32-bit 2's complement value and added to the address in each record of the data file, to obtain the load address.

Arguments:

file name	The name of the file containing the required data.
format name	The ASCII format in which the data was previously stored. Either "intel" or "motorola". (Default = "motorola".)
offset	This value is added to each address in the data file to obtain the load address. (Default = 0.)

Examples:

```
loadblock file "test_lcd", format "motorola"

loadblock ("pgm1", "motorola", $10000)
```

Related Commands:

readblock, writeblock

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

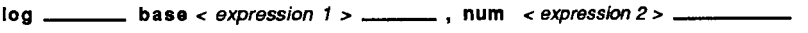
log function

IEEE-488

Syntax:

```
log base <expression 1>, num <expression 2>

log (<expression 1>, <expression 2>)
```

Syntax Diagram:



Description:

Computes the logarithm of the floating-point number argument value in the base specified by the floating-point base argument value.

Arguments:

expression 1	The floating-point value to use as the base while computing the logarithm.
expression 2	The principal argument (the floating-point value) to the logarithm computation.

Returns:

The floating-point logarithm of expression 2 in the specified base (expression 1).

Examples:

```
f = log base 10.0, num 100.0  ! result is 2.0
f = log (2.0, 64.0)           ! result is 6.0
```

Remarks:

If any of the following is true, an error will result:

- the base argument value is less than or equal to 0.0.
- the num argument value is less than or equal to 0.0.
- the base argument value is equal to 1.0.

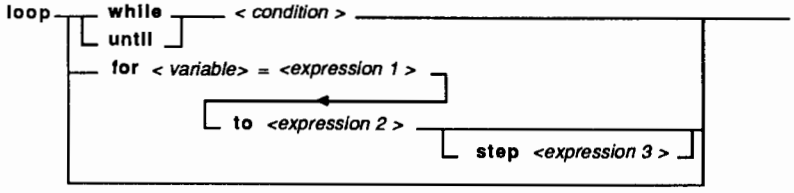
Related Commands:

pow, natural

Syntax:

```
loop while <condition>

loop until <condition>

loop for <variable> = <expression 1> to
     <expression 2> [step <expression 3>]

loop
```

Syntax Diagram:



Description:

Specifies the beginning of a *loop* block.

Arguments:

condition	A logical expression which controls loop termination.
variable	A variable; used as an index.
expression 1	An integer expression for the lowest value in the range.
expression 2	An integer expression for the highest value in the range.

expression 3	An integer expression which specifies the increment after each loop iteration. (Default = 1)

Example 1:

```
     ! Establish a loop using the variable "a" for
     ! control.  The statement within the loop
     ! block, in this example "a = read (porta)" is
     ! repeated until the variable "a" has a non-
     ! zero value.  In the "a = read (porta)"
     ! statement, "a" is set to the value of the
     ! data read from the address specified in the
     ! variable "porta".

a = 0
loop while a = 0
     a = read (porta)
end loop
```

Example 2:

```
b = 0                        ! "b" is set to zero.
loop until (a = $FF)          ! "a" is set to data read
     a = read (b)            ! from address "b".
     b = b + 1               ! b is incremented by 1.
end loop                     ! the loop repeats until
                             ! "a" (which is read
                             ! from b) obtains the
                             ! value FF.
```

Example 3:

```
loop for i = 1 to 5 step 2

    .                ! establish loop control with
    .                ! variable i. Each time that
    .                ! end loop is reached, the
    .                ! variable i is incremented by
    .                ! 2. This process continues
    .                ! until i is greater than 5.
                     ! Then, loop terminates.
end loop
```

Remarks:

The *loop* block executes a list of statement lines repeatedly under control of a condition. The condition controls one of the following:

- Continuation.
- Termination.
- Number of iterations.

The *loop . . . while* block repeats the controlled statements as long as a condition is true. The condition is evaluated first. If it is true, the block is executed and the condition is evaluated again. If the condition is false at any time it is evaluated, execution continues with the statement on the line following the *end loop* statement.

The *loop . . . until* block repeats the controlled statements until an exit condition becomes true. The condition is evaluated first. If it is false, the block is executed, and the condition is evaluated again. If the condition is true at any time it is evaluated, execution continues with the statement on the line following the *end loop* statement.

The *loop . . . for* block repeats the controlled statements for each value of an index variable within a specified range, after which

execution continues at the line following the *end loop* statement. The index variable must be a numeric (integer).

The step expression is an optional segment of the *loop . . . for* block which indicates how much to add to the loop control variable after each iteration of the loop.

The value of the index variable is undefined outside of the block.

A *loop* block with no termination condition repeats the controlled statements indefinitely.

Related Commands:

end

For More Information:

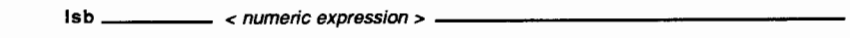
- The "Overview of TL/1" section of the *Programmer's Manual.*

lsb
operator

Syntax:

```
lsb <numeric expression>
```

Syntax Diagram:



Description:

Returns the index of the least-significant set bit in the operand.

Arguments:

numeric expression	Operand from which to determine the least-significant bit.

Returns:

An index number which ranges from 0 to 31 (where an index of 0 corresponds to the least-significant bit).

Example:

```
v = $FA      ! sets the variable v to hexadecimal FA
x = lsb v    ! the variable x is set to 1
```

Remarks:

A numeric expression that evaluates to zero causes an error since no bits are set.

Related Commands:

msb

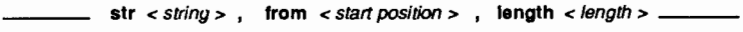
mid function

IEEE-488

Syntax:

```
mid str <string>, from <start position>,
     length <length>

mid (<string>, <start position>, <length>)
```

Syntax Diagram:



Description:

Extracts a new string of the specified length from the given string beginning with the character at the specified index. The sum of the start position and the length number cannot exceed the number of characters in the string plus 1.

Arguments:

string	String or string expression from which to perform extraction.
start position	Integer expression. The left-most character of the string is position number 1.
length	Integer expression.

Returns:

The extracted string.

Example:

```
x = mid str "data 56", from 6, length 2
                          ! the variable x is set
                          ! to "56"
```

Syntax:

```
msb <numeric expression>
```

Syntax Diagram:

Description:

Returns the index of the most-significant set bit in the operand.

Arguments:

numeric expression	Operand from which to determine the most-significant set bit.

Returns:

An index number which rages from 0 to 31 (where an index of 0 corresponds to the least-significant bit).

Example:

```
x = msb $A        ! the variable x is set to 3
```

Remarks:

A numeric expression that evaluates to zero causes an error since no bits are set.

Related Commands:

lsb

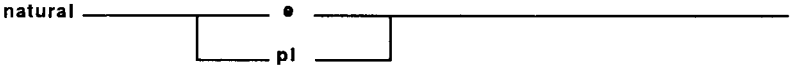
natural
special function

IEEE-488

Syntax:

```
natural e

natural pi
```

Syntax Diagram:



Description:

Returns the floating-point value of a selected natural constant.

Argument:

The *natural* function takes a single symbolic argument, specifying which natural constant is desired. This argument is actually an argument name that does not take an associated value. This means you will usually need to surround the natural command with parentheses to avoid accidentally associating a value with it, as demonstrated in the examples below.

Choose one of the arguments:

e — Specifies the value of e (for use with the log function to compute natural logarithms).

pi — Specifies the value of pi.

Examples:

```
theta = natural pi           ! theta is set to pi
theta = (natural pi)/2.0     ! theta is set to pi/2
f = log base (natural e),num 3.0
      ! f is set to natural logarithm of 3.0
```

Remarks:

The *natural* function allows access to useful natural constants to maximum precision, without having to look them up and type them in by hand.

Related Commands:

log, sin, cos, tan

Syntax:

```
next
```

Syntax Diagram:

Description:

Terminates a *for . . . next* block.

Example:

```
      .
      .
for k = 1 to 100    ! begins a for ... next block
      .
      .
next                ! ends a for ... next block
```

Remarks:

The *for . . . next* block is provided for ANSI compatibility, but the recommended TL/1 structure is the loop *for . . . end loop* block.

Related Commands:

end, for, loop

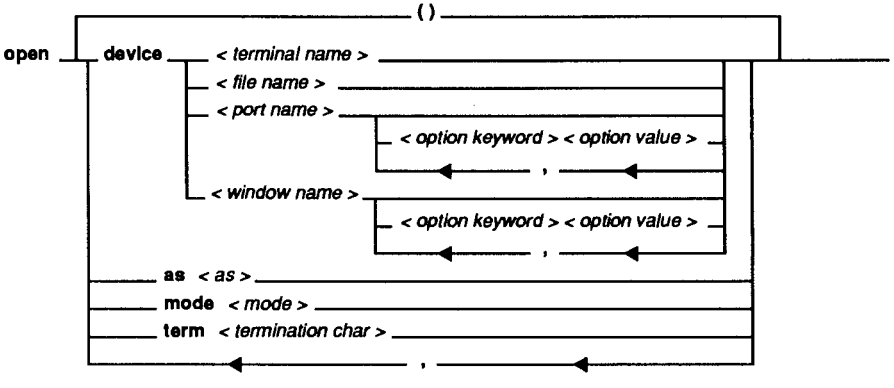
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
open device <terminal name> [, as <as>]
     [,mode <mode>] [,term <termination char>]

open device <file name> [, as <as>]
     [, mode <mode>] [,term <termination char>]

open device <port name> [{, <option keyword>
     <option value>}] [, as <as>] [, mode <mode>]
     [,term <termination char>]

open device <window name> [{, <option keyword>
     <option  value>}] [, as <as>] [, mode
     <mode>] [,term <termination char>]

open device <ieee-488 name> [, as <as>]
     [,mode <mode>] [,term <termination char>]

open ()
```

Syntax Diagram:



Description:

Connects the program to a serial port, terminal, text file, or other interface for input or output by allocating the device to the program and returning an I/O channel number. It also sets parameters for serial ports and windows.

Arguments:

device — Terminal name:

"/term1"	for the operator's interface.
"/term2"	for the programmer's interface.

File name:

The name of any text file enclosed in quotes.

Port name:

"/port1"	for RS-232 port1
"/port2"	for RS-232 port2

Window name:

"/term1/win"	for operator's display.
"/win" (or "term2/win")	for monitor.

(Default = "/term1")

IEEE-488 interface or device name:

"/ieee"	for the IEEE_488 interface.
"/ieee/address list"	for one or more devices attached to the IEEE-488 interface.

where address list is a list of comma-separated IEEE-488 addresses. Each address is either a single radix 10 number indicating the device address or a pair of numbers separated by a colon character, indicating the primary and secondary addresses of the device. For example:

"/ieee/1"	for the device at address 1.
"/ieee/2,4:10"	for the group consisting of the device at address 2 and the device with primary address 4 and secondary address 10.

as

"output", "update", "input", or "append". (Note: "update" is not allowed for file names.)

Default for terminal names: "update"
Default for file names: "output"
Default for port names: "update"
Default for window names: "update"
Default for IEEE-488: "update"

mode

"buffered" or "unbuffered".
(Default = "buffered")

The mode argument determines how newline characters are translated by TL/1 *print*, *print using*, *input*, or *input using* commands for serial ports, and how string values are input.

term — The termination character string argument specifies the termination character to associate with the channel. The length of the string must be less than or equal to one character. If the length is zero, no termination character is associated with the channel. The termination character is used to mark the end of input records and is appended to printed expressions.

If the term argument is not supplied, the termination character default is the newline character. An exception to this is an IEEE-488 channel, where the default is the linefeed character.

The following option keywords apply only if the device is a window:

Option Keyword	*Option Value(s)*
xorg, yorg	Numeric expressions for the location of the upper left-hand corner of the window in characters. Default values: xorg = 0 yorg = 0
xdim, ydim	Numeric expressions for the size of the window in characters. Default values for operator's display: xdim = 42 ydim = 3 Default values for monitor: xdim = 80 ydim = 24

xscale, yscale — Numeric expressions for the full-scale coordinates for objects to be displayed by a window. The maximum value for xscale and for yscale is 10000.

Default values:
xscale = 1000
yscale = 1000

border — A string expression for the title to be centered at the top of the border. If the string expression is a null string (""), the border will not have a title.
(Default: no border)

The following option keywords apply only if the device is an RS-232-C serial port:

Option Keyword	*Option Value(s)*
speed	Baud rate: 19200, 9600, 4800, 2400, 2000, 1800, 1200, 600, 300, 134, or 110.
bits	Number of data bits: 5, 6, 7, or 8.
stop	Number of stop bits: 0, 1, or 2 (0 represents 1.5 stop bits).
parity	"even", "odd", or "none".
stall	Stall/Unstall control: "on" or "off".
cts	Clear-to-Send handshake: "on" or "off".
autolf	Auto linefeed: "on" or "off".

Example 1:

```
ichannel = open device "hexdata", as "input"
```

Example 2:

```
file2 = open device "/dr1/rombytes", as "output"
```

Example 3:

```
p1 = open device "/port1", speed 1200, bits 8,
 parity "even"
```

Example 4:

```
        ! Open a window on the monitor with its origin
        ! at (20,6) and with a dimension of 40 by 12
        ! (centered on the display).  Full-scale
        ! coordinates of the objects to be displayed
        ! in the window are to be (1000, 1000).

channel = open device "/win", xorg 20, yorg 6,
  xdim 40, ydim 12, xscale 1000, yscale 1000
```

Remarks:

Input and output are handled through I/O channels. An I/O channel is a connection between TL/1 and the operating system. You create an I/O channel with the *open* command before conducting any input or output operations. When communication to an I/O channel is complete, you close the channel using the *close* command. Attempted access to a channel that has not already been opened results in a run-time error.

Typically, you open one channel for each device that requires input or output. You may open more than one channel to the same device, but this action is not recommended as it may result in non-standard operation.

The minimal open statement:

```
open ()
```

is used to open a channel to the operator's display and keypad for update. In this case, you need not specify the channel in subsequent *print* and *input* commands. Some other forms of the minimal *open* command include:

```
open device "/term2"   ! programmer's interface
open device "/port1"   ! RS232C port #1
open device "/port2"   ! RS232C port #2
open device "/ieee"    ! IEEE-488 interface
open device "/ieee/1"  ! IEEE-488 device at
                         address 1
```

In these cases, channel specification with *print* or *input* is also optional.

You may open channels to the programmer's monitor and keyboard, the operator's display and keypad, the two RS-232 ports, to text files on the hard or floppy disk drives to the IEEE-488 interface, and to devices attached to the IEEE-488 interface. Special rules apply to input and output depending on the device you are using. These rules are summarized below:

- **Operator's Display and Keypad** - to open a channel to the operator's interface, specify "/term1" as the device name. TL/1 accesses the operator's display as a 3-line by 42-column text area and escape sequences for display attributes are recognized.

  Input is read from the operator's keypad, including the soft keys and optional foot pedal (via the external switch interface). In the default line-buffered mode, the keypad alpha lock is engaged which causes each key press to return the alphabetical representation after the ENTER key is pressed. In the unbuffered mode, characters are sent to TL/1 immediately. If update mode is specified, input characters automatically appear on the display. The CLEAR key erases the keystrokes in the input buffer and on the display allowing for re-entry.

- **Monitor and Programmer's Keyboard** - to open a channel to the programmer's interface, specify "/term2" as the device name. TL/1 accesses the monitor through a text window, which becomes active until the TL/1 program completes execution. TL/1 accesses the monitor as an ANSI 3.64 compatible terminal. See Appendix B, "Control Codes for Monitor and Operator's Display," for more information. If a program which uses the monitor as an output channel is executed under the debugger, the message window will cover the debugger window.

  Input is read from the programmer's keyboard. In line-buffered mode, characters are stored until the Return key is pressed. In unbuffered mode, characters are sent to TL/1 immediately. If update mode is specified, input characters automatically appear on the display, and the rubout key is used to erase characters before they are sent to TL/1. In addition, typing Ctrl-U will erase all the characters on a line and typing Ctrl-R will reprint a line. Tabs are converted to sequences of spaces.

- **Windows** - to open a channel to a window, specify "/term1/win" as the device name for a window on the operator's display and "/win" (or "/term2/win") as the device name for a window on the monitor. These extensions allow window operations to be performed on /term1 and /term2 since they are actually implemented as windows covering each display.

  A window is created with the *open* command. This allows normal print and input to be done on windows just as it is done on any other display device. Windows are permitted to overlap each other. What is displayed is determined by the order in which the windows were created. A new window is always on top of all the other windows. An existing window may be moved to the front or the back using the *winctl* command. The *winctl* command also permits making a window invisible by "hiding" it, and making an invisible window visible by "unhiding" it.

The location of the upper, left-hand corner of a window is specified by xorg and yorg. The size of a window is specified by xdim and ydim. The size of the object to be displayed in the window is controlled by xscale and yscale. All references to locations inside a window and sizes of objects displayed in a window are made relative to the full-scale coordinates specified. For example, if xscaleand yscale are both 1000, the center of the window is (500, 500). If the object size is larger than the window, only part of the object will be visible at any given time.

All normal *print* and *input* statements operate on a window. Doing input on a window device open in update or read mode will cause input from programmer's keyboard (in the case of a window on the monitor) and from the operator's keypad (in the case of a window on the operator's display). Each window is an ANSI terminal with all of the escape sequences and control codes active as defined in Appendix B.

- **RS-232-C Ports** - to open a channel to one of the two RS-232-C ports, specify either "/port1" or "/port2" as the device name.

  When you open a port for input in buffered mode, characters are held until a carriage return character (hexadecimal OD) is read. Upon reading the carriage return, the input is sent to TL/1. In unbuffered mode, characters are sent to TL/1 immediately.

  When you open a port for update, the input characters are immediately echoed as output. If a delete character (hexadecimal 7F) is read as input, the last character is deleted and a backspace (08), space (hexadecimal 20), backspace (08) sequence is sent as output. If Ctrl-U is read as input, the input line is deleted. If Ctrl-R is read as input, all input since the last carriage return is re-sent as output. When a port is opened in any mode other than update, characters are not echoed, and the delete, Ctrl-U, and Ctrl-R characters have no special effect.

In both buffered and unbuffered mode, when XON/XOFF flow control is enabled, Ctrl-S is used to stall the output and Ctrl-Q is used to restore its flow.

- **IEEE-488 Interface and Devices** - When the 9100A/9105A is configured as an IEEE-488 talker/listener, the appropriate way to open the IEEE-488 interface is to open "/ieee". Since the 9100A/9105A is not a controller and cannot address other devices to listen or talk, IEEE-488 device name arguments containing an address list (e.g. "/ieee/1", "/ieee/2,4:2") are not relevant to operation as a talker/listener. Values printed to and input from the IEEE-488 interface are sent to and received from other devices on the IEEE-488 bus. It is the responsibility of the controller to ensure that the 9100A/9105A is addressed to listen or talk at the appropriate times.

  When the 9100A/9105A is configured as a controller, the IEEE-488 interface may be opened by opening device name "/ieee". A channel to a group of devices attached to the bus may be opened by opening device name "/ieee/*address list*", where *address list* is a comma-separated list of IEEE-488 addresses (described in the Arguments section above). The former form of device name is principally useful for IEEE-488 bus control via the *ieee* command, while the latter form is useful for access by the *print, input, poll,* and *ieee* commands.

- **Text Files on Disk** - to open a channel to a text file on a disk, specify the file name as the device name. Text files may be opened:

  - as output: this is the default. If the file named by the device <file name> option already exists, it is truncated to zero size; otherwise a new (empty) file having the specified name is created.

  - as append: the file named by the device <file name> option is opened to permit *print* commands to place new data after the existing file contents. The file must already exist or an error will be reported.

- as input: the file named by the device <file name> option must already exist, or an error is reported. The next *input* command on the channel will read data starting at the beginning of the file.

It is an error to have more than one channel open to a particular file at any given time. It is also an error to open a text file for both read and write access at the same time or to open a text file as "update."

Related Commands:

close, draw, draw ref, draw text, input, input using, poll, print, print using

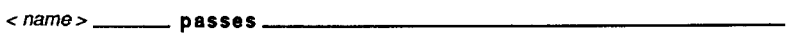
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Control Codes for Monitor and Operator's Display," appendix of this manual.
- The "Operator's Keypad Mapping to TL/1 Input," appendix in this manual.
- The "Programmer's Keyboard Mapping to TL/1 Input," appendix in this manual.
- The"9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488, 9100A-015" section of the *Technical User's Manual*".

Syntax:

```
<invocation> passes
```

Syntax Diagram:



Description:

Tests the termination status of a called program or function. The *passes* operator evaluates as true if the called function or program ends with a "passes" status and as false otherwise.

Arguments:

invocation	Program or function call.

Example 1:

```
if testbus addr $8000 passes then y = 1
```

Example 2:

```
if testramfull addr $1000, upto $1FFF passes then
 x = 1
```

Remarks:

Termination status indicates whether or not a UUT passes functional tests. Termination status is revised for every invocation.

Termination status can be:

passes	represents completion of a test without any unhandled fault conditions. The UUT is free from any faults that the test can detect.
fails	represents the existence of one or more unrepaired faults at the end of test execution.

A program that runs to completion without detecting any faults indicates that the UUT passes. Detection of a fault by the program (or any programs it calls) affects the termination status of the program. Any unhandled, unexercised fault condition causes the program to indicate that the UUT fails. Any fault condition that is exercised causes the program to indicate that the UUT fails if the last full iteration of the exerciser detected a fault and allows the program to indicate a "passes" if the last full iteration of the exerciser did not detect a fault. The termination status of a program is accumulated in the program that called it, so that if any called programs indicated a failure, the calling program also indicates that the UUT fails.

A fault condition can be handled by a block of statements called a fault condition handler. The fault condition handler has access to the arguments of the fault and the global variables of the test program. When a fault condition handler encounters either a *return* statement or its last statement the handler terminates, and execution resumes at the statement following the *fault* command.

If the handler does not execute a *fault* command, the fault condition is handled and disappears. In this case, the termination status is "passes".

A *fault* command with no fault name or arguments unconditionally sets the termination status to "fails."

When a *refault* or a *fault* command with a fault name is executed, the termination status is affected by the presence of other handlers or exercisers for the fault condition.

Related Commands:

execute, exercise, fault, fails, handle, if, refault, while

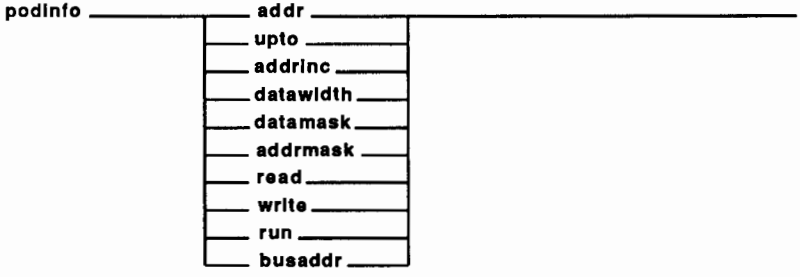
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
podinfo addr
podinfo upto
podinfo addrinc
podinfo datawidth
podinfo datamask
podinfo addrmask
podinfo read
podinfo write
podinfo run
podinfo busaddr
```

Syntax Diagram:



Description:

Returns the requested information about the current space.

Options:

addr	Returns the lowest valid address in the current space. All valid addresses in the current space are greater than or equal to this number.
upto	Returns the highest valid address in the current space. All valid addresses in the current space are less than or equal to this number.

addrinc	Returns the minimum valid address increment for the current space. All valid address increments are multiples of this number.
datawidth	Returns the width, in bits, of the data words in the current space.
datamask	Returns a bitmask with bits set for the valid data bits in the current space.
addrmask	Returns a bitmask of valid address bits. Note the least significant couple of bits may not be set if addrinc is greater than one.
read	Returns 1 if read is permitted in the current space and 0 otherwise.
write	Returns 1 if write is permitted in the current space and 0 otherwise.
run	Returns 1 if run UUT is permitted in the current space and 0 otherwise.
busaddr	Returns the default bus test address.

Returns:

A number is always returned.

Examples:

```
function testme (addr, upto)

      if addr < (podinfo addr) then fault\return
      if addr > (podinfo upto) then fault\return

      testramfast addr addr upto upto,
        addrstep 2

end function
```

Related Commands:

setspace, getspace, sysspace, podsetup, sysaddr, sysdata

For More Information:

- The "Overview of TL\1" section of the *Programmer's Manual.*
- *Supplemental Pod Information for 9100/9105A User's Manual.*

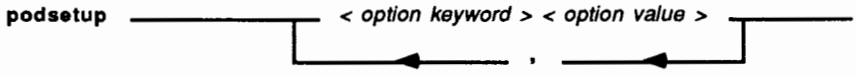
podsetup
special function

IEEE-488

Syntax:

```
podsetup { <option keyword> <option value> }
```

Syntax Diagram:



Description:

Accesses the pod to enable or disable reporting of faults directly sensed by the pod hardware.

Arguments:

Option Keyword	*Option Value(s)*
'report power'	"on" or "off". Enables or disables reporting of bad power supply level.
'report forcing'	"on" or "off". Enables or disables reporting of active forcing-input lines.
'report intr'	"on" or "off". Enables or disables reporting of active interrupt lines.
'report address'	"on" or "off". Enables or disables reporting of undrivable address-output lines.

'report data'	"on" or "off". Enables or disables reporting of undrivable data bus lines.
'report control'	"on" or "off". Enables or disables reporting of undrivable control-output lines.
'report special'	"on" or "off". Enables or disables reporting of special pod errors.
'enable *string*'	"on" or "off". Enables or disables a pod-dependent forcing line. The enable phrases all begin: 'enable ' and end with a pod-dependent *string*.
timeout	<expression> Changes the timeout time. The expression must be numeric.
option	<value> A pod-dependent setup option. The option and value are defined by the pod data file.

Example 1:

```
podsetup 'report power' "on"
```

Example 2:

```
podsetup 'enable ready' "off"
```

Example 3:

```
podsetup timeout 1000
```

Example 4:

```
podsetup 'report intr' "off", 'report power' "off"
```

Remarks:

The *podsetup* function may be written in keyword notation only.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- *Supplemental Pod Information for 9100A/9105A Users Manual.*
- The Fluke pod manual for the microprocessor you are using.

poll function

Syntax:

```
poll channel <channel>, event <condition>

poll (<channel>, <condition>)
```

Syntax Diagram:

Description:

The *poll* function allows you to examine the status of a device for certain conditions. For most conditions, the *poll* function returns a 1 if the condition is present, a 0 if it is not.

Arguments:

channel	An expression which identifies an open channel.
condition	"input", "output", "blocked", "errors", "break", or "srq". "input" - One or more characters are available on an input channel. If the input is a file, the end-of-file character has not been reached. If the specified channel is not open for input, a 0 is returned. "output" - The output buffer is empty on a serial channel open for output. If the channel is not open for output or is not a serial channel, a 0 is returned.

"blocked" - Output to a serial device is blocked (suspended) waiting for necessary protocol signals (as in CTS/RTS).

"errors" - Parity, framing, or overrun errors have occurred on a serial channel open for input.

"break" - A break character has been detected on the serial channel open for input.

"srq" - (IEEE-488 channels only) If the channel is open on the IEEE-488 interface, then the *poll* command returns 1 if a device is asserting SRQ (Service Request) on the IEEE-488 bus, and 0 if no device is asserting SRQ. If the channel is open on a group of devices, a serial poll is performed on the first device in the address list, and the resulting serial poll status byte is used as the return value of the *poll* command. (The 9100A/9105A must be configured as a controller in order to conduct a serial poll.)

If an I/O error occurs while attempting to process the poll command (for example, a timeout error occurs while attempting to perform a serial poll on an IEEE-488 device), then the "io_error" fault is raised with numeric argument err_num containing the 9100A/9105A error number and string argument err_msg containing a description of the error. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error(err_num, err_msg)
   declare numeric err_num
   declare string err_msg
   print using "Error ?#:?#", err_num, err_msg
end handle
```

Returns:

1, if condition exists.

0, if condition does not exist.

Note: The "srq" condition can *return* the result of performing a serial poll, as discussed above.

Example:

```
loop while (poll channel ichan, event "input") = 0
     wait time 1000          ! wait for input
end loop
```

Related Commands:

open

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488, 9100A-015" section of the "*Technical User's Manual*".

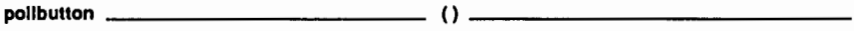
pollbutton function

IEEE-488

Syntax:

```
pollbutton ()
```

Syntax Diagram:



Description:

Determines if any I/O module or probe button presses are currently queued.

Returns:

1, if an I/O module or probe button is queued.

0, if the condition does not exist.

Example 1:

```
 ! Poll for any I/O module or probe button
sts = pollbutton ()
if (sts = 1) then
    readbutton()
end if
```

Remarks:

When using the READBUTTON TL/1 function, you may want to know before the function is executed, whether or not an I/O module or probe button has been pressed. The POLLBUTTON TL/1 function determines if there is at least one queued button press from the I/O module or the probe.

Related Commands:

clip, probe

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
polluut ()
```

Syntax Diagram:

Description:

Determines whether the pod is executing instructions in the RUNUUT mode.

Returns:

1, if the pod is executing instructions in RUNUUT mode.

0, if the pod is not executing instructions in RUNUUT mode. This could be caused by any of the following reasons:

- The pod wasn't put into RUNUUT mode.
- The pod has halted RUNUUT execution after reaching a breakpoint.
- A data compare equal (DCE) condition has occurred in an I/O module.

Examples:

```
program look
      .
      .
runuut addr $FFFFFF0
compare device "/mod1", patt "11011101"
```

(example is continued on the next page)

```
loop while polluut () = 1
     execute io_stimulus () ! a stimulus routine
                            ! you have written to
                            ! exercise the I/O
                            ! ports of your UUT
end loop
     .
     .
end program
```

Related Commands:

compare, runuut, waituut

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

pow function

IEEE-488

Syntax:

```
pow num <expression 1>, power <expression 2>

pow (<expression 1>, <expression 2>)
```

Syntax Diagram:

Description:

Computes the value of one argument raised to the power of the other argument.

Arguments:

expression 1	The floating-point value to raise by the power argument value.
expression 2	The floating-point power argument value.

Returns:

A floating-point number.

Examples:

```
f = pow num 3.0, power 3.0     ! result is 27.0
f = pow (6.0, 1.0/3.0)         ! result is cube root
                               ! of 6.0
```

Remarks:

If any of the following is true, an error will result:

- Both expression 1 and expression 2 are equal to 0.0.
- The expression 1 value is negative.

Also, overflow errors may occur for certain ranges of argument values which cause excessively large returned values.

Related Commands:

log

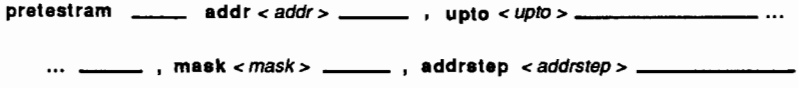
pretestram function

IEEE-488

Syntax:

```
pretestram addr <addr>, upto <upto>, mask <mask>,
     addrstep <addrstep>

pretestram (<addr>, <upto>, <mask>, <addrstep>)
```

Syntax Diagram:



Description:

Performs a very fast pretest of RAM to find any simple faults such as a totally dead memory chip, stuck address lines, or stuck data lines.

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask for data bits to test.
addrstep	Address increment.

Example:

```
if pretestram addr 0, upto $FFFE, mask $FFFF,
  addrstep 2 passes then
       ! If pretestram passes, do your customized
       ! test (or Pod Quick test) here.

       (example is continued on the next page)
```

```
        ! If your test finds an error, then execute
        ! diagnoseram, using the values for
        ! faultaddr, expdata, and data discovered by
        ! your program.

else

        ! No need to program anything.  When
        ! pretestram fails, it has full diagnostics.

end if
```

Remarks:

The action of *prestestram* is included in *testramfast* and *testramfull.*

Related Commands:

diagnoseram, testramfast, testramfull

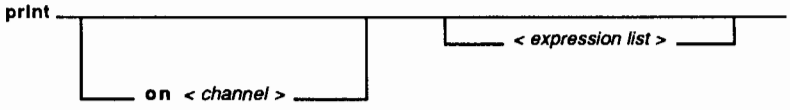
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
print [<expression list>]

print [on <channel>] [, <expression list>]
```

Syntax Diagram:



Description:

Outputs character strings and numbers to a character-oriented output device.

Arguments:

channel	A numeric expression that identifies a channel to a device open for output.
expression list	One or more expressions for output separated by commas.

Example 1:

```
print "text string"  ! output to operator's display
                      ! with no extra spaces
```

Example 2:

```
print on ch1, "text string = ", ts
                      ! output string followed by
                      ! value of ts in decimal
```

Remarks:

Before using the *print* command, the output device must be opened for output.

The *print* command without an "on" expression prints to the first device opened for output, append, or update. An error occurs if no device has been opened or if the device has subsequently been closed.

If an expression is a string, it is printed with no surrounding spaces. If an expression is numeric, it is printed in decimal with no surrounding spaces. If an expression is floating-point, it is printed in scientific notation format with six digits of precision following the decimal point.

The *print* command adds the termination character (which usually defaults to newline) associated with the channel after printing the last expression. A *print* command without any expression list prints a termination character (if any).

If the print channel is a serial port opened in buffered mode, the newline character is actually printed as either a carriage return or as a carriage return plus a line feed. This definition can be changed by pressing the SETUP MENU key and then the PORT1 or PORT2 softkeys found on the operator keypad. Output is not actually sent to a buffered serial port channel until a newline character is printed.

If the print channel is opened on either the IEEE-488 interface or a device attached to the IEEE-488 interface, EOI is automatically asserted with the termination character. In addition, if "EOI Enable" mode is turned on (see the *ieee* command for details), EOI is asserted with the last byte printed by the *print* command (which is not always a termination character, since the channel may have no associated termination character).

If an I/O error occurs during an attempt to process a *print* command (for example, a timeout error during an attempt to print to an IEEE-488 device), then the io_error fault is raised, with numeric argument err_num containing the 9100A/9105A error number, and string argument err_msg containing a description of the error. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error(err_num,err_msg)
     declare numeric err_num
     declare string err_msg
     print using "Error ?#:?#",err_num,err_msg
end handle
```

Related Commands:

open, print using

For More Information:

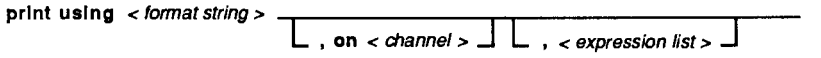
- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Control Codes for Monitor and Operator's Display," appendix of this manual.
- The "9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488, 9105-015" section of the *Technical User's Manual.*

print using statement

Syntax:

```
print using <format string> [, on <channel>]
     [, <expression list>]
```

Syntax Diagram:



Description:

Outputs strings and numbers to a character-oriented output device in a specified format.

Arguments:

format string	A string that defines the output format.
channel	A numeric expression that identifies a device open for output.
expression list	One or more expressions for output separated by commas.

Example 1:

```
print using "# # ## ####", 1, 2, 10, 110
     ! outputs the string:
     ! 1 2 10  110
```

Example 2:

```
print using "first &&&&&&&&, second %%%%, third
 is #@@@@", $FF, $10, 110
     ! outputs the string:
     ! first 11111111, second 0010, third is  0110
```

Example 3:

```
print using "########## ##########\nl", "hello",
 "world"
     ! outputs the string:
     ! hello      world
```

Example 4:

```
program squares
print "Table of Squares\nl"
     ! One new-line is generated by print,
     ! the other by the literal new-line.
     loop for i = 1 to 10
        print using "#@ squared is ##@\nl", i, i*i
     end loop
end squares
     ! outputs the following text:
     ! Table of Squares
     !
     !  1 squared is   1
     !  2 squared is   4
     !  3 squared is   9
     !  4 squared is  16
     !  5 squared is  25
     !  6 squared is  36
     !  7 squared is  49
     !  8 squared is  64
     !  9 squared is  81
     ! 10 squared is 100
```

Example 5:

```
print using "###^^^ ###^^^EEEEE\nl", 3.2, 3.2
     ! outputs the string:
     !    003  3.200E+00
```

Example 6:

```
print using "#^^^.^^^\nl", f
     ! with
     ! argument:     outputs the string:
     !
     ! 3.2              003.200
     ! 1000.3          1000.300
     ! -40.2359         -40.236
```

Example 7:

```
print using "?# ?% ?@ ?& ?^\nl", $32, $32, $32,
 $32, 50.0
     ! outputs the string:
     ! 50 32 50 110010 5.000000E+01
```

Remarks:

The *print using* command allows for highly structured or columnized output by use of format specifications contained in format strings. Format strings specify formatted output through combinations of literal characters and format pictures. There must be a format picture for every expression. It is a run-time error if there are more format pictures in the format string than expressions in the expression list.

If no channel is specified in the *print using* command, output is directed to the first device opened for output, append, or update.

A *print using* command does not automatically output a termination character at the end of the output. To print a new-line character with the *print using* command, include "\nl" in the format string.

Characters in the format string which are not part of a format picture are simply printed as given in the format string.

If the print channel is a serial port opened in buffered mode, the newline character is actually printed as either a carriage return or as a carriage return plus a line feed. This definition can be changed by pressing the SETUP MENU key and then the PORT1 or PORT2 softkeys found on the operator's keypad. Output is not actually sent to a buffered serial port channel until a newline character is printed.

If the print channel is opened on either the IEEE-488 interface or a device attached to the IEEE-488 interface, EOI is automatically asserted with the termination character (if any). In addition, if the 'EOI Enable' mode is turned on (see the *ieee* command for details), then EOI is asserted with the last byte printed by the *print using* command.

If an I/O error occurs while attempting to process a *print using* command (for example, a timeout error while attempting to print to an IEEE-488 device), the io_error fault is raised, with numeric argument err_num containing the 9100A/9105A error number and string argument err_msg containing a description of the error. For example, the following is a TL/1 code fragment for an io_error fault handler:

```
handle io_error (err_num, err_msg)
     declare numeric err_num
     declare string err_msg
     print using "Error ?#: ?#", err_num,
err_msg
end handle
```

A format picture is a string of one or more format characters; a format string contains zero or more format pictures. Format pictures are one of two types: fixed-width and variable-width. Fixed-width format pictures output data in columnized format, while variable-width format pictures output data using the minimum number of columns needed to express the data. (The result for variable-width pictures is the same as that achieved with the *print* command, except that it is possible to control the radix when printing numeric values.)

Fixed-Width Formatted Output

A fixed-width string format picture is a string of one or more "#" characters, where each "#" character represents one character in the string data to be printed.

A fixed-width numeric format picture is a string of zero or more optional digit places (leading "#" characters) followed by one or more required digit places, which also determine the radix (hexadecimal, decimal, or binary) for numeric data.

A fixed-width floating-point format picture is a string of zero or more optional digit places (leading "#" characters), followed by a required digit sequence, optionally followed by the string "EEEEE" to denote the exponent. If the exponent sequence is included, the number will be printed in scientific notation; otherwise, it will be printed in fixed-point notation. For both notations, the required digit sequence contains "^" characters. In addition, fixed-point notation allows at most one decimal point character in the sequence of "^" characters to specify the position of the decimal point. (Note: At least one "^" character must precede the decimal point.)

The fixed-width format picture characters are defined below:

Symbol	*Description*
#	An optional digit place for numeric and floating-point formats or a character place for string formats. In numeric formats, if the place is a non-significant zero, a space is printed. When the "#" symbol appears as the last position of a numeric format picture, it is a required digit place and defines the radix to be decimal.
%	A hexadecimal digit for numeric format pictures. If this digit is a non-significant zero, a zero is printed.
@	A decimal digit for numeric format pictures. If this digit is a non-significant zero, a zero is printed.

&	A binary digit for numeric format pictures. If the digit is a non-significant zero, a zero is printed.
^	Required digit place for floating-point numbers. A sequence of "^" characters may contain a single decimal-point character (which must be preceded by at least one "^" character), or may be followed by a fixed sequence of five "E" characters denoting the exponent (in which case the sequence of "^" characters may not include a decimal point).

When a numeric value is printed into a fixed-width format picture, the right-most digit place provides the radix in which the number is printed. The character "%" in the right-most position causes the number to be printed in hexadecimal. The characters "#" and "@" cause the number to be printed in decimal. The character "&" causes the number to be printed in binary. If there are fewer characters in the format picture than there are significant digits in the number, a run-time error occurs. If there are fewer significant digits in the number than there are format picture characters, the number is right-justified in the format picture.

For example, the expression "10" will be printed as " A" with "#%" as the format picture, but it will be printed as "0A" with "%%" as the format picture.

When a fixed-width format picture is used to print a string, the string is left-justified in the picture. If there are more characters in the string than in the picture, a run-time error occurs. If there are fewer characters in the string than in the picture, the extra picture characters are replaced by spaces. Only the "#" character may represent string characters in a format picture. For example, the string "data" will be printed as "data " when "######" is used as the format picture.

Variable-Width Formatted Output

Like a fixed-width format picture, a variable-width format picture describes the format for a single data value. The difference is that a variable-width format picture prints as many characters as are necessary to express the corresponding value. The result for each value is identical to what is generated by the *print* command, except that it is possible to specify the radix to use for printing a numeric value.

A variable-width format picture consists of a single "?" character, followed by a single character denoting the picture type.

The following special picture type characters are defined:

Symbol	*Description*
#	Print a variable-width string or decimal number.
%	Print a variable-width hexadecimal number.
@	Print a variable-width decimal number.
&	Print a variable-width binary number.
^	Print a variable-width floating-point number (in scientific notation with six digits of precision following the decimal point).
?	Print a single "?" character.

If the picture type character is not one of the above, then it will be printed literally (the same as if it had not been preceded by a "?" character).

Related Commands:

open, print

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "9100A/9105A Error Numbers" appendix in this manual.
- The "IEEE-488 9105-015" section of the *Technical User's Manual.*

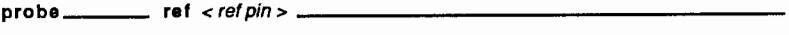
probe function

IEEE-488

Syntax:

```
probe ref <ref pin>

probe (<ref pin>)
```

Syntax Diagram:



Description:

Prompts the operator (on the operator's display) to probe the specified pin and press the ready button on the probe.

Arguments:

ref pin	Name of pin to probe.

Example:

```
probe ref "U1-1"
```

Remarks:

This command is used to verify that the probe is set up for a measurement. The 9100A/9105A waits until the probe button is pressed before continuing on past this command.

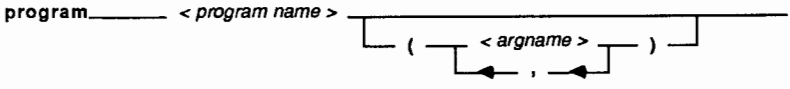
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
program <program name> [(<argname> {, <argname>})]
```

Syntax Diagram:



Description:

Specifies the beginning of a program block.

Arguments:

program name	Name of the program defined on the lines following the *program* statement.
argname	Name of an argument for this program.

Example 1:

```
program test8
      .
      .
      .
end program
```

Example 2:

```
program my_prog (a,b,c)
      .
      .
      .
end my_prog
```

Example 3:

```
program 'my.file'        ! Single quotes allow the
     .                   ! use of a period in
     .                   ! the program name.
     .
end program
```

Remarks:

The first statement of every TL/1 program must be a *program* statement. The *program* statement contains the program name and the names of any arguments passed to the program. The program block must be terminated with an end program statement or an end statement containing the same program name.

The name in the *program* and *end* statements must be the same. A valid program name contains from 1 to 10 characters and consists of only letters, numbers, and underscore characters (or periods, if the program name is enclosed in single quotes). A program name cannot be the same as the name of a built-in function.

Although TL/1 requires that a program name be spelled exactly the same wherever it appears, the case of letters is ignored when a program is looked up on the disk. Thus it is not possible to define two program names that differ only in case (such as PROG1 and prog1).

The scope of a program name is the UUT directory containing it plus the program library. Therefore, a program can call any other program in the same UUT directory, but not in another UUT directory.

The *program* argument list consists of one or more argument names, separated by commas; the argument list is enclosed in parentheses. The order of the names in this list is the same order in which the values for these arguments must be listed in positional notation calls to this program. If any arguments have default values, these values are assigned in the subsequent declaration blocks.

Program arguments may not be declared as arrays nor as global or persistent variables.

Declarations consist of any and all declaration blocks, function definition blocks, handler definition blocks, and exerciser definition blocks.

Related Commands:

declare, end, execute, exercise, function, handle, return

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

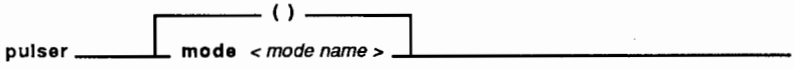
pulser function

IEEE-488

Syntax:

```
pulser mode <mode name>

pulser (<mode name>)

pulser ()
```

Syntax Diagram:



Description:

Turns on the probe in a pulser mode synchronized as specified by the *sync* command. The probe can pulse low, high, or toggle alternately high and low.

Arguments:

mode name	"off", "high", "low", or "toggle". (Default = "off")

Example:

```
pulser mode "toggle"
```

Remarks:

When the probe threshold is set to RS-232, a low level is pulsed to 0 volts. Since this is not a valid RS-232 low level, the green light of the probe will not be illuminated.

Related Commands:

sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

rampaddr function

IEEE-488

Syntax:

```
rampaddr addr <address>, mask <mask>

rampaddr (<address>, <mask>)
```

Syntax Diagram:

Description:

Performs a series of read operations, each at a different address. The number of reads and the values of the addresses are determined by the specified mask.

Arguments:

address	Address.
mask	Hexadecimal mask of ramp bits.

Example 1:

```
rampaddr addr $1000, mask 3
```

The 9100A/9105A performs the following:

```
read addr $1000
read addr $1001
read addr $1002
read addr $1003
```

Example 2:

```
rampaddr addr $123B, mask $42
! 123B hexadecimal = 0001 0010 0011 1011 binary
! 42 hexadecimal = 0000 0000 0100 0010 binary
```

The 9100A/9105A performs the following:

```
read addr $1239 ! 0001 0010 0011 1001 binary
read addr $123B ! 0001 0010 0011 1011 binary
read addr $1279 ! 0001 0010 0111 1001 binary
read addr $127B ! 0001 0010 0111 1011 binary
```

Remarks:

In the second example, the mask (42 hex) specifies the two address bits, 6 and 1. Four read operations are performed (2 raised to the power of 2.) The bits specified in the mask are ramped from all zeros to all ones with the right-most bit (bit 1) considered the LSB and the left-most bit (bit 6) considered the MSB of the ramped bits. The other address bits remain unaltered. It is much faster to ramp in groups of smaller masks (F0, F, F00) than to ramp one large mask (FFF), although the larger mask provides more complete coverage.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

rampdata function

IEEE-488

Syntax:

```
rampdata addr <address>, data <data>, mask <mask>

rampdata (<address>, <data>, <mask>)
```

Syntax Diagram:

Description:

Performs a series of *write* operations at the specified address.

Arguments:

address	Address.
data	First data value.
mask	Hex mask of the data bits to be ramped.

Example:

```
rampdata addr $123B, data $25, mask $43
! data 25 hex = 0010 0101 binary
! mask 43 hex = 0100 0011 binary
```

The system performs eight writes as shown below:

```
write addr $123B, data $24   ! 0010 0100
write addr $123B, data $25   ! 0010 0101
write addr $123B, data $26   ! 0010 0110
write addr $123B, data $27   ! 0010 0111
```

(example is continued on the next page)

```
write addr $123B, data $64    ! 0110 0100
write addr $123B, data $65    ! 0110 0101
write addr $123B, data $66    ! 0110 0110
write addr $123B, data $67    ! 0110 0111
```

Remarks:

You specify the original data and the data bits to be ramped. In the previous example, the mask (43 hex) specifies the three data bits 6, 1 and 0. This means that there are eight *write* operations (2 raised to the third power). The data bits are ramped from all zeros to all ones with the right-most bit (bit 0) considered the LSB and the left-most bit (bit 6) considered the MSB of the ramped bits. The other data bits remain unaltered.

The following command performs a ramp for an eight-bit pod that is equivalent to the 9000-series RAMP @ 1122 command:

```
rampdata addr $1122, data 0, mask $FF
```

This statement performs 256 writes (data 00-FF) at address 1122 on a 16-bit pod.

Similarly, the following command performs 65536 writes (data 0000 - FFFF) at address 1122:

```
rampdata addr $1122, data 0, mask $FFFF
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

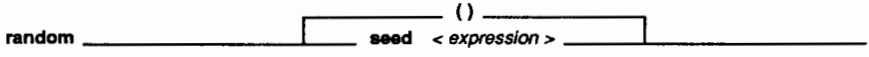
random function

IEEE-488

Syntax:

```
random [seed <expression>]

random ([<expression>])
```

Syntax Diagram:



Description:

Produces pseudorandom sequences of numbers.

Arguments:

seed	Providing this optional value changes the sequence of numbers. (Default = $FFFFFFFF)

Returns:

Returns a pseudorandom 32-bit numeric value.

Example:

```
loop for i = 1 to 100 ! print 100 random numbers
    print random()
end loop
```

Remarks:

The seed argument can change the sequence of pseduorandom numbers that are generated by the random function. Setting the seed to zero begins a new sequence of numbers based on the current time of day.

Setting the seed to any number other than zero or $FFFFFFFF, begins a new sequence based on that number. Using the same number again generates the same sequence of numbers.

The seed value can also be set to $FFFFFFFF (Default), which returns the next number in the current pseudorandom sequence.

The sequence of numbers returned by random has the appearance of randomness, but no specific property of the sequence is guaranteed. The sequences of numbers generated by *random* may change in subsequent software revisions to provide more nearly random sequences.

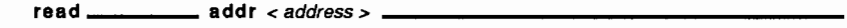
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
read addr <address>

read (<address>)
```

Syntax Diagram:



Description:

Returns the data located at the specified address.

Arguments:

address	Address from which to read data.

Returns:

The data read at the specified address.

Example 1:

```
data2 = read addr here
     ! Reads the data at the address specified by
     ! the value of the user-defined variable here
     ! and stores it in the variable data2.
```

Example 2:

```
data2 = read (here)
     ! The positional notation equivalent
     ! of the keyword notation in the above
     ! example.
```

Example 3:

```
data1 = read ($8FF0)      ! Reads the hex data at
                          ! address 8FF0 into
                          ! the variable data1.
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual*.

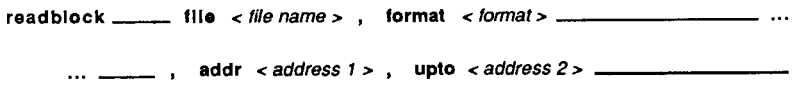
readblock function

IEEE-488

Syntax:

```
readblock file <file name>, format <format>,
     addr <address 1>, upto <address 2>

readblock (<file name>, <format>, <address 1>,
     <address 2>)
```

Syntax Diagram:



Description:

Reads the data from the specified address range and stores this data in the specified text file.

Arguments:

file name	The name of the file in which to store the data. If a full path name is not specified, the data will be stored in the specified file in the current UUT directory.
format	The ASCII format in which the data was previously stored. Either "motorola" or "intel". (Default="motorola".)
address 1	Start address.
address 2	End address.

Example 1:

```
readblock file "testlcd", format "motorola",
     addr $7000, upto $7FD4
```

Example 2:

```
readblock ("pgm1", "motorola", $1000, $1FFF)
```

Example 3:

```
readblock file "/HDR/testdata", format "motorola",
     addr $7000, upto $7FD4
```

Related Commands:

loadblock, writeblock

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

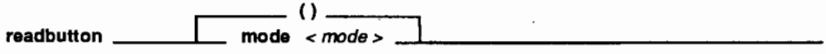
readbutton function

IEEE-488

Syntax:

```
readbutton mode <mode>

readbutton (<mode>)

readbutton ()
```

Syntax Diagram:



Description:

Waits for you to press the I/O module or probe button.

Arguments:

mode	"beep" or "nobeep". Default = "beep"

Returns:

The name of the device that is selected.

Example 1:

```
 ! Wait for any I/O module or probe button
device = readbutton ()
```

Example 2:

```
 ! Wait for probe button with no beep
device = readbutton("nobeep")
 ! value of device variable is now "/probe"
```

Example 3:

```
 ! Wait for I/O Module button with beep
device = readbutton mode "beep"
 ! value of device variable is now "/mod4B"
```

Remarks:

It is often inconvenient to continue pressing the <RETURN> key on the programmer's keyboard or the <ENTER/YES> key on the operator's keypad while probing points on a UUT. The *readbutton* command delays program execution until the I/O module or probe button can be pressed.

When the proper button is pressed, the function returns with the name of the device.

Related Commands:

clip, probe

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

readdate function

IEEE-488

Syntax:

```
readdate time <expression>

readdate (<expression>)
```

Syntax Diagram:

Description:

Returns a string which contains the date. To access the current date, the expression should be the value returned from the *systime* function.

Arguments:

expression	A number returned by the *systime* function that represents a number of elapsed seconds.

Returns:

A string representing a date.

Example:

```
t = systime ()
X = readdate time t    ! Stores the date
                       ! from variable t in string
                       ! format in the variable X

print readdate (t)     ! Prints the data from
                       ! variable t in string format
```

Remarks:

The *readdate* function returns a string containing the date in the format:

YYYY/MM/DD

where YYYY = year, MM = month, and DD = day.

For example, the string "1986/07/06" represents July 6, 1986.

Related Commands:

readtime, systime

For More Information:

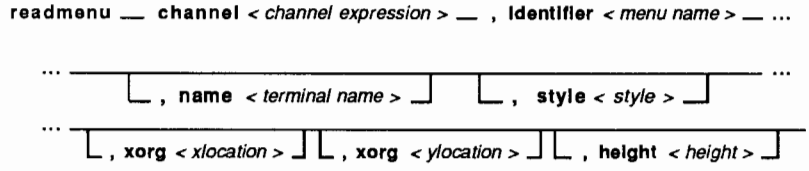
- The "Overview of TL/1" section of the *Programmer's Manual.*

readmenu
function

Syntax:

```
readmenu channel <channel expression>, identifier
    <menu name>  [, name <terminal name>] [, style
    <style>] [, xorg  <xlocation>] [, yorg
    <ylocation>] [, height <height>]
```

Syntax Diagram:



Description:

Used to display and read from a menu defined by the *define menu* command.

Arguments:

channel expression	A numeric expression to define a channel opened to read key presses used to make menu selections. Note that /term1 and /term2 are also considered windows.
menu name	A menu name as defined by the define menu command. If the menu does not exist, *readmenu* returns an empty string ("").
terminal name	The name of the display device to draw the menu on. The only name allowed is "/term2" (this is also the default value).

style	0 - The menu is a non-button menu (Default). 1 - The menu is a button menu.
xlocation	A numeric expression that defines the horizontal location of the upper left-hand corner of the menu in characters. If xorg is not specified, the default is to center the menu on the display.
ylocation	A numeric expression that defines the vertical location of the upper left hand corner of the menu in characters. If yorg is not specified, the default is to center the menu on the display.
height	A numeric expression that defines the maximum height of the menu in characters. If not specified, it defaults to the size required to list all the menu items.

Returns:

A string indicating which menu item is selected.

Example 1:

```
response = readmenu channel channel, identifier
 "M1"
            ! Read from menu M1
```

Example 2:

```
response = readmenu channel channel, identifier
 "M2"
            ! Read from menu M2
```

Example 3:

```
        ! Pop up a menu when the user pushes a key. On
        ! subsequent key presses, start the menu at
        ! the last place the user was on the menu.  If
        ! the user makes no selection, exit the
        ! program loop.  This example assumes the user
        ! has written a program called doaction which
        ! causes the action indicated by the menu
        ! selection to be done.

keyboard = open device "/term1", as "input", mode
 "unbuffered"
response = "M1"
notdone = 1
loop while notdone
     if (poll channel keyboard, event "input") then
          response = readmenu channel keyboard,
           identifier response
          if response = "" then
               notdone = 0
          else
               doaction action response
          end if
     end if
end loop
```

Remarks:

Menus are drawn in two styles, button or no button. In the no button style, a menu is simply a list of selectable items. In the button style, the menu is drawn as a collection of buttons with a box drawn around each menu item.

The user can type keys to make a menu selection if keys were defined for that menu. The keys defined are displayed in front of each menu item.

There is also a menu cursor displayed in a menu. The menu cursor is shown by inversing the active field. To select the item at the menu cursor enter a carriage return. To select a submenu at the menu cursor (submenus are drawn with a "->" appearing in the right hand side of the menu item) enter a carriage return or a right arrow key. To move the menu cursor down, press the

down arrow key, and to move the menu cursor up, press the up arrow key. The height argument specifies the maximum height of the menu on the screen. If the menu items will not fit in the height specified, the menu becomes a scrollable menu. This is indicated by the menu having the word "More" and a down arrow appear on the bottom border. Pressing the cursor down key on the last visible field causes the menu to scroll down. The word "More" and an up arrow now appears on the top border to indicate the menu may be scrolled up by pressing the up arrow on the top most visible field.

If a selection is made by the user, a string indicating that selection is returned. That string is of the form MMM1-III1/MMM2-III2/MMM3-IIII, where IIII is the selection made on menu MMM3, a submenu of menu MMM2 which is a submenu of MMM1. If the size of the string is larger than 255 characters, a string overflow error will be generated.

A menu can be forced to start up at a particular selection or submenu by passing the menu name of a particular selection. The menu name is of the form returned when a menu selection is made. For example, to start a menu in a submenu selection, pass menu as "MMM1-III1/MMM2-IIII".

If the menu style was 1 (indicating buttons), readmenu will accept escape sequences from a touch-sense interface. The escape sequence recognized is "\1B[>2;*xxx*n". The *xxx* is the location on the screen, assuming the screen is divided into 10 columns by 12 rows. The locations are numbered from 000 (the upper left corner) through 119 (the lower right corner).

The device is a channel opened for reading by the *open* command. It is recommended that this channel be opened as "input" mode "unbuffered". If the channel is buffered, you will have to type line terminators to cause menu action and this is probably undesirable. This channel will be flushed before characters are read if it is not a disk file.

xorg and yorg specify the position of the menu on the screen in character cells in the default font. If xorg is missing, xorg will default to center the menu horizontally, and if yorg is missing, yorg will default to center the menu vertically.

Related Commands:

define menu, remove menu, open

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

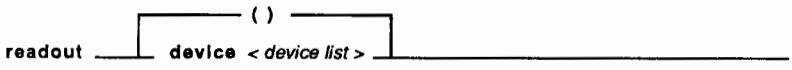
readout function

IEEE-488

Syntax:

```
readout device <device list>

readout (<device list>)

readout ()
```

Syntax Diagram:



Description:

Reads response data from the I/O module or the probe, and stores it in the system's memory. The response data is then available via the *sig*, *count*, and *level* commands.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")

Example 1:

```
mod = clip ref "u3", pins 40

arm device mod
     rampdata addr 0, data 0, mask $FF
     rampdata addr 0, data 0, mask $FF00
readout device mod
```

Example 2:

```
arm device "/mod1,/mod2"
      .
      .
readout device "/mod1,/mod2"
```

Remarks:

The *readout* command terminates signature gathering started by the *arm* command. Therefore *arm* and *readout* should be considered as beginning and ending statements for a TL/1 response-gathering block.

Related Commands:

arm, checkstatus, clip, count, level, sig, stopcount, sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

readspecial function

Syntax:

```
readspecial addr <address>

readspecial (<address>)
```

Syntax Diagram:

Description:

Returns the data located at the specified virtual address. This allows access to the virtual addresses that, in some pods, are used for special operations. This command should only be used when you know that the normal *read* command does not provide the required special operation.

Arguments:

address	Address from which to read data.

Returns:

The data at the specified address.

Example 1:

```
value = readspecial addr $F0000018
```

Example 2:

```
value = readspecial ($F0000018)
```

Remarks:

Incorrect use of this special-purpose command can place the pod and the 9100A/9105A mainframe in inconsistent states.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

readstatus
function

Syntax:

```
readstatus ()
```

Syntax Diagram:

Description:

The *readstatus* command reads the values on the status lines of the pod. To interpret the status data, refer to the pod's operator's manual. As with the *read* function, a value is accessed by assigning the function invocation to a variable.

Returns:

The status word.

Example:

```
status = readstatus()     ! the variable status is
                          ! loaded with data from
                          ! the function readstatus
```

Related Commands:

writecontrol

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

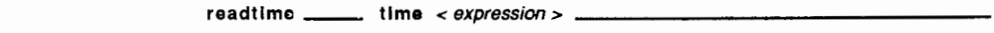
readtime function

IEEE-488

Syntax:

```
readtime time <expression>

readtime (<expression>)
```

Syntax Diagram:



Description:

Returns a string containing the time. To access the current time, the expression should be the value returned by the *systime* function.

Arguments:

expression	A number returned by calling *systime*, or a number representing elapsed seconds.

Returns:

A string representing a time.

Example 1:

```
t = systime ()
print "the time is ", readtime time t
      ! prints "the time is 15:36:58 (or
      ! whatever time the value of t indicates)
```

Example 2:

```
start = systime ()
testuut
print "TestUUT time ", readtime (systime () -
 start)
      ! prints: "TestUUT time 00:05:03" if,
      ! testuut took five minutes and three
      ! seconds to execute.
```

Remarks:

The *readtime* function returns a string containing the time in the format:

HH:MM:SS

where HH = hours, MM = minutes, and SS = seconds.

For example, the string "15:36:58" represents 36 minutes and 58 seconds past 3 o'clock in the afternoon. In this format, a list of strings ordered by the ASCII value of the characters from left to right is also ordered chronologically.

Related Commands:

readdate, systime

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

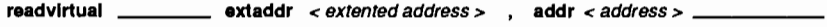
readvirtual function

IEEE-488

Syntax:

```
readvirtual extaddr <extended address>, addr
    <address>

readvirtual (<extended address>, <address>)
```

Syntax Diagram:



Description:

Readvirtual is a complete replacement for the obsoleted *readspecial* command. Returns the data located at the specified virtual address. This allows access to the virtual addresses that, in some pods, are used for special operations. This command should only be used when you know that the normal *read* command does not provide the required special operation.

Arguments:

extaddr	Upper 32-bits of virtual address.
addr	Address from which to read data.

Returns:

The data at the specified address.

Example 1:

```
value = readvirtual extaddr 0, addr $F0000018
```

Example 2:

```
value = readvirtual (0, $F0000018)
```

Remarks:

Incorrect use of this special-purpose command can place the pod and the 9100A/9105A mainframe in inconsistent states.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

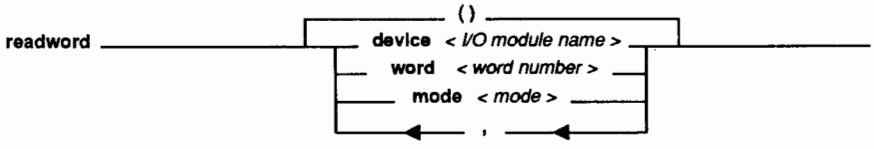
readword function

IEEE-488

Syntax:

```
readword [device <I/O module name>] [, word <word
    number>][, mode  <mode>]

readword(<I/O module name> , <word number> , <mode>)
```

Syntax Diagram:



Description:

Allows a group of I/O module pins to be read as a single group of values.

Arguments:

I/O Module name	I/O module name ("/mod1", "/mod2", "/mod3", or "/mod4"). (Default = "/mod1")
word number	The number of the word group. Valid values are from 1 through 5. (Default = 1)
mode	The mode in which to read the word. Valid values are "now" or "stored". (Default = "now")

Returns:

A string representing the state of the pins specified by the *setword* function.

Example 1:

```
! Read the current level of pins 1-4 of I/O Module #1
setword device "/mod1", word 1, as_pins "1 2 3 4"
binstr = readword device "/mod1", word 1, mode "now"
```

Example 2:

```
! Read the stored level of pins 1-4 of I/O Module #1
clockfreq device "/mod1", freq "1MHZ"
edgeoutput device "/mod1", start "at_vectordrive"
sync device "/mod1", mode "capture"
syncoutput device "mod/1", mode "intfreq"
setword device "/mod1", word 1, as_pins " 1 2 3 4"
vectorload device "/mod1", file "demo"
arm device "/mod1"
  vectordrive device "/mod1", startmode "now"
readout device "/mod1"
binstr = readword device "/mod1", word 1, mode "stored"
if instr(binstr,"*") = 0 then
  wordone = val(binstr,2)
else
  wordone = 0
end if
```

Remarks:

In the "now" mode, the *readword* function mimics the operation of the INPUT WORD operation from the front panel. The current logic level of the pins in the word are grouped into a string. In the "stored" mode, the *readword* function takes the clocked level history information from the values that are stored in memory from the latest readout command.

A pin level of HIGH is represented as a logic "1". A pin level of LOW is represented as a logic "0". Any other pin level is represented as an "*", so use the *level* function to determine the actual level.

Setword can make pin grouping assignments. Groupings may be made from 1 to 40 pins.

Related Commands:

setword, writeword

For More Information:

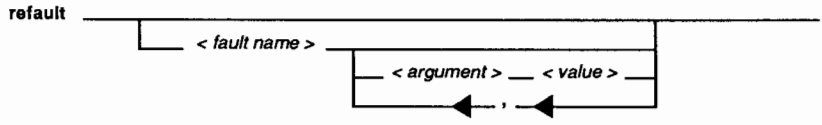
- The "Overview of TL/1" section of the *Programmer's Manual.*

refault statement

Syntax:

```
refault [ <fault condition> ]
```

Syntax Diagram:



Description:

Pass a fault condition to the caller for further processing.

Arguments:

fault condition	Name of the fault condition to be raised.
argname	Name of an argument.
argvalue	An expression which provides the value for an argument.

Example 1:

```
handle pia_out    ! Handler for the pia_out fault
                  ! condition
                  !
     refault      ! Raise the pia_out fault
                  ! condition in the caller of the
                  ! invocation that activated this
                  ! handler for pia_out.
end handle
```

Example 2:

```
! In this example, the outer program calls the
! inner function which raises a fault
! condition called fault1. After the handler
! processes the fault condition, it may wish
! to raise it to the attention of a handler
! activated by the outer program. The refault
! command allows raising the same fault
! condition in the calling block.

program outer

     handle fault1      ! Handler for the fault
                        ! fault condition
          print "executing outer's handler"
          fault         ! Set termination status to
                        ! "fails"
     end handle

     function inner
          handle fault1 ! Another handler for the
                        ! fault1 fault condition
               print "executing inner's handler"
               refault  ! Re-raise the same fault
                        ! condition in the control
                        ! program
          end handle

          if subcircuit5() fails then
                        ! Test sub-circuit 5
                        ! for faults
               fault fault1   !Raise the fault1
                              !fault condition
          end if
     end function

     execute inner()

end program
```

Example 3:

```
! In this example, the handler for fault2 in
! the inner function wants to raise a
! different fault condition, but doesn't want
! to activate the handler for that fault that
! is activated in the inner function. The
! refault command raises this new fault
! condition in the block that called the inner
! function (in this case, the outer program).

program outer

     handle fault1        ! Handler for the fault1
                          ! fault condition
          print "executing outer's handler for
           fault1"
          fault           ! Set termination status
                          ! to "fails"
     end handle

     function inner

          handle fault2   ! Handler for the fault2
                          ! fault condition
               print "executing inner's handler"
               refault fault1 ! Raise the fault1
                          ! fault condition in the
                          ! outer program, not in
                          ! the inner function
          end handle

          handle fault1
               print "executing inner's handler
                for fault1"
          end handle
```

```
            if subcircuit16() fails then
                 ! Test sub-circuit 16

                 fault fault2   ! Raise the fault2
                                ! fault condition

            end if

        end function

        execute inner()

    end program
```

Remarks:

The *refault* command is an advanced feature for fault condition handling. You will not need it for most test programs.

The purpose of the *refault* command is to permit fault condition handlers at outer levels of a program to see fault conditions even though they have been handled at a lower level. Once a fault condition is handled, it normally disappears. A *fault* command in the handler could be used to preserve the (failure) termination status for the program, but if the *fault* command has the same fault condition as the name of the handler, an infinite recursion will occur.

This recursion can be avoided by using a *refault* command instead of a *fault* command. When the *refault* command is used in a handler, the effect is as if the fault condition that invoked the handler had been raised in the invocation that activated that handler. This insures that the current handler is not invoked again, thereby avoiding an infinite recursion.

Using the *refault* command without a fault name raises the fault condition that invoked the handler, but raises it in the calling block. Using the *refault* command with a fault name raises the new fault condition in the calling block.

Related Commands:

abort, fault, handle, return

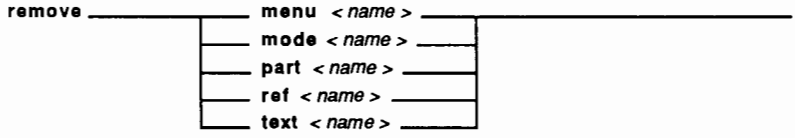
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
remove menu [<menu name>]

remove mode [<mode name>]

remove part [<part name>]

remove ref  [<ref name>]

remove text [<text name>]
```

Syntax Diagram:



Description:

Removes definitions made by the *define* commands.

Options:

menu name: The name of the menu definition to remove from the list of menu definitions. If the menu string is of the form "MMMM", that menu is removed. If the menu string is of the form "MMMM-IIII", that menu item is removed, but the rest of the menu remains. If the name is the null string (""), all of the menu definitions are removed.

mode name: The name of the mode definition to remove from the list of mode definitions. If the name is the null string (""), all of the mode definitions are removed. A comma-separated list of mode names (with no spaces in the list) allows more than one mode definition to be removed.

part name: The name of the part definition to remove from the list of part definitions. If the name is the null string (""), all of the part definitions are removed. A comma-separated list of part names (with no spaces in the list) allows more than one part definition to be removed.

ref name: The name of the reference designator definition to remove from the list of ref definitions. If the name is the null string (""), all of the ref definitions are removed. A comma-separated list of ref names (with no spaces in the list) allows more than one ref name to be removed.

text name: The name of the text definition to remove from the list of text definitions. If the name is the null string (""), all of the text definitions are removed. A comma-separated list of mode names (with no spaces in the list) allows more than one text name to be removed.

Example 1:

```
remove part ""          ! Remove all part definitions
```

Example 2:

```
remove part "box"       ! Remove the part definition
                        ! named "box"
```

Example 3:

```
remove ref ""          ! Remove all ref definitions
```

Example 4:

```
remove menu "M1"       ! Remove menu M1
```

Example 5:

```
remove menu "M1-a"     ! Remove item a from menu M1
```

Remarks:

Removing a definition frees up any memory used by that definition. All of the definitions are removed automatically at the start of each new TL/1 invocation. A new TL/1 invocation is started by pressing the REPEAT or EXEC keys on the operator's keypad or by pressing the EXECUTE or INIT softkeys when in the debugger.

Related Commands:

define mode, define menu, define part, define ref, define text

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

reset function

IEEE-488

Syntax:

```
reset device <device list>

reset (<device list>)

reset ()
```

Syntax Diagram:

Description:

Configures the response-gathering hardware for the probe or an I/O module to a default state.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")

Example 1:

```
reset device "/mod1"
```

Example 2:

```
iomod = clip ref "u5",pins 40
reset device iomod
```

Remarks:

The settings as a result of reset are as follows:

counter	=	"transition"
edge	=	"+" (rising edge for external start, stop, and clock
enable	=	"always"
sync	=	"pod" for an I/O module
	=	"freerun" for the probe
threshold	=	"ttl"
stopcount	=	"1"
syncoutput=		"intfreq"
edgeoutput=		"+" (rising edge for drive start, stop, and clock)
enableoutput		"always"
clockfreq	=	"1MHz"

Also note that although the syncoutput, edgeoutput, enableoutput, and clockfreq settings are reset, they are only used when a 9100-017 Vector Output I/O Module is connected.

In addition, all I/O module lines are placed in the high-impedance state.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

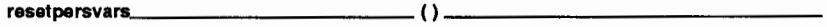
resetpersvars function

IEEE-488

Syntax:

```
resetpersvars ()
```

Syntax Diagram:



Description:

Resets the persistent variable set to the empty set.

Examples:

For each of the following example programs, assume that the persistent variable set initially contains:

```
Name    Type        Value

pv1     numeric     3
pv2     string      "foo"
pv3     string      "bar"
```

After executing the following command:

```
resetpersvars()
```

the persistent variable set is empty.

After executing the following program:

```
program prog1
     declare persistent numeric pv1
     resetpersvars()
     pv1 = 4
end program
```

the persistent variable set contains:

```
Name     Type          Value

pv1      numeric       4
```

After executing the following program:

```
program prog2
     declare persistent string pv2
     declare persistent string pv3
     function foo
        declare persistent string pv2
     end function
     resetpersvars()
     foo()
end program
```

the persistent variable set contains:

```
Name     Type          Value

pv2      string        "foo"
```

Remarks:

A local copy of any persistent variables known by the currently executing TL/1 program is retained, along with the current values. If any such persistent variables are subsequently redeclared or are assigned a value, they are added back to the persistent variable set.

The *resetpersvars* command allows purging of the persistent variable set, which would otherwise accrete forever (or until power was cycled).

Related Commands:

clearspersvars

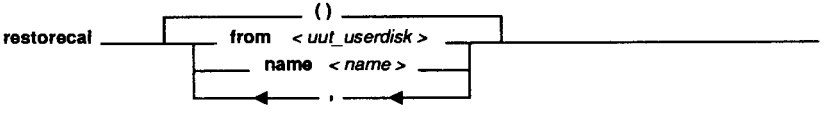
restorecal function

IEEE-488

Syntax:

```
restorecal [from <uut_userdisk>,name<name>]

restorecal (<uut_userdisk>,<name>)

restorecal ()
```

Syntax Diagram:



Description:

Restores the calibration values for the I/O module and the probe from the requested UUT or USERDISK.

Arguments:

uut_userdisk	USERDISK or UUT Default = "USERDISK"
name	USERDISK or UUT name Default = "" (current USERDISK or UUT)

Example 1:

```
! restore calibration values from the current UUT
restorecal from "UUT"
```

Example 2:

```
! restore calibration value from the UUT DEMO
restorecal from "UUT", name "DEMO"
```

Example 3:

```
! restore calibration value from USERDISK /DR1
restorecal from "USERDISK", name "/DR1"
```

Remarks:

This function is similar to the front panel RESTORE CALDATA operation and restores calibration values from a TL/1 program. Calibration values may be restored from a USERDISK or UUT.

If the name of the USERDISK or UUT is the null string (""), then the current USERDISK or UUT is used. If the USERDISK or UUT is named, the calibration values are restored from the named USERDISK or UUT. Calibration values are restored for all I/O modules and the probe.

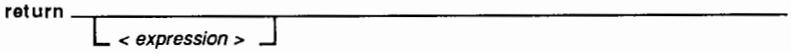
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
return [<expression>]
```

Syntax Diagram:



Description:

Causes a program, function, handler, or exerciser to terminate. Execution continues at the statement following the invocation statement (for programs and functions) or at the statement following the *fault* command (for handlers).

Arguments:

expression	The value to return.

Returns:

As an option, the value of an expression can be returned when a function or program terminates.

Example 1:

```
return          ! Return control to calling program
```

Example 2:

```
return d + 1    ! Return control to calling program.
                ! The value that the function
                ! returns, is the value of the
                ! variable d, plus 1.
```

Remarks:

The value of an expression can be returned when a program or function terminates. When the program or function is invoked as part of an expression the returned value is used in the expression.

A program or function that returns a value must do so explicitly. A program or function may not return values of two different types or return a value in one place without returning a value in every other place.

After the last statement of a block is executed, a *return* command is performed implicitly.

Related Commands:

execute, exercise, function, handle, program

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

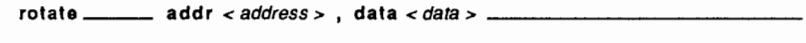
rotate function

IEEE-488

Syntax:

```
rotate addr <address>, data <data>

rotate (<address>, <data>)
```

Syntax Diagram:



Description:

Writes a data pattern to the specified address. The data is then rotated to the right and a *write* operation is performed. This is repeated as many times as there are data bits. Therefore, the last *write* performed writes the original data rotated one bit left.

Arguments:

address	All write operations are written to this address.
data	Initial data value.

Examples:

```
rotate addr $FFFFE, data $1234
```

The system performs the following sixteen operations:

```
write addr $FFFFE, data $1234
write addr $FFFFE, data $091A
write addr $FFFFE, data $048D
write addr $FFFFE, data $8246
write addr $FFFFE, data $4123
write addr $FFFFE, data $A091
```

(example is continued on the next page)

```
write addr $FFFFE, data $D048
write addr $FFFFE, data $6824
write addr $FFFFE, data $3412
write addr $FFFFE, data $1A09
write addr $FFFFE, data $8D04
write addr $FFFFE, data $4682
write addr $FFFFE, data $2341
write addr $FFFFE, data $91A0
write addr $FFFFE, data $48D0
write addr $FFFFE, data $2468
```

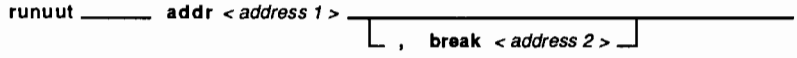
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
runuut addr <address 1> [, break <address 2>]
```

Syntax Diagram:



Description:

Causes the UUT to begin executing instructions from its own memory, asynchronously to the system.

Arguments:

address 1	Start address.
address 2	Break address.

Example 1:

```
runuut addr $1235
     ! start at hex address 1235
```

Example 2:

```
runuut addr (read addr $FFFE)
     ! start at address read from location FFFE
```

Example 3:

```
runuut addr $1235, break $2000
     ! start at hex address 1235
     ! stop at hex address 2000
```

Remarks:

Typically, *runuut* would be followed by *waituut* with a suitable maxtime value which would allow the UUT time to complete its operation.

Some pods have a breakpoint capability which can optionally be enabled by specifying a stop address with the "break" argument. Generally, pods designed before the 80286-era cannot use the "break" feature. Refer to your pod manual for more specific information.

Execution of *runuut* continues until one of the following events occurs:

- The pod encounters a breakpoint.
- A DCE condition occurs.
- A *haltuut* is executed.
- The time specified in a *waituut* expires.
- The RESET key is pressed on the operator's keypad.
- The RUN UUT HALT command is entered from the operator's keypad.

Faults which occur during the execution of *runuut* are reported on the subsequent *haltuut* or *waituut*. Attempts to perform any pod-related operations except *waituut*, *haltuut*, or *polluut* will result in an error if the *runuut* is still active. You must execute *haltuut* or *waituut* before attempting any other pod-related operations.

Related Commands:

compare, haltuut, polluut, waituut

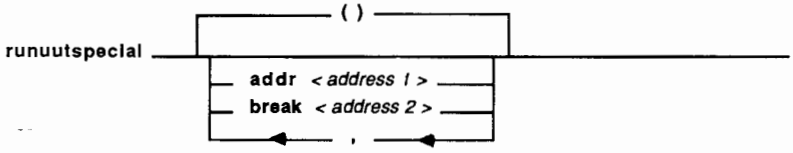
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

Syntax:

```
runuutspecial addr <address 1> [, break <address 2>]

runuutspecial (<address 1>, <address 2>)

runuutspecial ()
```

Syntax Diagram:



Description:

Causes the UUT to begin executing instructions from its own memory, asynchronously to the system. Execution begins at the virtual address specified. If none is specified, *runuutspecial* defaults to an address that is pod dependent.

Arguments:

address 1	Virtual start address.
address 2	Stop address. (Default = 0)

Example 1:

```
runuutspecial ()
     ! start at pod-dependent starting
     ! address.
```

Example 2:

```
runuutspecial addr (read addr $FFFE)
     ! start at virtual address read from
     ! location FFFE
```

Example 3:

```
runuutspecial addr $1235, break $2000
     !  start at hex virtual address 1235
     !  stop at hex address 2000
```

Remarks:

Some pods have special addresses where a *runuut* may begin. Such addresses may read a reset or interrupt vector from memory and begin execution at that address.

Typically, *runuutspecial* would be followed by *waituut* with a suitable maxtime value that allows the UUT time to complete its operation.

Some pods have a breakpoint capability which can optionally be enabled by specifying a stop address with the "break" argument. Generally, pods designed before the 80286-era cannot use the "break" feature. Refer to your pod manual for more specific information.

Execution of *runuutspecial* continues until one of the following events occurs:

- The pod encounters a breakpoint.
- A DCE condition occurs.
- A *haltuut* is executed.
- The time specified in a *waituut* expires.
- The RESET key is press on the operator's keypad.
- The RUN UUT HALT command is entered from the operator's keypad.

Faults that occur during the execution of *runuutspecial* are reported on the subsequent *haltuut* or *waituut*. Attempts to perform any pod-related operations except *waituut*, *haltuut*, or *polluut* results in an error if the *runuutspecial* is still active. You must execute *haltuut* or *waituut* before attempting any other pod-related operations.

Related Commands:

compare, haltuut, polluut, waituut, runuut

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.
- *Supplemental Pod Information for 9100A/9105A User's Manual.*

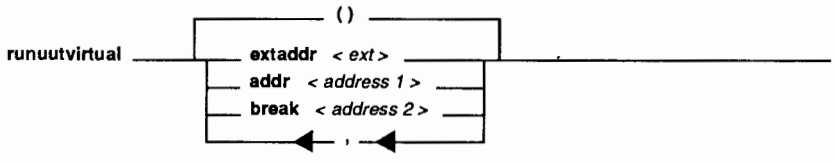
runuutvirtual
special function

IEEE-488

Syntax:

```
runuutvirtual [extadddr <ext>,] addr <address 1>
 [, break <address 2>]

runuutvirtual (<address 1>, <address 2>)

runuutvirtual ()
```

Syntax Diagram:



Description:

Runuutvirtual is a compete replacement for the obsoleted *runuutspecial* command. *Runuutvirtual* causes the UUT to begin executing instructions from its own memory, asynchronously to the system. Execution begins at the virtual address specified. If none is specified, *runuutvirtual* defaults to an address that is pod dependent.

Arguments:

ext	Extended address bits used to form virtual addresses from address 1 and address 2.
address 1	Start address.
address 2	Breakpoint address. (Default = 0)

Example 1:

```
runuutvirtual ()
     ! start at pod-dependent starting
     ! address.
```

Example 2:

```
runuutvirtual addr (read addr $FFFE)
     ! start at virtual address read from
     ! location FFFE
```

Example 3:

```
runuutvirtual addr $1235, break $2000
     !  start at hex virtual address 1235
     !  stop at hex address 2000
```

Remarks:

Some pods have special addresses where a *runuut* may begin. Such addresses may read a reset or interrupt vector from memory and begin execution at that address.

Typically, *runuutvirtual* would be followed by *waituut* with a suitable maxtime value that allows the UUT time to complete its operation.

Some pods have a breakpoint capability which can optionally be enabled by specifying a stop address with the "break" argument. Generally, pods designed before the 80286-era cannot use the "break" feature. Refer to your pod manual for more specific information.

Execution of *runuutvirtual* continues until one of the following events occurs:

- The pod encounters a breakpoint.
- A DCE condition occurs.
- A *haltuut* is executed.
- The time specified in a *waituut* expires.
- The RESET key is press on the operator's keypad.
- The RUN UUT HALT command is entered from the operator's keypad.

Faults that occur during the execution of *runuutvirtual* are reported on the subsequent *haltuut* or *waituut*. Attempts to perform any pod-related operations except *waituut*, *haltuut*, or *polluut* results in an error if the *runuutvirtual* is still active. You must execute *haltuut* or *waituut* before attempting any other pod-related operations.

Related Commands:

compare, haltuut, polluut, waituut, runuut

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.
- *Supplemental Pod Information for 9100A/9105A Users Manual.*

setbit
operator

Syntax:

```
setbit <number>
```

Syntax Diagram:

Description:

Calculates the number that results from setting the bit whose index is given by the operand.

Arguments:

number	Index on bit to set.

Example:

```
x = setbit 3    ! the variable x is set to 8
```

Remarks:

An argument value greater than 31 (decimal) causes an error.

Related Commands:

bitmask

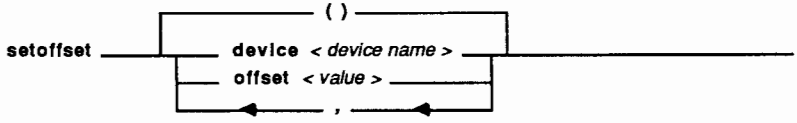
setoffset
function

IEEE-488

Syntax:

```
setoffset [device <device name>] [, offset <value>]

setoffset (<device name>, <value>)

setoffset ()
```

Syntax Diagram:



Description:

Sets the delay offset for the specified I/O module or probe device. The value entered is biased by a value of 1000000 (decimal). For example, if data is to be sampled 10 nanoseconds before the calibration point, enter a value of 999990 (decimal). The delay lines in the hardware (I/O module or probe) will be set to the closest tap possible, which provides the desired delay offset. The I/O module has a resolution of about 15 nanoseconds per tap, and the probe has a resolution of about 4 nanoseconds per tap.

Each sync mode has a separate offset associated with it. For this reason, changing sync modes will change the offset.

Arguments:

device name	I/O module name or probe name. (Default = "/probe")
value	Desired offset from calibration point. The value entered is biased by 1000000 (decimal). (Default = 1000000)

Returns:

Returns a status of 1 or 0. A one indicates that the setting of the offset was successful (within the range of the hardware). A zero indicates that when setting the offset with the current calibration value, the resulting delay line setting is out of the range of the hardware. In this case, the hardware will still be set as close as possible to the desired offset; that is, the offset will be set to either the maximum or minimum of its range, depending on the offset value.

Example 1:

```
     ! Set an offset of -10 nanoseconds for the
     ! probe in pod address sync

sync device "/probe" mode "pod"
sync device "/pod", mode "addr"
offset_value = 1000000 - 10
sts = setoffset device "/probe", offset
 offset_value
if (sts = 1) then
     print "setoffset succeeded"
else
     print "setoffset failed"
end if
```

Example 2:

```
     ! Set an offset of 0 nanoseconds for I/O
     ! module 1 in pod data sync

sync device "/mod1", mode "pod"
sync device "/pod", mode "data"
offset_value = 1000000
sts = setoffset device "/mod1", offset offset_value
if (sts = 1) then
     print "setoffset succeeded"
else
     print "setoffset failed"
end if
```

Example 3:

```
     ! Set an offset of 25 nanoseconds for I/O
     ! module 4 in ext sync

sync device "/mod4", mode "ext"
offset_value = 1000000 + 25
sts = setoffset device "/mod4", offset offset_value
if (sts = 1) then
     print "setoffset succeeded"
else
     print "setoffset failed"
end if
```

Remarks:

The *setoffset* command is valid only if the sync mode is "pod" or "ext"

In most cases, this function will not need to be used. Once the probe and I/O module have been calibrated to a particular pod, their delay lines will have been set properly for the selected sync mode. But if special situations arise, where it is desired to "move" the clock point around, this command can be used.

These offsets are always relative to an edge. The calibration procedure for the probe and I/O module finds the correct delay settings for a particular edge (for example, falling edge of ALE on the 8088 pod in address sync mode). The calibration procedure then sets an offset from that edge as defined in the pod database. When calibrating to ext, the offset is set to zero. This *setoffset* command allows other values of offset to be used, rather than these defaults.

Remarks concerning example 1 (using pod sync):

In the 80286 pod, assume that address sync is specified to occur 45 nanoseconds prior to the rising edge of signal ~S1. When calibration is performed, the hardware delay lines will be set as close as possible to this point. If getoffset is performed at this point, it will return a value of 999955 (decimal) or a value close to that, since the hardware provides delays in incremental values. If it is desired to sample a signal 20 nanoseconds before

the rising edge of ~S1, an offset of 999980 (decimal) would be used. If it is desired to sample a signal exactly on the edge of ~S1, an offset of 1000000 (decimal) would be used. And if it was desired to sample 35 nanoseconds after ~S1, an offset of 1000035 (decimal) would be used.

Remarks concerning example 3 (using external sync):

If getoffset were performed just after an external calibration, the offset would return 1000000 (decimal) or some number near 1000000. This indicates that data will be sampled on the edge of the clock signal. If it is desired to sample the signal 28 nanoseconds before the clock edge, the offset would be set to 999972 (decimal). Likewise, sampling after the clock edge would require an offset value greater than 1000000 (decimal).

The range of the *setoffset* command may vary from unit to unit. You need to be careful with programs that require large offsets. These offsets may be legal on some units but illegal on others. The recommended procedure is to determine the allowable range of offsets for a particular sync mode, and then make sure that some guard band is left. This range can be determined by selecting two very large offsets, such as +1000 nanoseconds and -1000 nanoseconds. Since these values will drive the hardware to its maximum settings, performing a *getoffset* will return the maximum and minimum allowable offset values respectively.

Related Commands:

arm, getoffset

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The "Offset Command" section of the *Programmer's Manual* for information on using offsets with GFI.

setspace function

IEEE-488

Syntax:

```
setspace space <expression>

setspace (<expression>)
```

Syntax Diagram:

Description:

Sets the address space to the specified number.

Arguments:

expression	A value returned by getspace or sysspace.

Example:

```
program test15
     s = getspace space "memory", size "byte"
     setspace space s   ! Sets the address space
                        ! parameters.
          .
          .
     s2 = sysspace ()   ! Saves the last-used
                        ! address space parameters.
     execute test12     ! Suppose the program test12
                        ! changes the address space.
     setspace space s2  ! Restores the original
                        ! address space parameters.
          .
          .
end program
```

Remarks:

Set the address space to the number returned by *sysspace* or *getspace*. It is meaningless to give *setspace* a number other than one returned by *getspace* or *sysspace*. An invalid space will cause an error.

Related Commands:

getspace, sysspace

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- Appendix I, "Pod-Related Information," in this manual.

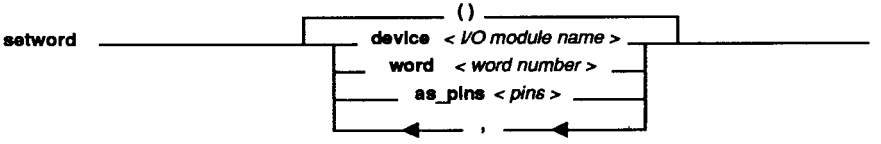
setword function

IEEE-488

Syntax:

```
setword [device <device name>] [, word <word number>]
     [, as_pins <pins>]

setword (<device name>, <word number> , <pins>)
```

Syntax Diagram:



Description:

Allows pin numbers to be grouped together to form a user defined word. *Setword* is identical to the front panel operation IOMOD SET WORD. Up to 40 unique pins may be grouped together for each I/O module in five different groups.

Arguments:

device name	I/O module name, ("/mod1", "/mod2", "/mod3", or "/mod4"). (Default = "/mod1")
word number	The number of the word group. Valid values are from 1 through 5. (Default = 1)
pins	A string of from 1 through 40 unique pin numbers. (Default = "40 39 38 37 ... 4 3 2 1")

Example 1:

```
setword device "/mod1", word 1, as_pins "1 2 3 4"
```

Example 2:

```
setword device "/mod1", word 5, as_pins "40 39 2 1"
```

Remarks:

The *setword* function is identical in operation to the front panel operation IOMOD SET WORD. The purpose of this command is to group together user related pins to form specific words for use with the *readword* and *writeword* commands.

Only unique pin numbers between 1 and 40 inclusive are valid.

Related Commands:

readword, writeword

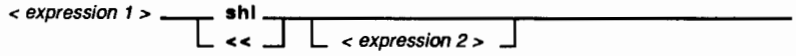
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
<expression 1> shl [<expression 2>]
```

Syntax Diagram:



Description:

Shifts the operand left by either one bit, or a specified number of bits. The first operand is shifted left: by one bit if the second operand is omitted, or by the number of bits specified by the second operand.

Arguments:

expression 1	The operand to be shifted left.
expression 2	The number of bits to shift. (Default = 1)

Returns:

The shifted number.

Example 1:

```
x = 7 shl       ! the variable x is set to E
```

Example 2:

```
x = 7 shl 2     ! the variable x is set to 1C
```

Example 3:

```
x = 12 shl 2   ! the variable x is set to 30
               ! hexadecimal
```

Remarks:

The symbol, << , also denotes the *shl* operation.

An error is raised if expression 2 is greater than decimal 31.

Since *shl* operations are carried out from right to left,

```
a shl b shl c
```

is the same as:

```
a shl (b shl c)
```

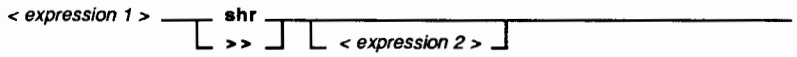
Related Commands:

shr

Syntax:

```
<expression 1> shr [<expression 2>]
```

Syntax Diagram:



Description:

Shifts the operand right by either one bit, or a specified number of bits. The first operand is shifted right: by one bit if the second operand is omitted, or by the number of bits specified by the second operand.

Arguments:

expression 1	The operand to be shifted right.
expression 2	The number of bits to shift. (Default = 1)

Returns:

The shifted number.

Example 1:

```
y = $19 shr      ! the variable y is set to C
```

Example 2:

```
y = $19 shr 3    ! the variable y is set to 3
```

Example 3:

```
y = 19 shr 3   ! the variable y is set to 2
```

Remarks:

The symbol, >> , also denotes the *shr* operation.

An error is caused if expression 2 is greater than decimal 31.

Since *shr* operations are carried out from right to left,

```
a shr b shr c
```

is the same as:

```
a shr (b shr c)
```

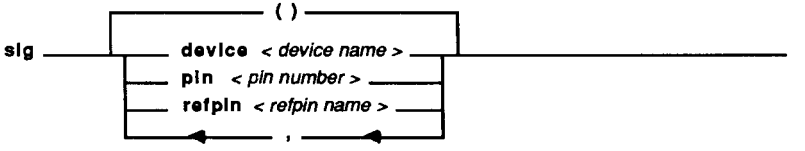
Related Commands:

shl

Syntax:

```
sig [device <device name>] [, pin <pin number>]
     [, refpin <refpin name>]

sig (<device name>, <pin number>, <refpin name>)

sig ()
```

Syntax Diagram:



Description:

Returns the signature for one pin. The signature can be requested either in terms of an I/O module pin, a component pin, or the probe. This command will return useful information only after an *arm . . . readout* block has taken a measurement.

Arguments:

device name	I/O module name, clip module name, probe name, or reference designator. (Default = "/probe")
pin number	Pin number of device specified. (Default = 1)
refpin name	Specifies the device and pin in string format. The refpin argument is used to override the device and pin values. (Default = "")

Returns:

The 16-bit signature read.

Example 1:

```
modlist = clip ref "U3", pins 40
modsig = sig device "U3", pin 12
     ! Get sig on U3 pin 12
```

Example 2:

```
modsig = sig device "/mod1", pin 12
     ! Get sig on pin 12 of I/O module 1
```

Example 3:

```
modsig = sig ("/mod1A", 12,"")
     ! Get sig on pin 12 of I/O module 1, clip A
     ! A refpin value must be supplied.
```

Remarks:

The signature can be requested for a specific pin of an I/O module by specifying the module name ("/mod1", "/mod2", etc.) as the device argument. The pin argument is interpreted as an I/O module pin. Refer to Appendix E for tables that show what I/O module pin numbers to use for every possible clip module.

If a component name ("U1", "U2", etc.) is specified as the device argument, the pin argument is interpreted as a component pin. The *sig* function determines the I/O module and pin number that corresponds to the specified component pin. The named component must have been previously named in a *clip* command.

If the string value for refpin is not a null string (""), the values of the device and pin arguments are ignored.

The *sig* function should be called only after the execution of an *arm . . . readout* block.

Related Commands:

arm, count, level, readout

For More Information:

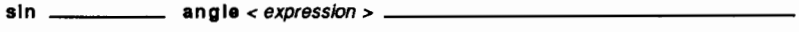
- The "Overview of TL/1" section of the *Programmer's Manual.*

sin function

Syntax:

```
sin angle <expression>

sin (<expression>)
```

Syntax Diagram:



Description:

Returns the sine function of the floating-point argument value.

Argument:

expression	The floating-point argument value, expressed in radians.

Returns:

A floating-point number

Examples :

```
f = sin ((natural pi)/ 2.0)

f = sin angle theta
```

Related Commands:

asin, natural

sqrt function

IEEE-488

Syntax:

```
sqrt num <expression>

sqrt (<expression>)
```

Syntax Diagram:

Description:

Returns the square root of a floating-point argument value.

Arguments:

expression	A floating argument value, which must be greater than or equal to 0.0.

Returns:

A floating-point number.

Examples:

```
f = sqrt  num f
f = sqrt (3.0)
```

Related Commands:

pow, log

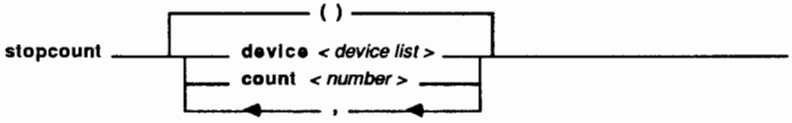
stopcount function

IEEE-488

Syntax:

```
stopcount [device <device list>] [,count <number>]

stopcount (<device list>, <number>)

stopcount ()
```

Syntax Diagram:



Description:

Sets the programmable stop count that turns off the response-gathering hardware after the specified number of enabled clock pulses. The stopcount function assigns this feature to the probe or a single I/O module and specifies the value of the stop count (1-65535).

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
number	Number of clock pulses that the response-gathering hardware should count. (Default = 1)

Example:

```
mod = clip ref "u55", pins 24
stopcount device mod, count 100
```

Remarks:

When using *stopcount*, the *edge* command is required to set the stop condition to "count".

To get the same count as that entered at the operator's keypad, use a decimal number for the stopcount.

Example: stopcount count 1000

Related Commands:

arm, edge, readout

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

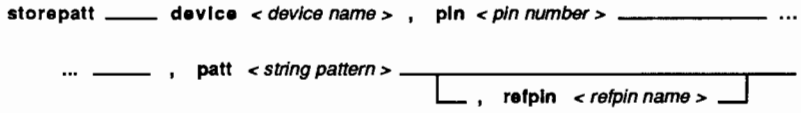
storepatt function

IEEE-488

Syntax:

```
storepatt device <device name>, pin <pin number>,
     patt <string pattern> [, refpin <refpin name>]

storepatt (<device name>, <pin number>, <string
     pattern>, <refpin name>)
```

Syntax Diagram:



Description:

Stores an arbitrary sequence of patterns that will be overdriven with the I/O module. The last pattern driven can be latched or pulsed.

Arguments:

device name	I/O module name, clip module name, or reference designator.
pin number	Pin number affected.
string pattern	String, composed of "1", "0", "X", or "x".
refpin name	Specifies the device and pin in string format. The refpin argument is used to override the device and pin values. (Default = "")

Example:

```
! driving a 7400 through its truth table

     mod = clip ref "u1", pins 14
     sync device mod, mode "int"
     clearpatt device "u1"
     storepatt device "u1", pin 1, patt "0011"
     storepatt device "u1", pin 2, patt "0101"
     storepatt device "u1", pin 4, patt "0011"
     storepatt device "u1", pin 5, patt "0101"
     storepatt device "u1", pin 9, patt "0011"
     storepatt device "u1", pin 10, patt "0101"
     storepatt device "u1", pin 12, patt "0011"
     storepatt device "u1", pin 13, patt "0101"
                .
                .
                .
     arm device mod
          writepatt device "u1", mode "latch"
     readout device mod
```

Remarks:

The *storepatt* command stores an arbitrary sequence of patterns that will be overdriven with the I/O module. The last pattern driven can be either latched or pulsed. The overdrivers are turned off either by writing a pattern that sets all pins to high-impedance, or by using the *clearoutputs* command.

You describe the pattern to be driven using a pattern string. Each string describes the pattern of highs, lows, and high-impedance states that a *single pin* should be driven through.

Pattern strings are composed of the following characters: "1" = high; "0" = low; "X" or "x" = high-impedance.

For example, to use an I/O module pin and drive it alternately high and low, you would specify the following type of pattern string: "10101010101010101010".

The pattern can be driven in the following terms: I/O module pins, pins on an I/O module clip, or reference designator pins:

For I/O module names: pin numbers are interpreted as I/O module pins. Each I/O module can have up to two clips connected. The clips are referred to as "A" and "B", depending on which end of the I/O module they are connected to.

For clip module names: pin numbers are interpreted as clip pin numbers.

For reference designator names: pin numbers are interpreted as component pins. An error is generated if the system does not recognize that the clip is connected to the component. This error occurs when the programmer does not use the *clip* command in the program.

If the string value for refpin is not a null string (""), the values of the device and pin arguments are ignored.

Related Commands:

clearoutputs, clearpatt, writepatt

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

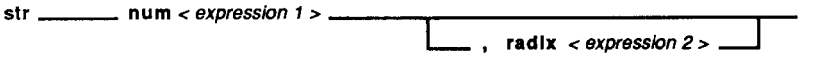
str function

IEEE-488

Syntax:

```
str num <expression 1> [, radix <expression 2>]

str (<expression 1>, <expression 2>)
```

Syntax Diagram:



Description:

Returns the string representation of the numeric operand.

Arguments:

expression 1	A numeric expression for the number to be converted into a string representation.
expression 2	A numeric expression for the radix of the number to be converted. The allowed radices are 2, 8, 10(default), or 16.

Returns:

The string representing the converted number.

Examples:

```
x = str(256,10) ! the variable x is set to the
                ! character string "256"

x = str(256,16) ! the variable x is set to the
                ! character string "100"
```

Related Commands:

fstr, val

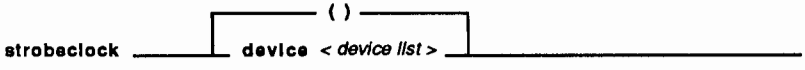
strobeclock function

IEEE-488

Syntax:

```
strobeclock device <device list>

strobeclock (<device list>)

strobeclock ()
```

Syntax Diagram:



Description:

Strobes the internal clock of the probe or the specified I/O module for clocking CRC signatures and synchronous level histories.

Arguments:

device list	I/O module name, clip module name, or probe name. (Default = "/probe")

Example:

```
mod = "/mod1"
sync device mod, mode "int"
            ! "int" mode is required for the
            ! strobeclock command
counter device mod, mode "transition"
arm device mod
     strobeclock device mod
readout device mod

crc = sig device mod, pin 1
lvl = level device mod, pin 1, type "clocked"
```

Remarks:

The "int" sync mode is required when using the *strobeclock* command.

Related Commands:

arm, readout, sync

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

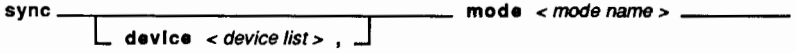
sync function

IEEE-488

Syntax:

```
sync [device <device list>,] mode <mode name>

sync (<device list>, <mode name>)
```

Syntax Diagram:



Description:

Sets the synchronization mode for the pod, the probe, or a single I/O module. The allowable modes are a pod sync, external sync, freerun sync, or internal sync.

Arguments:

device list	Pod, I/O module name, clip module name, probe name, or combinations of these. (Default = "/pod")
mode name	Pod sync modes: to match those on the pod database. Probe or I/O module sync modes: "pod", "ext", "int", or "freerun".

Example:

```
mod = clip ref "u1", pins 16
sync device mod, mode "pod"

sync device "/pod", mode "addr"

sync device "/probe", mode "freerun"
```

Remarks:

When using the "pod" mode for the *sync* command, you must tell the pod what kind of pod sync signal should be generated. This is done with a second *sync* command:

```
sync device "/probe", mode "pod"
sync device "/pod", mode "data"
```

Four I/O module or probe sync modes are allowed: external sync, pod sync, freerun sync, and internal sync.

External Sync Mode

This mode qualifies the external Clock line with the external measurement control Start, Stop, and Enable lines. For the probe, the lines are available through the clock module. The I/O module has its own measurement control lines.

Start, Stop, and Clock are edge-sensitive inputs from the UUT. Each can be made to respond to falling or rising edges. The sync period can also be programmed to end after a specified number of valid clock pulses, in which case the Stop input is ignored. Enable is a level-sensitive signal and can be specified by the *enable* command.

After an *arm* command is entered, and after a valid Start edge is detected, the sync measurement period begins. Start is recognized independently of the enabling condition, but data is only gathered after the enabling condition becomes true. Asynchronous data is gathered immediately after this point. Synchronous data is gathered after the same point but only at the selected clock edges.

The data gathering period ends when the Stop condition becomes true: the selected Stop edge occurs, or a programmed number of clock edges completes. The data-gathering period ends when a *readout* command is executed.

In addition, this synchronization mode can be used to synchronize the probe's pulsing output resulting from the *pulser* command.

Pod Sync Mode

This mode uses Pod Sync, an internal pod signal, as the clock. The generation of Pod Sync can be made to depend on valid address, data, or other (pod-dependent) cycles. The external measurement control lines are ignored.

Data is gathered after an *arm* command until a *readout* command is executed.

In addition, this synchronization mode can be used to synchronize the probe's pulsing output resulting from the *pulser* command.

Freerun Sync Mode

In this mode, the probe uses the 9100A/9105A system's internal 1k Hz clock for asynchronous output. The external Start, Stop, Clock, and Enable lines are ignored.

For both the I/O modules and the probe, asynchronous level history and transition count data can be gathered between the *arm* and *readout* commands. CRC signatures and clocked level histories are not gathered.

In addition, this synchronous mode can be used to synchronize the probe's pulsing output resulting from the *pulser* command.

Internal Sync Mode

This sync mode is designed to use a clock signal generated internally be the *strobeclock* command. The external measurement control lines (Start, Stop, Clock, and Enable) are ignored.

In addition, this synchronous mode can be used to synchronize the probe's pulsing output resulting from the *pulser* command.

Related Commands:

arm, checkstatus, connect, enable, readout, strobeclock

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

sysaddr function

IEEE-488

Syntax:

```
sysaddr ()
```

Syntax Diagram:

Description:

Returns the last address written to or read from.

Returns:

The last address written to or read from.

Example:

```
lastaddr = sysaddr ()   ! stores the last address
                        ! written to or read from in
                        ! the variable lastaddr.

write addr sysaddr (), data $34
```

Remarks:

The value returned by *sysaddr* is used as the default address for the next read or write initiated from the operator's keypad.

In some cases, the last address written to or read from does not update *sysaddr*.

Related Commands:

sysspace

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

sysdata function

IEEE-488

Syntax:

```
sysdata ()
```

Syntax Diagram:

Description:

Returns the last data read or written.

Returns:

The last data read or written.

Example:

```
lastdata = sysdata ()    ! stores the last data read
                         ! in the variable lastdata

write data sysdata (), addr next
```

Remarks:

The value returned by *sysdata* is used as the default data for the next read or write initiated from the operator's keypad.

In some cases, the last data written or read does not update *sysdata*.

Related Commands:

sysspace

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
sysinfo get <attribute name>
```

Syntax Diagram:

Description:

The *sysinfo* command quieries information about the system. Each system attribute which can be manifulated, has a unique name. Currently, only the following system attribute can be determined:

/system/version	The string describing the system software release version (for example, "6.0").
/system/model	The string describing the system model (for example, "9100", "9105", "9100FT", or "9105FT").

When invoked with the 'get' argument, *sysinfo* returns a string representing the value of the requesting attribute.

Arguments:

get	The name of the attribute to retrieve.

Example:

The following retrieves the system software version string:

```
s = sysinfo get "/system/version"

s = sysinfo get "/system/model"
```

Remarks:

This command first appeared in the 6.0 software release.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

sysspace function

IEEE-488

Syntax:

```
sysspace ()
```

Syntax Diagram:

Description:

Returns the number associated with the last address space accessed.

Returns:

The number associated with the last address space accessed.

Examples:

```
program test15
     s = getspace space "memory", size "byte"
     setspace space s   ! Sets the address space
                        ! parameters.
          .
          .
     s2 = sysspace ()   ! Saves the last-used
                        ! address space parameters.
     execute test12     ! Suppose the program test12
                        ! changes the address space.
     setspace space s2  ! Restores the original
                        ! address space parameters.
          .
          .
end program
```

Related Commands:

getspace, setspace, sysaddr, sysdata

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

systime function

IEEE-488

Syntax:

```
systime ()
```

Syntax Diagram:

Description:

Returns the number of seconds that have elapsed since a particular date (January 1, 1980). This number alone is generally not useful. However, the difference between the numbers returned by two invocations of *systime* yields the elapsed time in seconds between the two invocations. The *systime* function also provides the argument for the *readdate* and *readtime* functions that produce the current date and time.

Returns:

The number of elapsed seconds.

Example:

```
start = systime ()
testramfull addr 0, upto $7FFF
finish = systime ()
print "Full RAM test took ", finish - start,
 " seconds"

     ! prints: "Full RAM test took 368 seconds"
     ! if, testramfull requires six minutes and
     ! eight seconds to execute
```

Related Commands:

readdate, readtime

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual*.

tan function

IEEE-488

Syntax:

```
tan angle <expression>

tan (<expression>)
```

Syntax Diagram:

Description:

Returns the tangent function of a floating-point argument value.

Argument:

expression	The argument (floating-point) value in radians.

Returns:

A floating-point number.

Examples:

```
f = tan ((natural pi)/4.0)
f = tan angle theta
```

Related Commands:

atan, natural

testbus function

IEEE-488

Syntax:

```
testbus addr <address>

testbus (<address>)
```

Syntax Diagram:

Description:

Checks the address, data, and control lines for drivability.

Argument:

address	Readable/writable address used for data bus testing.

Example:

```
testbus addr $FFFF
```

Remarks:

The *testbus* test performs the following checks:

- Tests for control-line drivability.
- Tests for address drivability and tied address lines.
- Tests for data drivability and tied data lines.

The specified address must be a RAM location to prevent erroneous data line faults from being reported.

It is not a good idea to use *testbus* as a stimulus in GFI. The algorithm for this functional test could change and this would also change any resulting signatures. The commands *toggleaddr*, *toggledata*, *rampaddr*, and *rampdata* should be used instead.

Related Commands:

fails, *passes*

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

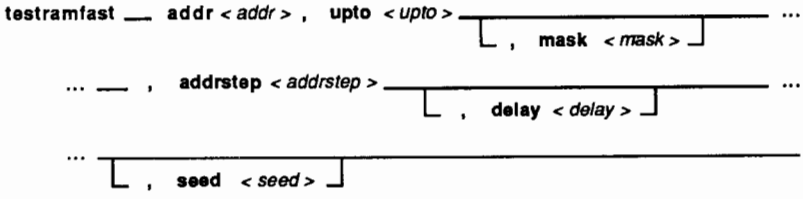
testramfast
function

IEEE-488

Syntax:

```
testramfast addr <addr>, upto <upto> [, mask
     <mask>], addrstep <addrstep> [, delay <delay>]
     [, seed <seed>]

testramfast (<addr>, <upto>, <mask>, <addrstep>,
     <delay>, <seed>)
```

Syntax Diagram:



Description:

Performs a probabilistic test on RAM.

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask of testable data bits. (Default = $FFFFFFFF)
addrstep	Address increment.
delay	Milliseconds to delay between sweeps. (Default = 250)

seed	Number to be used as seed for pseudo-random number generator. (Default = 0; seed based on real-time clock.)

Example 1:

```
testramfast addr $B, upto here, addrstep 2
     ! address hex B to address in variable
     ! "here"
```

Example 2:

```
testramfast addr $1234, upto $12FE, mask $7F,
 addrstep 2
     ! only the lower 7 data bits are tested
```

Remarks:

The fast RAM test is a probabilistic test which has complete coverage of a wide range of common faults and very thorough coverage of nearly every possible RAM fault. Each word is accessed five times using pseudo-random data.

The coverage of *testramfast* is shown on the next page. "P" represents the probability that a fault is *not* detected. "K" represents the number of addresses affected by the fault. The values which appear in parentheses are the actual probabilities for a 16K memory. Larger memories have better fault coverage.

testramfast Coverage

Fault Condition	*Coverage*
Stuck cells	Always found.
Aliased cells	Always found.
Stuck address lines	Always found.
Stuck data lines	Always found.
Shorted address lines	Always found.
Shorted data lines	$P = 0.5^K$.
Multiple selection decoder	$P = 0.5^K (2.9 \times 10^{-39})$ for a row or column decoder.
Dynamic coupling	$P = 0.75^K$ (0.75 for 1 cell coupling to 1 other cell).
Aliasing between bits in same word	$P = 0.5^K$.
Refresh problems	Always detected if the delay is sufficiently long and standby reads do not mask the problem.
Pattern sensitive faults	Because of the random nature of this test, some pattern sensitive faults can be detected.

The seed parameter controls the exact sequence of pseudo-random values which are generated. When the seed argument is zero, the 9100A/9105A generates a different sequence of pseudo-random data at each pass through the memory. You should specify zero in most applications.

The mask parameter provides a convenient method for testing the parity bit of memories so equipped. First you perform a *testramfast* test with a mask that tests all valid data bits. Then you do a second *testramfast* test, but use a mask with an odd number of bits set. In addition, this second test is done with a line of a clip module connected to the output of the memory parity tree. The DCE condition is generated by using a *compare*

command with a compare string only one bit long, which checks for the desired logic level at the parity tree output.

The procedure for testing memories with error-correcting codes is similar, except that you should choose a mask such that toggling the bits specified by the mask will cause all the check bits to toggle.

Related Commands:

fails, passes, testramfull

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

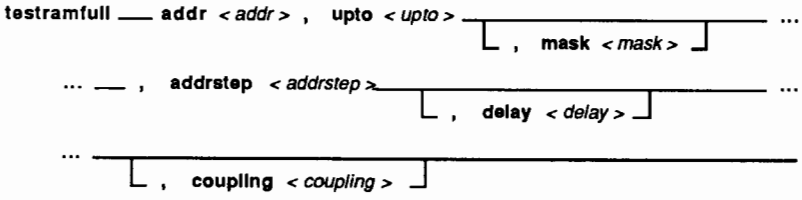
testramfull
function

IEEE-488

Syntax:

```
testramfull addr <addr>, upto <upto> [, mask
     <mask>], addrstep <addrstep> [, delay <delay>]
     [, coupling <coupling>]

testramfull (<addr>, <upto>, <mask>, <addrstep>,
     <delay>, <coupling>)
```

Syntax Diagram:



Description:

Performs a deterministic test of RAM functionality. It couples a test for static and dynamic coupling of cells in the same data word with a comprehensive Suk and Reddy B-test (a standard algorithm for testing memory).

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask of valid data bits. (Default = $FFFFFFFF)
addrstep	Address increment.

delay	Milliseconds to delay between sweeps. (Default = 250)
coupling	"on" or "off". (Default = "off")

Example 1:

```
testramfull addr $B, upto here, addrstep 2
     ! address hex B to variable address "here"
```

Example 2:

```
testramfull addr $1234, upto $13FF, mask $7F,
 addrstep 2
```

Remarks:

The full RAM test is a deterministic test of RAM functionality. With coupling disabled, each word is accessed 17 times. With coupling enabled, each word is accessed 29 times for 8-bit spaces, 33 times for 16-bit spaces, and 37 times for 32-bit spaces.

This test has been found to be superior to *testramfast* in finding faults caused by electrical transients in the bus where worst-case fault detection occurs when writing data with all ones or all zeros. Fault coverage for *testramfull* is shown on the next page.

testramfull Coverage

Fault Condition	*Coverage*
Stuck cells	Always found.
Aliased cells	Always found.
Stuck address lines	Always found.
Stuck data lines	Always found.
Shorted address lines	Always found.
Shorted address lines	Always found if coupling is enabled or in buses through which the first or last address must pass.
Multiple selection decoder	Always found.
Dynamic coupling	Always found.
Aliasing between bits in same word	Always found if coupling is enabled.
Refresh problems	Always found if delay is sufficiently long and standby reads do not mask the problem.
Pattern sensitive faults	Not found.

The delay parameter specifies the amount of time that the test waits between sweeps. This delay can be increased to find faults related to the dynamic memory refresh circuit.

The mask parameter provides a convenient method for testing the parity bit of memories so equipped. First you perform a testramfull test with a mask that tests all valid data bits. Then you do a second *testramfull* test, but use a mask with an odd number of bits set. In addition, this second test is done with a line of a clip module connected to the output of the memory parity tree. The DCE condition is generated by using a *compare* command with a compare string only one bit long, which checks for the desired logic level at the parity tree output.

The procedure for testing memories with error-correcting codes is similar, except that you should choose a mask such that toggling the bits specified by the mask will cause all the check bits to toggle.

The coupling test verifies that, for every word, each pair of bits in that word can store opposite values. For many memory systems, the coupling test will not be necessary since the pre-test checks for this with the first and last addresses, even with coupling disabled. However, for some memory systems, enabled coupling will be required in order to have a fully conclusive test.

You can specify an exhaustive check for coupling between bits in the same word (shorted data bus lines manifest as this fault at every address) by enabling coupling. For an eight-bit space the values 55, AA, 33, CC, 0F, and F0 are written to every word and read back. Wider spaces are similar.

Related Commands:

fails, passes, testramfast

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

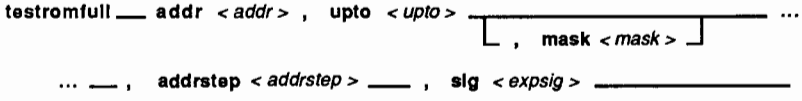
testromfull
function

IEEE-488

Syntax:

```
testromfull addr <addr>, upto <upto> [, mask
     <mask>], addrstep <addrstep>, sig <expsig>

testromfull (<addr>, <upto>, <mask>, <addrstep>,
     <expsig>)
```

Syntax Diagram:



Description:

Verifies that the signature data contained by a range of ROM matches the signature obtained from a correctly programmed ROM.

Arguments:

addr	Starting address.
upto	Ending address.
mask	Bit mask of valid data bits. (Default = $FFFFFFFF)
addrstep	Address increment.
expsig	Expected signature.

Example 1:

```
testromfull addr 0, upto $7FF, addrstep 1, sig
 $B826
```

Example 2:

```
testromfull addr first, upto last, mask $7C,
 addrstep 4, sig $31BC
```

Remarks:

The ROM test (*testromfull*) verifies that the signature data contained by a range of ROM matches the signature obtained from a correctly programmed ROM. If the measured signature does not match the expected signature, a diagnostic routine attempts to locate the fault.

As with all signature-based schemes, there is some probability (over the space of possible faults) that a faulty ROM will still have the correct signature. Only one in 65536 randomly chosen faults are missed. However, all faults that are confined to one bit-slice and 16 or fewer consecutive addresses are reported.

Should the signature be wrong, a diagnostic attempts to ascertain a likely cause. The following conditions may be reported as possible problems:

- A bit slice all zeros.
- A bit slice all ones.
- Two bit slices identical.
- An address bit ineffective.

Related Commands:

fails, getromsig, passes

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

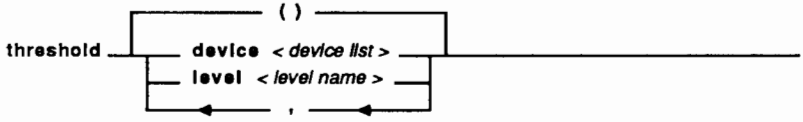
threshold
function

IEEE-488

Syntax:

```
threshold [device <device list>] [, level <level
     name>]

threshold (<device list>, <level name>)

threshold ()
```

Syntax Diagram:



Description:

Sets the input threshold levels for the probe or an I/O module. The allowable levels are "ttl", "cmos", or (for the probe only) "rs232". "ecl" is allowed for the probe if ecl capability is installed.

Arguments:

device list	I/O module name, clip module name, probe name, or combinations of these. (Default = "/probe")
level name	I/O module threshold levels "ttl" or "cmos". Probe threshold levels: "ttl", "cmos", or "rs232" if ecl capability is not installed.

Probe threshold levels: "ttl", "cmos", "rs232", or "ecl" if ecl capability installed.

Example 1:

```
mod = clip device "a1", pins 24
threshold device mod, level "ttl"
```

Example 2

```
threshold ("/mod1", "ttl")
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

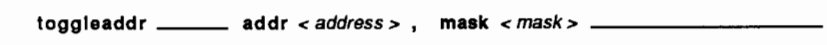
toggleaddr function

IEEE-488

Syntax:

```
toggleaddr addr <address>, mask <mask>

toggleaddr (<address>, <mask>)
```

Syntax Diagram:



Description:

Performs a series of *read* functions to stimulate the microprocessor's address bus. For each 1 in the mask, two accesses are performed: one at the address with the masked bit set and one at the address with the masked bit cleared.

Arguments:

address	Address.
mask	Bit mask of address bits to toggle.

Example:

```
toggleaddr addr $1004, mask $25
! addr 1004 hex = 0001 0000 0000 0100
! mask 25 hex = 0000 0000 0010 0101
```

The system performs a series of six *reads* as shown below:

```
read addr $1004
read addr $1005

read addr $1004
read addr $1000

read addr $1004
read addr $1024
```

Remarks:

Accesses are made in pairs for each bit in the mask, first at the original address and then at the address with the bit toggled.

The number of *read's* is equal to twice the number of bits set in the mask. The data read is not returned.

A 9000-series toggle address (ATOG) performs the equivalent of a *toggleaddr* for a single bit. Therefore, the following pairs of commands are equivalent:

9100A/9105A	*9010*
toggleaddr addr $100, mask 1	ATOG @ 100 Bit 0
toggleaddr addr $100, mask $10	ATOG @ 100 Bit 4

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

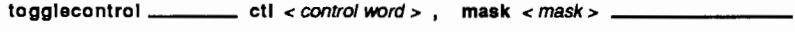
togglecontrol function

IEEE-488

Syntax:

```
togglecontrol ctl <control word>, mask <mask>

togglecontrol (<control word>, <mask>)
```

Syntax Diagram:



Description:

Performs a series of *writecontrols* to stimulate the microprocessor's control bus. For each 1 in the mask, two *writecontrols* are performed: one with the masked bit set, and one with the masked bit cleared.

Arguments:

control word	Control word.
mask	Mask of control bits to toggle.

Example:

```
togglecontrol ctl 1, mask $41
```

The system performs two pairs of *writecontrols*:

```
writecontrol ctl 1
writecontrol ctl 0

writecontrol ctl 1
writecontrol ctl $41
```

Remarks:

Writes are performed in pairs for each bit in the mask, first with the original control word and then with the toggled value.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

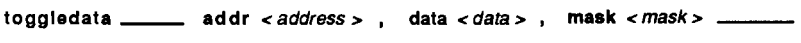
toggledata function

IEEE-488

Syntax:

```
toggledata addr <addr>, data <data>, mask <mask>

toggledata (<addr>, <data>, <mask>)
```

Syntax Diagram:



Description:

Performs a series of *writes* to stimulate the microprocessor's data bus. For each 1 in the mask, two *writes* are performed: one with the masked data bit set and one with the masked data bit cleared.

Arguments:

addr	Address.
data	Data value.
mask	Mask of bits to toggle.

Examples:

```
toggledata addr $EED0, data $AA, mask $55
```

The system performs:

```
write addr $EED0, data $AA
write addr $EED0, data $AB

write addr $EED0, data $AA
write addr $EED0, data $AE
```

(example is continued on the next page)

```
write addr $EED0, data $AA
write addr $EED0, data $BA

write addr $EED0, data $AA
write addr $EED0, data $EA
```

Remarks:

Writes are performed in pairs for each bit in the mask, first with the original data and then with the toggled data.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

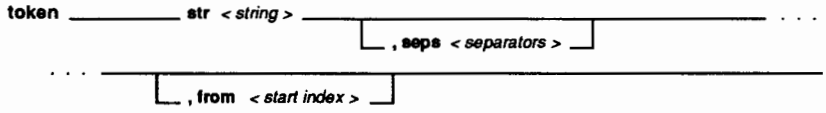
token function

IEEE-488

Syntax:

```
token str <string> [ , seps <separators> [ , from
 <start index> ]


token (<string>, <seps>, <start index>)
```

Syntax Diagram:



Description:

Implements token scanning from strings, where the tokens can be distinguished by the presence of one or more separator characters.

Arguments:

string	The string expression that is being scanned for a token.
separators	A string containing the set of separator characters that can separate token values. Each character in the set is considered equivalent to all other characters in the set. No character from the separator set is included in any returned token string value. The default set of separator characters is (newline, linefeed, space, tab).

start index — The index in the string to start scanning for the next token substring, where an index value of 1 corresponds to the first character in the string. If the start index is not supplied, scanning starts at the beginning of the string unless the previous call to the token command supplied the same string argument. In that case, scanning continues at the position in the string immediately following the last returned token.

Returns:

The next and largest possible substring from the source string that does not contain any characters from the set of separators, skipping over any initial separator prefix sequence. If there are no more substrings that do not contain non-separator characters, then an empty string is returned.

Examples:

Given an arbitrary input string in_string, extract the "words" (where a word is any non-white- space sequence of characters), assigning up to 10 words to the elements of a string array:

```
declare string array [1:10] words
! extract the first word
next_word = token str in_string, from 1
i = 1
loop while (next_word <> "")

      if (i <= 10) then
            words [i] = next_word

            ! extract the next word
            next_word = token str in_string

      else
            next_word = ""
      end if

      i = i + 1
end loop
```

Suppose an IEEE-488 device outputs a single floating-point number string value, with an arbitrary number of leading spaces and a carriage return, linefeed character sequence after each value. The following program example strips the whitespace (the spaces, carriage return and linefeed), then converts the remaining string to a floating-point number. If the device is opened with the termination character of the channel set to linefeed, the problem reduces to stripping the spaces and the carriage return:

```
declare numeric ieee_chan
declare numeric term_chan
declare string val_string
declare floating val_number

term_chan = open device "/term1"
ieee_chan = open device "/ieee/1", term "\0A"
print on ieee_chan, "val?"
input on ieee_chan, val_string
val_string = token str val_string, from 1
! strip whitespace
if (isflt(val_string)) then
      val_number = fval(val_string)
      print on term_chan,
      "value is ", val_number
else
      print on term_chan,
      "received a bogus value: ", val_string
end if
```

Suppose an IEEE-488 device outputs analog measurements in the following format:

```
chan(<nn>):<ss> <ffff...>
```

where <nn> represents a "channel number" in radix 10, <ss> represents a measurement status in radix 16, and <ffff...> is a string representation of the floating-point measurement value.

The following TL/1 fragment extracts these various fields, given the initial input string in the string variable meas_string:

```
meas_seps = "(): "    ! the set of separators
! skip the "chan" substring
token (meas_string, meas_seps, 1)
```

```
! extract the channel number
chan_string = token str meas_string, seps
  meas_seps
if (isval(chan_string, 10)) then
      chan_num = val(chan_string, 10)
else
      ! measurement string syntax error
      ! detected
      return(0)
endif

! extract the measurement status
status_string = token str meas_string,
  seps meas_seps
if (isval(status_string, 16)) then
      meas_status = val(status_string, 16)
else
      !measurement string syntax error
      !detected
      return(0)
endif

! extract the measurement value
value_string = token str meas_string, seps
  meas_seps
if (isflt(value_string)) then
      meas_value = fval(value_string)
else
      ! measurement string syntax error
      ! detected
      return(0)
endif
```

Remarks:

The *token* command is useful for separating the fields of an input string from one another. Given a particular input string, *token* can be called repeatedly to extract the fields of the string.

Related Commands:

isval, isflt

val
function

IEEE-488

Syntax:

```
val str <string> [, radix <radix>]
val (<string>, <radix>)
```

Syntax Diagram:

Description:

Calculates the numeric value of the string operand using the specified radix.

Arguments:

string	A string which represents a number.
radix	A numeric expression for the radix to use for the returned number. Allowed values for radix are 2, 8, 10 (default), and 16.

Returns:

A numeric value.

Example:

```
x = val ("15",16) ! the variable x is set to
                  ! the numeric value of
                  ! hexadecimal 15

x = val ("15",10) ! the variable x is set to
                  ! the numeric value of decimal 15
```

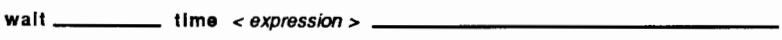
Related Commands:

fval, str, isval

Syntax:

```
wait time <expression>

wait (<expression>)
```

Syntax Diagram:



Description:

Causes program execution to pause for the specified number of milliseconds.

Argument:

expression	Approximate length of time in milliseconds.

Example:

```
wait time 1000
          ! Generate a one-second pause (plus or
          ! minus about 50 milliseconds)
```

Remarks:

You must use *wait* commands carefully since they can cause a program to pause indefinitely. The *wait* command cannot be used to generate exact timing. The precision of the timer is approximately 50 milliseconds; longer waits may result if the 9100A/9105A must service interrupts or perform other operations.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

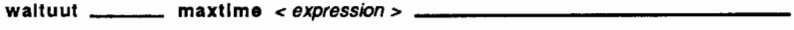
waituut function

IEEE-488

Syntax:

```
waituut maxtime <expression>

waituut (<expression>)
```

Syntax Diagram:



Description:

Suspends TL/1 program execution until one of the following conditions occurs:

- The pod encounters a breakpoint.
- The DCE condition occurs.
- The number of milliseconds specified by the expression expires.

Argument:

expression	A numeric expression for the maximum timeout limit value in milliseconds.

Example:

```
runuut addr $FF00
waituut maxtime 4000    ! suspends TL/1 program
                        ! execution for 4 seconds
                        ! while the UUT is controlled
                        ! by its own program
```

Remarks:

After executing *runuut*, you must invoke either *haltuut* or *waituut* to regain control of the pod before executing other statements that send commands to the pod.

The command *waituut (0)* is equivalent to *haltuut ()*.

The *waituut* command is usually preferred over continuous looping using *polluut* because it frees the 9100A/9105A processor for other functions.

Related Commands:

haltuut, polluut, runuut

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

winctl
function

Syntax:

```
winctl channel <channel expression>, position
     <position>
```

Syntax Diagram:

Description:

Controls the window position for the specified channel.

Arguments:

channel expression	A numeric expression to define a channel opened to write on the desired window. Remember that /term1 and /term2 are also considered windows.	
position	The position is one of the following:	
	"front"	The window covers all other windows.
	"back"	The window is covered by all other windows.
	"hide"	The window is invisible.
	"unhide"	The window becomes visible.

Example 1:

```
winctl channel chan1, position "front"
```

Example 2:

```
winctl channel chan2, position "hide"
```

Remarks:

A window in the front covers all other windows. A window in the back is covered by all other windows. A window when hidden becomes invisible, but remains in the same position on the screen.

Related Commands:

open

For More Information:

- "The Overview of TL/1" section of the *Programmer's Manual.*

write
function

IEEE-488

Syntax:

```
write addr <address>, data <data>

write (<address>, <data>)
```

Syntax Diagram:

Description:

Writes the specified data to the specified address.

Arguments:

address	Address at which to write.
data	Data to write.

Examples:

```
write addr $5567, data $21        ! writes hex 21 at
                                  ! hex address 5567

write data $34, addr $CDD3FF      ! writes hex 34
                                  ! at hex address
                                  ! CDD3FF
```

Remarks:

Refer to the pod manual for the microprocessor you are using to find specific address and data formats.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

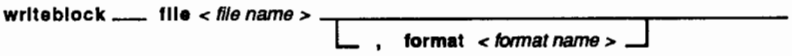
writeblock function

IEEE-488

Syntax:

```
writeblock file <file name> [, format
     <format name>]

writeblock (<file name>, <format name>)
```

Syntax Diagram:



Description:

Loads the contents of a file in a standard ASCII form (Motorola S-Record format or Intel Hex format) into UUT or pod overlay RAM. The file contains information about the starting address and number of data bytes.

Arguments:

file name	The name of the file containing the required data.
format name	The ASCII format in which the data was previously stored. Either "motorola" or "intel". (Default = "motorola".)

Examples:

```
writeblock file "test_lcd", format "motorola"

writeblock ("pgm1", "motorola")
```

Related Commands:

loadblock, readblock

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

IEEE-488

Syntax:

```
writecontrol ctl <control word>

writecontrol (<control word>)
```

Syntax Diagram:

Description:

The *writecontrol* command writes the specified data to the control lines of the pod.

Arguments:

control word	Control word.

Examples:

```
writecontrol ($A1)

writecontrol ctl $13
```

Remarks:

Not all control lines are writable; the lines that are writeable depend on the type of pod you are using. The lines are asserted for a moment while drivability is tested. Refer to your pod manual for more information.

Related Commands:

readstatus

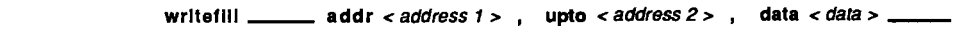
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

Syntax:

```
writefill addr <address 1>, upto <address 2>, data
     <data>

writefill (<address 1>, <address 2>, <data>)
```

Syntax Diagram:



Description:

The *writefill* command writes the specified data to each address within the specified address range.

Arguments:

address 1	Starting Address.
address 2	Ending Address.
data	Data value.

Examples:

```
writefill addr $1000, upto $1FFF, data $21

writefill (0, $7F, $123D567E)
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

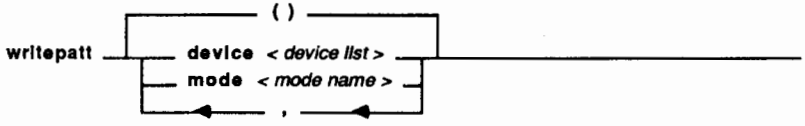
writepatt function

IEEE-488

Syntax:

```
writepatt [device <device list>] [, mode <mode
     name>]

writepatt (<device list>, <mode name>)

writepatt ()
```

Syntax Diagram:



Description:

Overdrives the specified sequence of patterns through the I/O module. The pattern to be written to each pin is stored beforehand using the *storepatt* command.

Arguments:

device list	I/O module name, clip module name, reference designator, or combinations of these. (Default = "/mod1")
mode name	"pulse", or "latch". (Default = "latch")

Example:

```
! driving a 7400 through its truth table:

mod = clip ref "U1, pins 14
clearpatt device "U1"
storepatt device "U1", pin 1, patt "0011"
storepatt device "U1", pin 2, patt "0101"
storepatt device "U1", pin 4, patt "0011"
storepatt device "U1", pin 5, patt "0101"
storepatt device "U1", pin 9, patt "0011"
storepatt device "U1", pin 10, patt "0101"
storepatt device "U1", pin 12, patt "0011"
storepatt device "U1", pin 13, patt "0101"
                .
                .
        ! internal sync mode required for writepatt
sync device mod, mode "int"
arm device mod
     writepatt device "U1", mode "latch"
readout device mod
```

Remarks:

The *writepatt* command requires that the sync mode be set to "int".

The maximum pattern depth for *writepatt* for each pin depends on the number of I/O modules used:

Number of I/O Modules Used	Maximum Pattern Depth
1	255
2	128
3	85
4	64

Appendix E, "I/O Module Clip/Pin Mapping," shows, for each clip module, which I/O module pin is connected to each clip pin.

Related Commands:

storepatt, clearpatt, clearoutputs

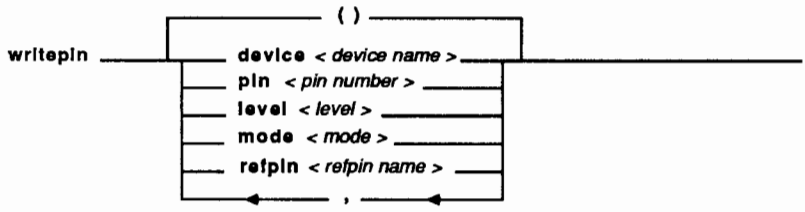
For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Syntax:

```
writepin [device <device name>] [, pin <pin
     number>] [, level <level>] [, mode <mode>]
     [, refpin <refpin name>]

writepin (<device name>, <pin number>, <level>,
     <mode>, <refpin name>)

writepin ()
```

Syntax Diagram:



Description:

Sets the specified I/O module pin to the desired state by either latching or pulsing.

Arguments:

device name	I/O module name, clip module name, or reference designator. (Default = "/mod1")
pin number	I/O module pin number. (Default = 1)
level	"1" for high, "0" for low, and "X" or "x" for high-impedance. (Default = "0")

mode	"latch" or "pulse". (Default = "latch")
refpin name	Specifies the device and pin in string format. The refpin argument is used to override the device and pin values. (Default = "")

Example 1:

```
                ! Latching I/O module #1, pin #40 HIGH

writepin device "/mod1," pin 40, level "1", mode
 "latch"
```

Example 2:

```
                ! Pulsing I/O module #2, pin #20 LOW

writepin device "/mod2", pin 20, level "0", mode
 "pulse"
```

Example 3:

```
writepin refpin "U26-F", level "1", mode "pulse"
```

Remarks:

If the string value for refpin is not a null string (""), the values of the device and pin arguments are ignored.

Appendix E, "I/O Module Clip/Pin Mapping," shows, for each clip module, which I/O module pin is connected to each clip pin.

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

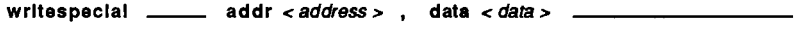
writespecial function

IEEE-488

Syntax:

```
writespecial addr <address>, data <data>

writespecial (<address>, <data>)
```

Syntax Diagram:



Description:

Writes the specified data to the specified virtual address. This allows access to the virtual addresses that, in some pods, are used for special operations. This command should only be used when you know that the normal *write* command does not provide the required special operation.

Arguments:

address	The virtual address where data will be written.
data	Data to write.

Examples:

```
writespecial addr $F0000018, data $21
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

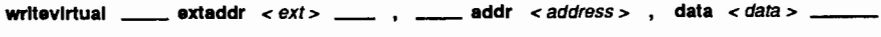
writevirtual function

IEEE-488

Syntax:

```
writevirtual extaddr <ext>, addr <address>, data
      <data>

writevirtual (<ext>, <address>, <data>)
```

Syntax Diagram:



Description:

Writevirtual is a complete replacement for the obsoleted *writespecial* command. The *writevirtual* command writes the specified data to the specified virtual address. This allows access to the virtual addresses that, in some pods, are used for special operations. This command should only be used when you know that the normal *write* command does not provide the required special operation.

Arguments:

extaddr	Extended address bits.
address	The virtual address where data will be written.
data	Data to write.

Examples:

```
writevirtual extaddr 0, addr $F0000018, data $21
```

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*
- The Fluke pod manual for the microprocessor you are using.

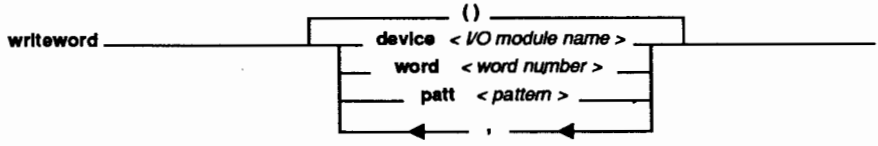
writeword function

IEEE-488

Syntax:

```
writeword [device <I/O module name>] [ , word <word
     number>] [ , patt <pattern>]

writeword (<I/O module name> , <pattern> , <word number>)
```

Syntax Diagram:



Description:

Writes a data pattern to a group of I/O module pins. The group of I/O module pins is set using *setword*.

Arguments:

I/O module name	I/O module name ("/mod1", "/mod2", "/mod3", or "/mod4"). (Default = "/mod1")
word number	This specifies the pin grouping to use in writing out the word. (Default = 1)
pattern	Specifies the levels to be driven. Valid values are 1 (HIGH), 0 (LOW), and X or x (driver off, 3-stated) (Default = "0")

Example 1:

```
writeword device "/mod1", word 4, patt "10X"
```

Example 2:

```
writeword device "/mod2", word 1, patt "000000000000"
```

Remarks:

If not enough levels have been specified in the pattern, they are assumed to be LOW. If too many values are specified, the leading values are ignored.

Related Commands:

readword, setword

For More Information:

- The "Overview of TL/1" section of the *Programmer's Manual.*

Appendix A
ASCII Codes

CHR	HEX	DEC	CHR	HEX	DEC	CHR	HEX	DEC	CHR	HEX	DEC
NUL	0	0	SP	20	32	@	40	64	`	60	96
SOH	1	1	!	21	33	A	41	65	a	61	97
STX	2	2	"	22	34	B	42	66	b	62	98
ETX	3	3	#	23	35	C	43	67	c	63	99
EOT	4	4	$	24	36	D	44	68	d	64	100
ENQ	5	5	%	25	37	E	45	69	e	65	101
ACK	6	6	&	26	38	F	46	70	f	66	102
BEL	7	7	'	27	39	G	47	71	g	67	103
BS	8	8	(	28	40	H	48	72	h	68	104
HT	9	9	)	29	41	I	49	73	i	69	105
LF	A	10	*	2A	42	J	4A	74	j	6A	106
VT	B	11	+	2B	43	K	4B	75	k	6B	107
FF	C	12	,	2C	44	L	4C	76	l	6C	108
CR	D	13	-	2D	45	M	4D	77	m	6D	109
SO	E	14	.	2E	46	N	4E	78	n	6E	110
SI	F	15	/	2F	47	O	4F	79	o	6F	111
DLE	10	16	0	30	48	P	50	80	p	70	112
DC1	11	17	1	31	49	Q	51	81	q	71	113
DC2	12	18	2	32	50	R	52	82	r	72	114
DC3	13	19	3	33	51	S	53	83	s	73	115
DC4	14	20	4	34	52	T	54	84	t	74	116
NAK	15	21	5	35	53	U	55	85	u	75	117
SYN	16	22	6	36	54	V	56	86	v	76	118
ETB	17	23	7	37	55	W	57	87	w	77	119
CAN	18	24	8	38	56	X	58	88	x	78	120
EM	19	25	9	39	57	Y	59	89	y	79	121
SUB	1A	26	:	3A	58	Z	5A	90	z	7A	122
ESC	1B	27	;	3B	59	[	5B	91	{	7B	123
FS	1C	28	<	3C	60	\	5C	92	|	7C	124
GS	1D	29	=	3D	61	]	5D	93	}	7D	125
RS	1E	30	>	3E	62	^	5E	94	~	7E	126
US	1F	31	?	3F	63	_	5F	95	RUB	7F	127

Appendix B
Control Codes for Monitor and Operator's Display

ERASING B.1.

Cursor to end-of-line	ESC [K
	or ESC [0 K
Beginning-of-line to cursor	ESC [1 K
Line containing cursor	ESC [2 K
Cursor to end-of-screen	ESC [J
	or ESC [0 J
Beginning-of-screen to cursor	ESC [1 J
Entire screen	ESC [2 J

NOTE

Since the ESC key cannot be entered from the keyboard, substitute a backslash followed by the ASCII code for the ESC character. The example below erases the entire screen:

```
Print "\1B[2J"
```

CURSOR CONTROL SEQUENCES B.2.

Up	ESC [Pn A
Down	ESC [Pn B
Right	ESC [Pn C
Left	ESC [Pn D
Direct cursor addressing	ESC [Pl;Pc H
Where Pl = line number and Pc = column number	or ESC [Pl;Pc f
Index	ESC D
Next line	ESC E
Reverse index	ESC M
Save cursor and attributes	ESC 7
Restore cursor and attributes	ESC 8

DISPLAY ATTRIBUTES B.3.

Change display attributes ESC [Ps m

Where Ps =
- 0 (All attributes off)
- 1 (Bold)*
- 4 (Underscore)*
- 5 (Blink)
- 7 (Inverse/Reverse video)

* Not available for the operator's display.

DISPLAY MODE SEQUENCES B.4.

Insert mode enabled	ESC [4 h
Replacement mode enabled	ESC [4 l
Auto Line-feed mode enabled	ESC [20 h
Auto Line-feed mode disabled	ESC [20 l

Auto Wrap mode enabled	ESC [? 7 h
Auto Wrap mode disabled	ESC [? 7 l
Text cursor enabled	ESC [? 25 h
Text cursor disabled	ESC [? 25 l

TAB STOPS B.5.

Set at current column	ESC H
Clear at current column	ESC [g
	or ESC [0 g
Clear all tabs	ESC [3 g

EDITING CONTROL B.6.

Insert line	ESC [Pn L
Delete line	ESC [Pn M
Insert characters	ESC [Pn @
Delete characters	ESC [Pn P

ANNUNCIATOR CONTROL B.7.

Set Annunciators ESC [Ps q

Where Ps =

0 (All Annunciators Off)
1 (Busy)
2 (Stopped)
3 (Storing Seq)
4 (More Softkeys)
5 (More Information)
6 (Alpha)

When control returns from TL/1 to the operator interface, the operator interface sets the annunciators back to the correct state.

BEEPER CONTROL B.8.

Set Beeper ESC [Pl ; Pt y

Where Pl = length in milliseconds. The actual length is set in increments of 16 milliseconds with a maximum of 1.008 seconds.

and Pt =
0 Bell 1
1 Bell 2 (Default on power up)
2 Bell 1 and Bell 2

Setting the beep tone from the operator interface restores the beeper to Bell 2.

SPECIAL DISPLAY CHARACTERS FOR THE OPERATOR'S DISPLAY B.9.

Two-digit IC pin numbers (in GFI displays) are displayed in single-character cells. The special codes for these two-digit numbers are generated by this formula:

```
number + 95hex.
```

For example, the number 23, as displayed below, would use the hex code AC.

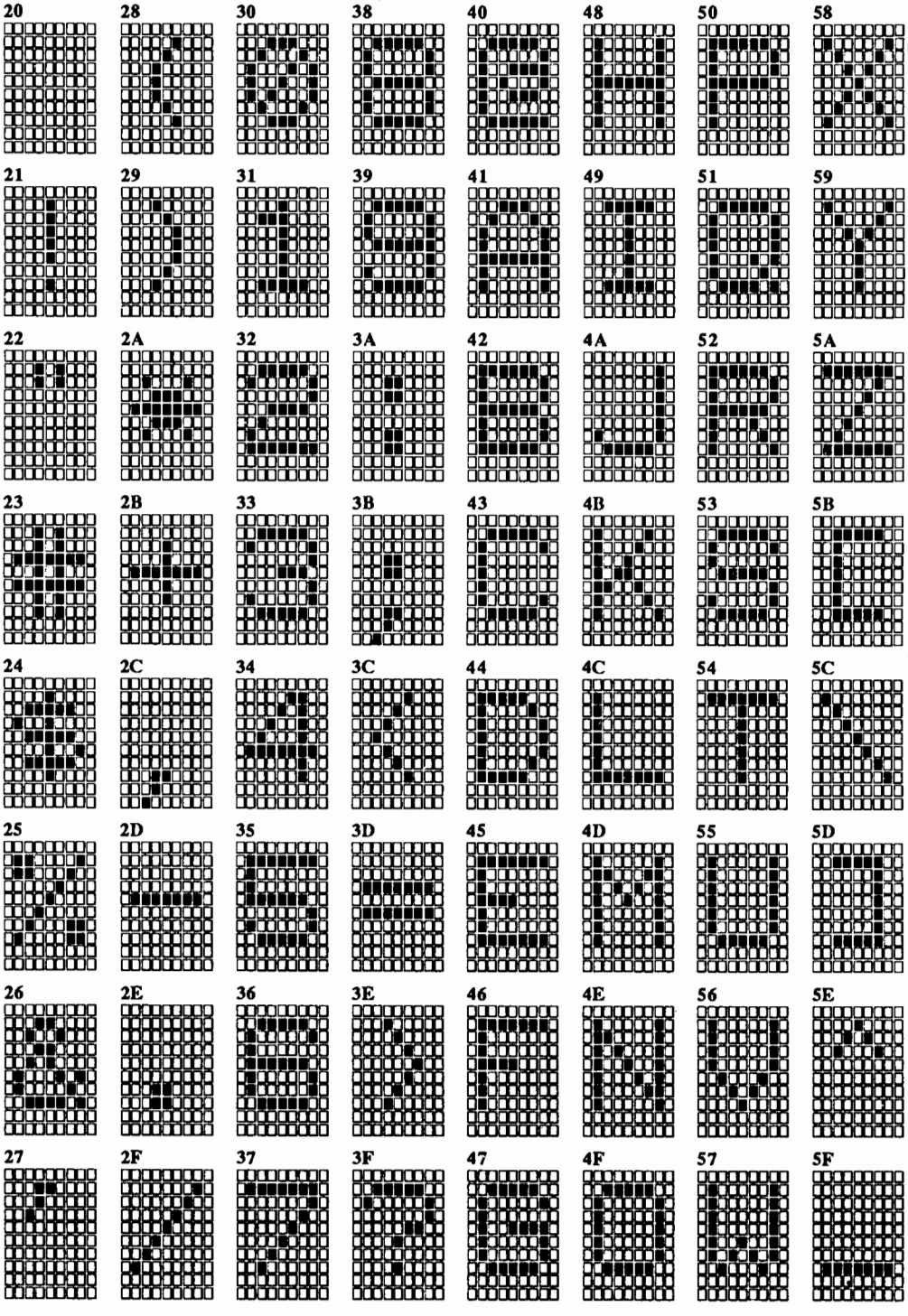
```
+--+
|2 |
| 3|
+--+
```

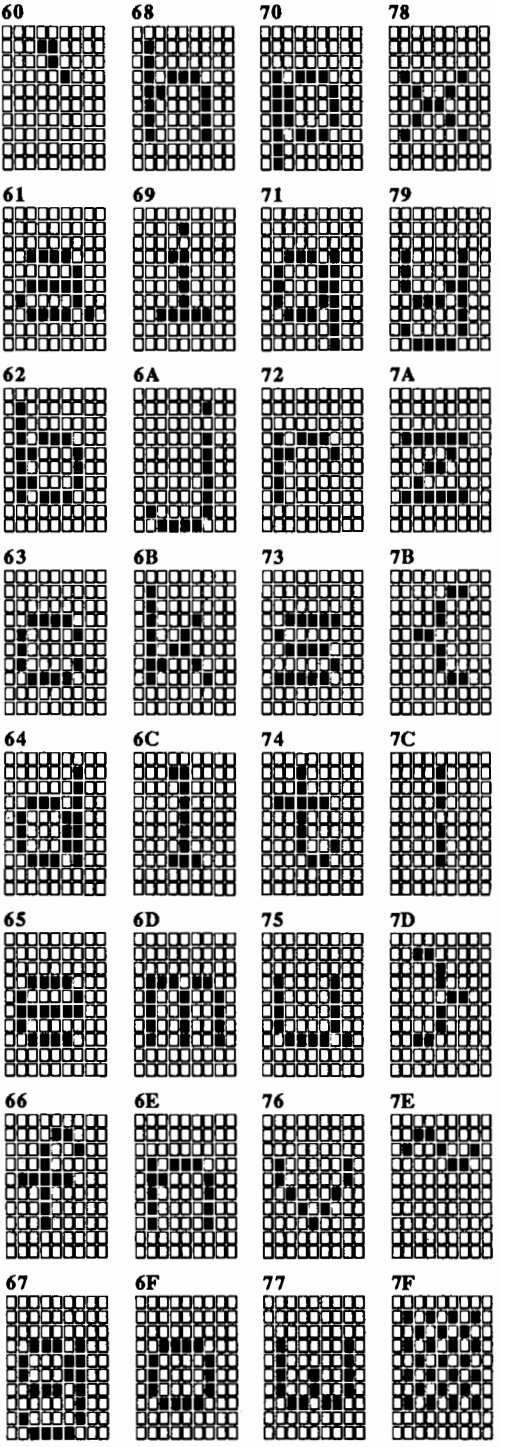
Other special symbols used in displaying ICs on the operator's display are shown below.

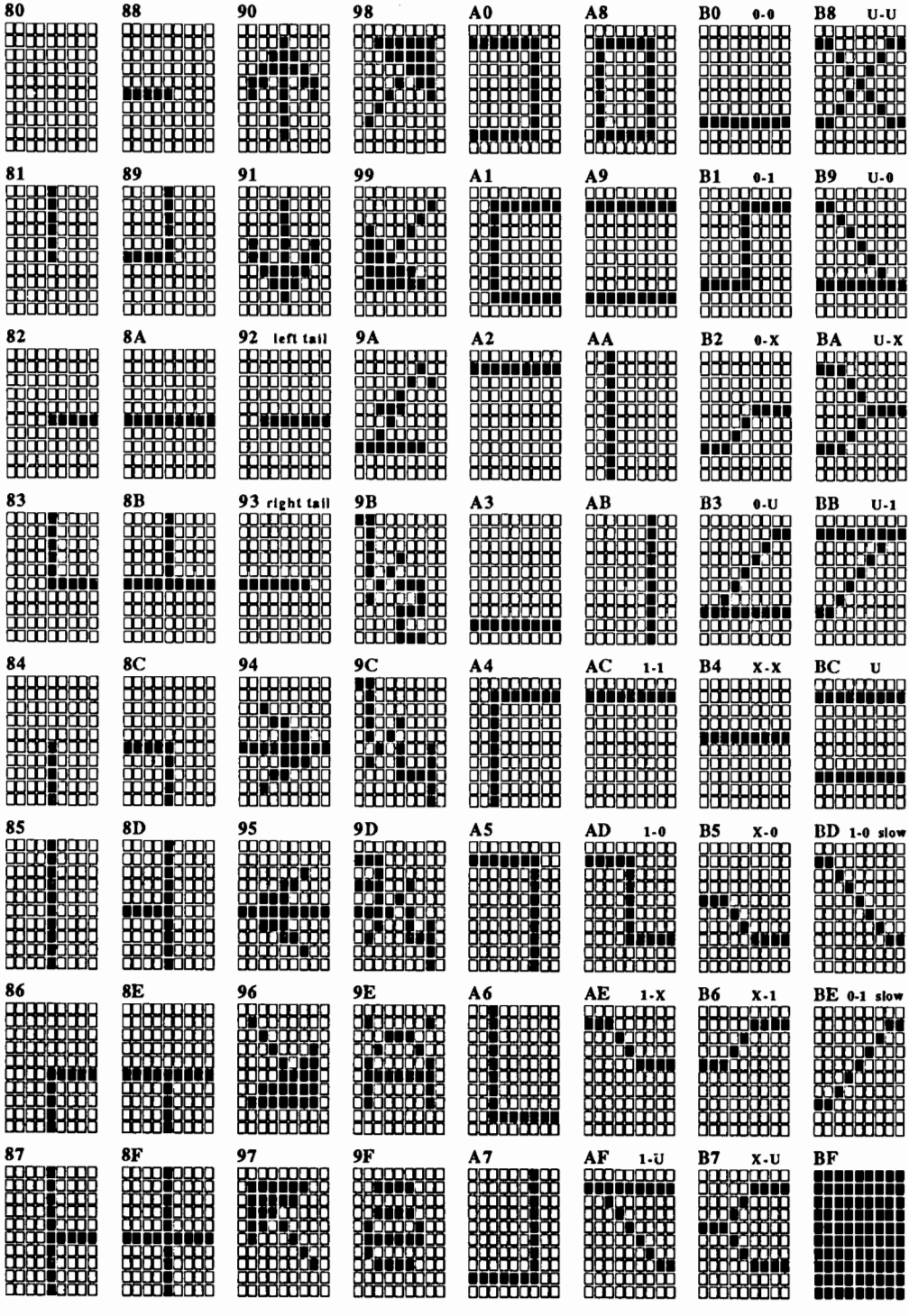
Symbol	*Hex*
IC Head	F9
IC Body	FA
IC Upper Leg	FB
IC Lower Leg	FC
IC Long Upper Leg	94
IC Long Lower Leg	93
Up Arrow	FD
Down Arrow	FE
Bidirectional Arrow	92
Left-adjusted Vertical Bar	FF

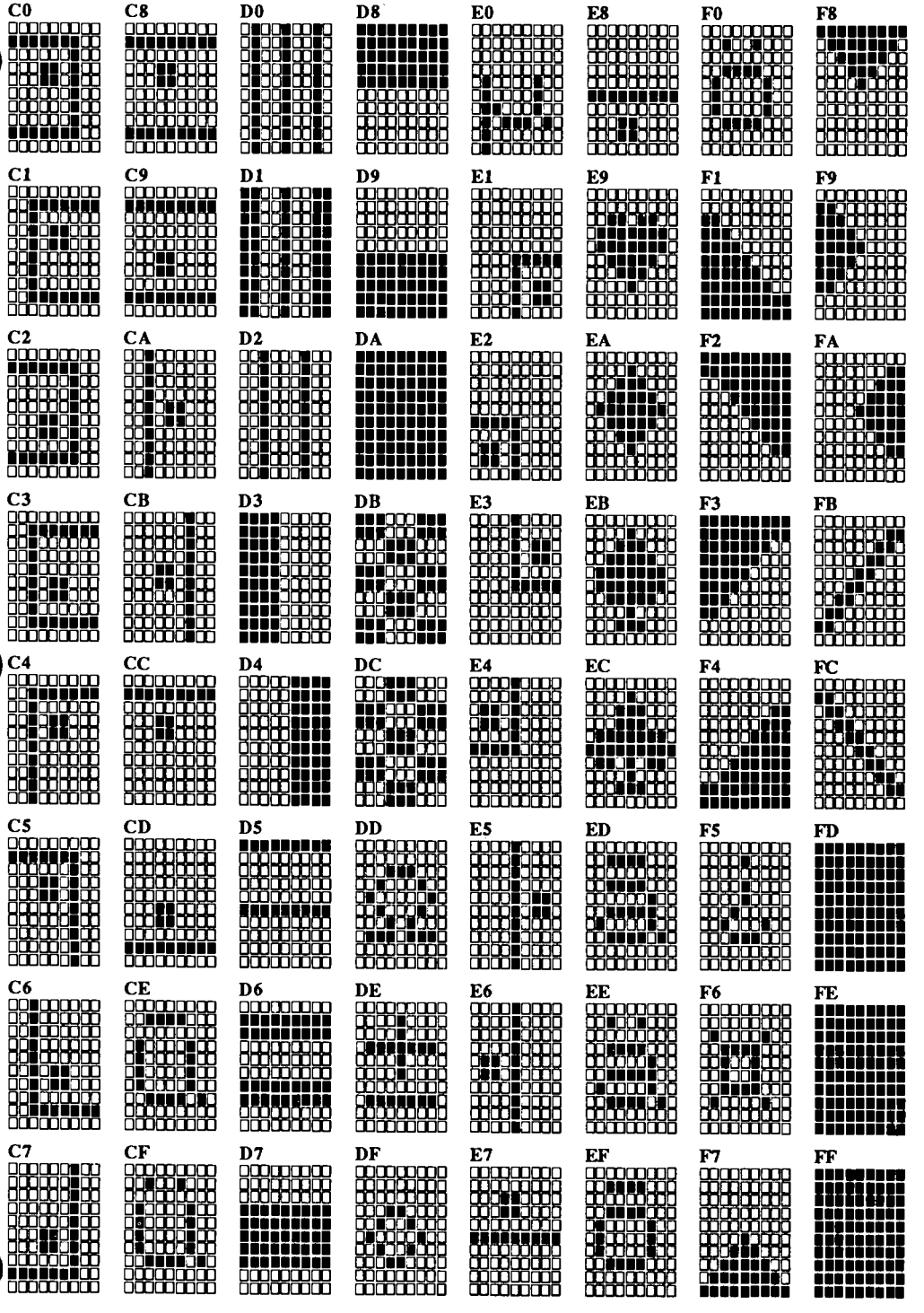
DISPLAY CHARACTERS FOR THE MONITOR B.10.

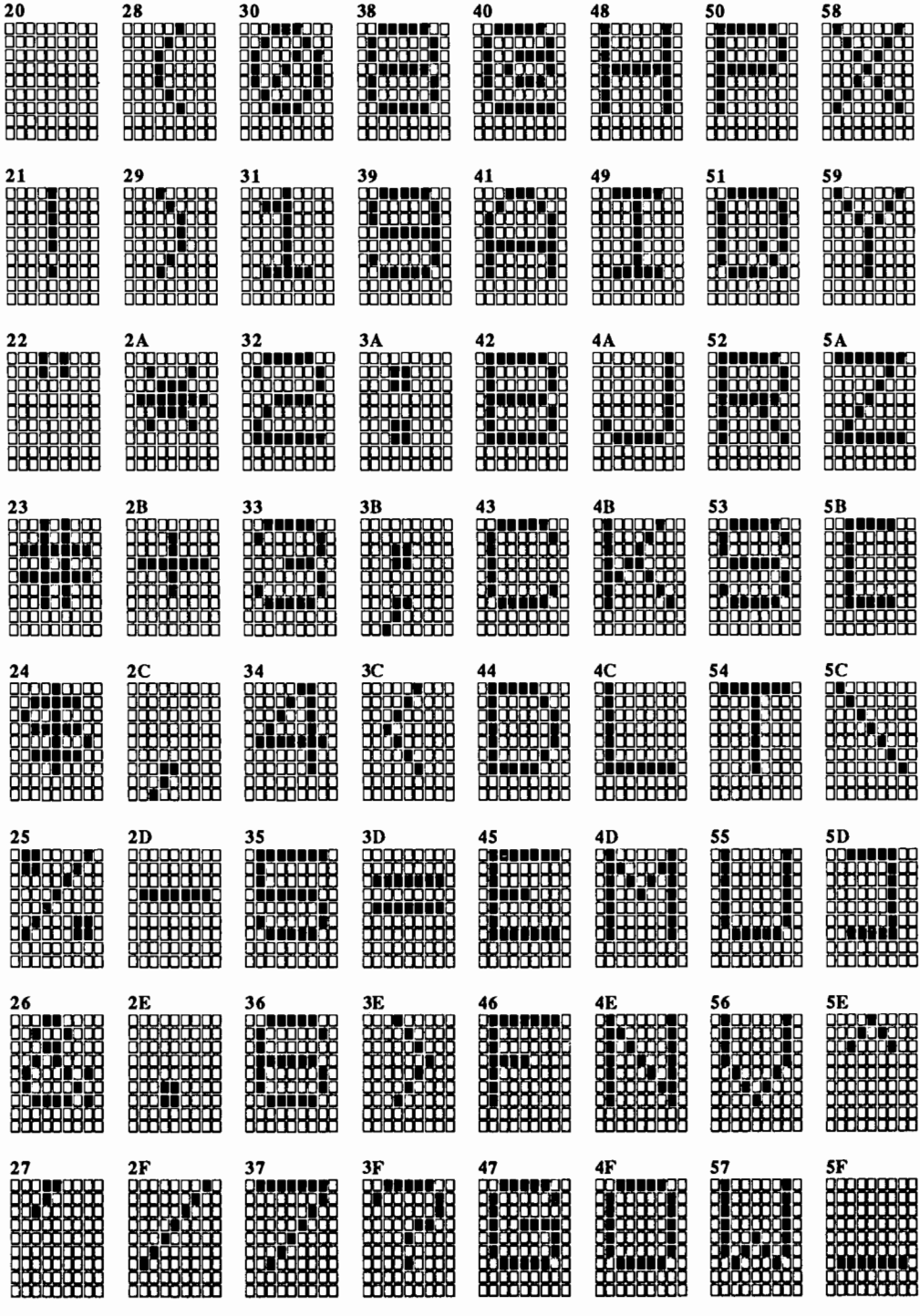
The 9100A/9105A may use either the Fluke monochrome monitor or a customer-supplied IBM-compatible color monitor. Both ASCII characters and special graphics characters may be displayed on either type of monitor. The following pages show the 9x10 pixel character cells used for monochrome display and the 9x9 pixel character cells used for color display. The number above each character represents the hexadecimal control code required to use the character.

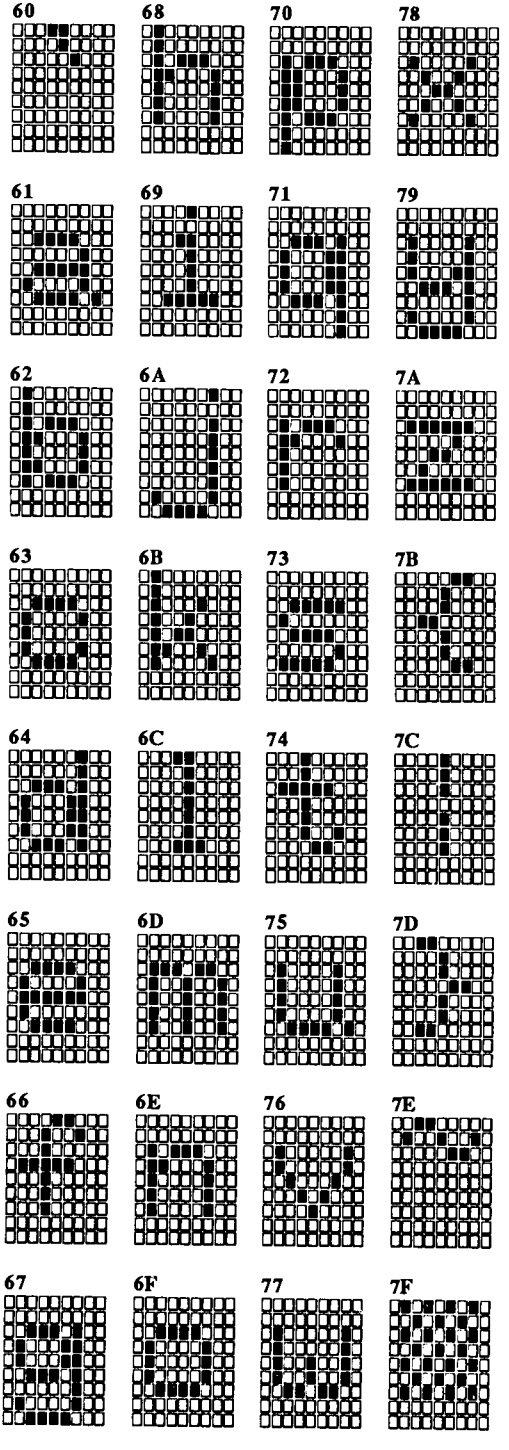

20 28 30 38 40 48 50 58
21 29 31 39 41 49 51 59
22 2A 32 3A 42 4A 52 5A
23 2B 33 3B 43 4B 53 5B
24 2C 34 3C 44 4C 54 5C
25 2D 35 3D 45 4D 55 5D
26 2E 36 3E 46 4E 56 5E
27 2F 37 3F 47 4F 57 5F

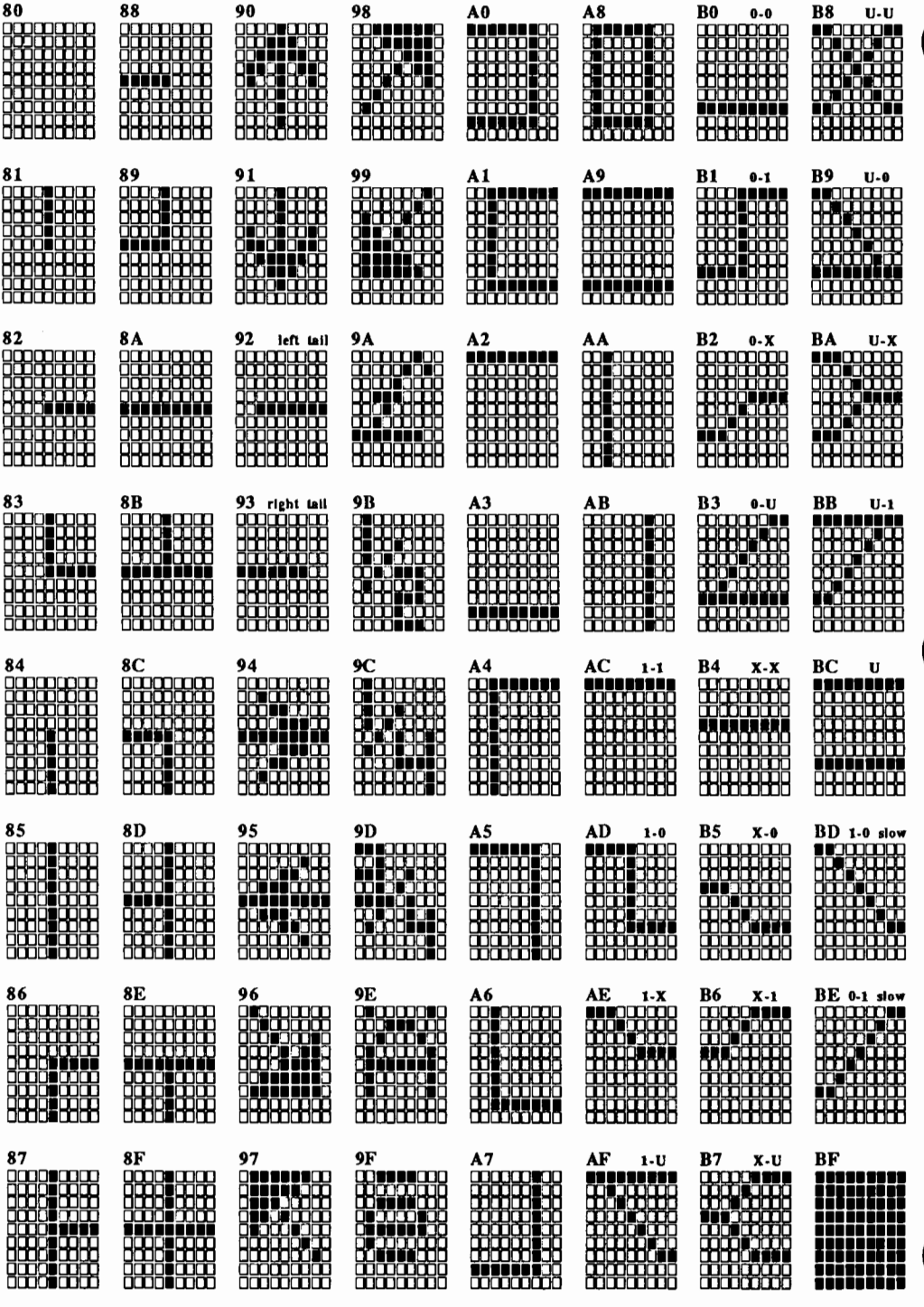

60
68
70
78
61
69
71
79
62
6A
72
7A
63
6B
73
7B
64
6C
74
7C
65
6D
75
7D
66
6E
76
7E
67
6F
77
7F

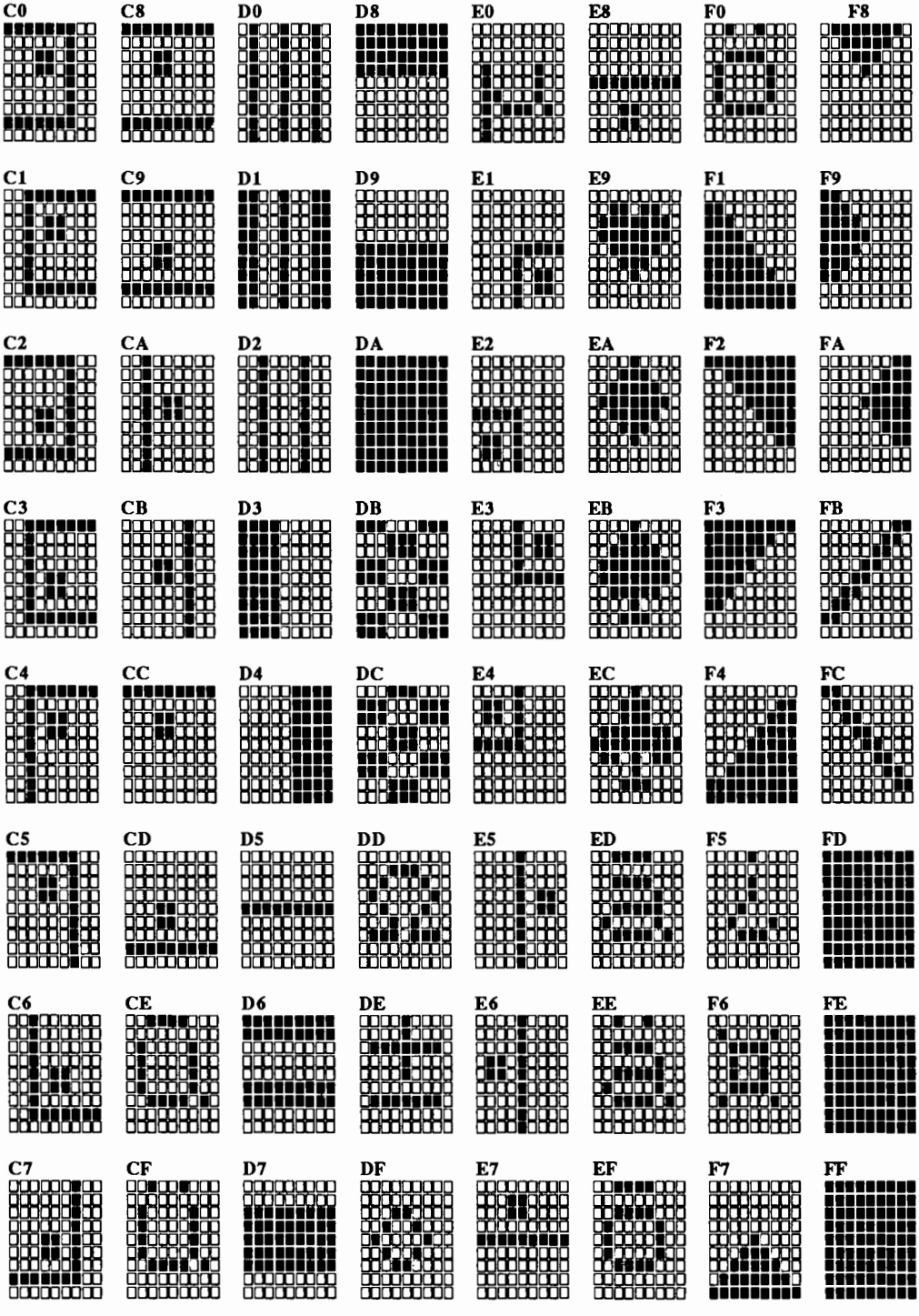

80
88
90
98
A0
A8
B0 0-0
B8 U-U
81
89
91
99
A1
A9
B1 0-1
B9 U-0
82
8A
92 left tail
9A
A2
AA
B2 0-X
BA U-X
83
8B
93 right tail
9B
A3
AB
B3 0-U
BB U-1
84
8C
94
9C
A4
AC 1-1
B4 X-X
BC U
85
8D
95
9D
A5
AD 1-0
B5 X-0
BD 1-0 slow
86
8E
96
9E
A6
AE 1-X
B6 X-1
BE 0-1 slow
87
8F
97
9F
A7
AF 1-U
B7 X-U
BF


C0
C8
D0
D8
E0
E8
F0
F8
C1
C9
D1
D9
E1
E9
F1
F9
C2
CA
D2
DA
E2
EA
F2
FA
C3
CB
D3
DB
E3
EB
F3
FB
C4
CC
D4
DC
E4
EC
F4
FC
C5
CD
D5
DD
E5
ED
F5
FD
C6
CE
D6
DE
E6
EE
F6
FE
C7
CF
D7
DF
E7
EF
F7
FF


20
28
30
38
40
48
50
58
21
29
31
39
41
49
51
59
22
2A
32
3A
42
4A
52
5A
23
2B
33
3B
43
4B
53
5B
24
2C
34
3C
44
4C
54
5C
25
2D
35
3D
45
4D
55
5D
26
2E
36
3E
46
4E
56
5E
27
2F
37
3F
47
4F
57
5F


60
68
70
78
61
69
71
79
62
6A
72
7A
63
6B
73
7B
64
6C
74
7C
65
6D
75
7D
66
6E
76
7E
67
6F
77
7F


80
88
90
98
A0
A8
B0 0-0
B8 U-U
81
89
91
99
A1
A9
B1 0-1
B9 U-0
82
8A
92 left tail
9A
A2
AA
B2 0-X
BA U-X
83
8B
93 right tail
9B
A3
AB
B3 0-U
BB U-1
84
8C
94
9C
A4
AC 1-1
B4 X-X
BC U
85
8D
95
9D
A5
AD 1-0
B5 X-0
BD 1-0 slow
86
8E
96
9E
A6
AE 1-X
B6 X-1
BE 0-1 slow
87
8F
97
9F
A7
AF 1-U
B7 X-U
BF


C0
C8
D0
D8
E0
E8
F0
F8
C1
C9
D1
D9
E1
E9
F1
F9
C2
CA
D2
DA
E2
EA
F2
FA
C3
CB
D3
DB
E3
EB
F3
FB
C4
CC
D4
DC
E4
EC
F4
FC
C5
CD
D5
DD
E5
ED
F5
FD
C6
CE
D6
DE
E6
EE
F6
FE
C7
CF
D7
DF
E7
EF
F7
FF

Appendix C
Operator's Keypad Mapping to TL/1 Input

Input from the following keys is available in either buffered or unbuffered mode.

	ASCII			ASCII	
Key	*CHR*	*HEX*	*Key*	*CHR*	*HEX*
EXEC (G)	G	47	ROM (S)	S	53
PROBE (H)	H	48	STIM (T)	T	54
BUS (I)	I	49	4	4	34
READ (J)	J	4A	5	5	35
C	C	43	6	6	36
D	D	44	7	7	37
E	E	45	OPTION (U)	U	55
F	F	46	(V)	V	56
MAIN MENU (K)	K	4B	SYNC (W)	W	57
GFI (L)	L	4C	(X)	X	58
I/O MOD (M)	M	4D	RUN UUT (Y)	Y	59
RAM (N)	N	4E	0	0	30
WRITE (O)	O	4F	1	1	31
8	8	38	2	2	32
9	9	39	3	3	33
A	A	41	EDIT (.)	.	2E
B	B	42	REPEAT (_)	_	5F
SETUP MENU (P)	P	50	LOOP (Z)	Z	5A
SEQ (Q)	Q	51	CONT (SPACE)	SP	20
POD (R)	R	52			

Input from the following keys is only available in unbuffered mode.

Key	ASCII CHR	ASCII HEX	Key	ASCII CHR	ASCII HEX
ENTER/YES	CR	0D	SOFT KEYS		94
CLEAR/NO	RUB	7F	F1		81
LEFT ARROW		A4	F2		82
UP ARROW		A1	F3		83
RIGHT ARROW		A3	F4		85
DOWN ARROW		A2	F5		86
HELP		92	EXT SW (input)	CR	0D
ALPHA	ESC	1B			

Appendix D
Programmer's Keyboard Mapping to TL/1 Input

The hexadecimal character codes less than or equal to 7F are the standard ASCII codes as defined in Appendix A, "ASCII Codes." The chart on the next page shows the mapping of the programmer's keyboard to non-standard character codes. Input from these keys is only available in unbuffered mode. "Shifted" indicates that the Shift key is pressed.

Key	*HEX*	*Shifted HEX*
F1	81	B1
F2	82	B2
F3	83	B3
F4	85	B5
F5	85	B6
F6	87	B7
F7	88	B8
F8	8A	BA
F9	8B	BB
F10	8C	BC
Edit	8D	BD
Quit	8F	BF
Msgs	91	C1
Help	92	C2
Info	94	C4
Begin File	95	C5
End File	97	C7
Scroll Forward	99	C9
Scroll Backward	9B	CB
Begin Line	9D	CD
End Line	9F	CF
Left Arrow	A4	D4
Down Arrow	A2	D2
Right Arrow	A3	D3
Up Arrow	A1	D1
Field Select	F0	F1
Break	F3	F3

Appendix E
I/O Module Clip/Pin Mapping

Clip size = 14, module installed on "A" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	14	=	20
2	=	2	13	=	19
3	=	3	12	=	18
4	=	4	11	=	17
5	=	5	10	=	16
6	=	6	9	=	15
7	=	7	8	=	14

Clip size = 14, module installed on "B" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	21	14	=	40
2	=	22	13	=	39
3	=	23	12	=	38
4	=	24	11	=	37
5	=	25	10	=	36
6	=	26	9	=	35
7	=	27	8	=	34

Clip size = 16, module installed on "A" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	16	=	20
2	=	2	15	=	19
3	=	3	14	=	18
4	=	4	13	=	17
5	=	5	12	=	16
6	=	6	11	=	15
7	=	7	10	=	14
8	=	8	9	=	13

Clip size = 16, module installed on "B" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	21	16	=	40
2	=	22	15	=	39
3	=	23	14	=	38
4	=	24	13	=	37
5	=	25	12	=	36
6	=	26	11	=	35
7	=	27	10	=	34
8	=	28	9	=	33

Clip size = 18, module installed on "A" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	18	=	20
2	=	2	17	=	19
3	=	3	16	=	18
4	=	4	15	=	17
5	=	5	14	=	16
6	=	6	13	=	15
7	=	7	12	=	14
8	=	8	11	=	13
9	=	9	10	=	12

Clip size = 18, module installed on "B" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	21	18	=	40
2	=	22	17	=	39
3	=	23	16	=	38
4	=	24	15	=	37
5	=	25	14	=	36
6	=	26	13	=	35
7	=	27	12	=	34
8	=	28	11	=	33
9	=	29	10	=	32

Clip size = 20, module installed on "A" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	20	=	20
2	=	2	19	=	19
3	=	3	18	=	18
4	=	4	17	=	17
5	=	5	16	=	16
6	=	6	15	=	15
7	=	7	14	=	14
8	=	8	13	=	13
9	=	9	12	=	12
10	=	10	11	=	11

Clip size = 20, module installed on "B side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	21	20	=	40
2	=	22	19	=	39
3	=	23	18	=	38
4	=	24	17	=	37
5	=	25	16	=	36
6	=	26	15	=	35
7	=	27	14	=	34
8	=	28	13	=	33
9	=	29	12	=	32
10	=	30	11	=	31

Clip size = 24, module installed on "A" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	24	=	20
2	=	2	23	=	19
3	=	3	22	=	18
4	=	4	21	=	17
5	=	5	20	=	16
6	=	6	19	=	15
7	=	7	18	=	14
8	=	8	17	=	13
9	=	9	16	=	12
10	=	10	15	=	11
11	=	29	14	=	32
12	=	30	13	=	31

Clip size = 24, module installed on "B" side

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	21	24	=	40
2	=	22	23	=	39
3	=	23	22	=	38
4	=	24	21	=	37
5	=	25	20	=	36
6	=	26	19	=	35
7	=	27	18	=	34
8	=	28	17	=	33
9	=	29	16	=	32
10	=	30	15	=	31
11	=	9	14	=	12
12	=	10	13	=	11

Clip size = 28

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	28	=	40
2	=	2	27	=	39
3	=	3	26	=	38
4	=	4	25	=	37
5	=	5	24	=	36
6	=	6	23	=	35
7	=	7	22	=	34
8	=	8	21	=	33
9	=	9	20	=	32
10	=	10	19	=	31
11	=	11	18	=	30
12	=	12	17	=	29
13	=	13	16	=	28
14	=	14	15	=	27

Clip size = 40

Clip		*I/O Mod Pin*	*Clip*		*I/O Mod Pin*
1	=	1	40	=	40
2	=	2	39	=	39
3	=	3	38	=	38
4	=	4	37	=	37
5	=	5	36	=	36
6	=	6	35	=	35
7	=	7	34	=	34
8	=	8	33	=	33
9	=	9	32	=	32
10	=	10	31	=	31
11	=	11	30	=	30
12	=	12	29	=	29
13	=	13	28	=	28
14	=	14	27	=	27
15	=	15	26	=	26
16	=	16	25	=	25
17	=	17	24	=	24
18	=	18	23	=	23
19	=	19	22	=	22
20	=	20	21	=	21

Appendix F
TL/1 Reserved Words

```
abort
and
array
bitmask
cpl
declare
else
elseif
end
endif
execute
exercise
exit
fails
fault
floating
for
function
global
goto
handle
if
input
len
local
loop
lsb
msb
next
not
numeric
on
or
passes
persistent
print
program
refault
return
setbit
shl
shr
step
string
test
then
times
to
until
using
while
xor
```

Appendix G
Handling Built-in Fault Conditions

OVERVIEW G.1.

This appendix lists the arguments that are provided with TL/1's built-in tests. These arguments can be used in your fault condition handlers (the *handle* command) and fault condition exercisers (the *exercise* command).

A TL/1 fault-condition handling procedure for the *ram_addr_fault* fault condition might look like this, for example:

```
handle ram_addr_fault (mask, access_attempted,
 addr, data_mask)
     declare
          string mask
          string access_attempted
          numeric addr
          numeric data_mask
     end declare
  !  <insert your code here>
end ram_addr_fault
```

The TL/1 applications interface always provides a fault condition handler for each fault test listed here. In the absence of a handler written by you, TL/1's default handler simply displays the name of the fault condition encountered on the operator's display.

When you provide a handler for a given test, TL/1 will use your handler and the fault message will not be displayed.

The process of calling a fault condition handler is exactly like calling a function. The arguments are essential even to handlers designed to ignore the first several occurrences of a fault condition; for example, if your handler is to ignore the first five occurrences but act on the sixth, all of the relevant arguments must be in place so that they can be used on the sixth occurrence.

ARGUMENT NAMES

`mask, mask_tied, mask_low, mask_high, mask_stat, mask_ctrl, mask_misc, mask_addr, mask_data (string)` - A 64-character string of faulted bits, where 0 = not faulted, and 1 = faulted. The rightmost bit is the least significant.

`access_attempted (string)` - UUT access when a fault condition occurred, e.g. "READ", "WRITE".

`addr (numeric)` - Address at which a fault condition occurred (as first detected by the test phase of the test/diagnostic built-in) or the low address of an address range.

`upto (numeric)` - High address in an address range.

`ctl (numeric)` - Value written to control lines of the UUT.

`verified (string)` - Parameter indicating that a fault condition has been verified (not intermittent), where verified = "confirmed" or "not confirmed."

`addr_expected (numeric)`	-	Address expected during a test.
`data_expected (numeric)`	-	Data expected during a test.
`code (numeric)`	-	Pod self-test error code.
`index (numeric)`	-	Index into a table of pod specific error messages.
`data (numeric)`	-	Data read from or written to the UUT.
`sig (numeric)`	-	Faulty signature read in a ROM test.
`sig_expected (numeric)`	-	Signature expected in a ROM test.
`iomod_nums (numeric)`	-	A bit mask for the faulty I/O module number (the least-significant bit is I/O module 1). Example: If iomod_nums is 3, it refers to both I/O module 1 and I/O module 2.
`data_mask (numeric)`	-	Mask of testable data bits, where 0 = not testable bit, 1 = testable bit.
`addrstep (numeric)`	-	Address increment.
`no_pins (numeric)`	-	If this argument is present, pin numbers are not to appear in message text on the display (only signal names are displayed). If the no-pins argument is absent, both pin numbers and signal names will appear in fault messages on the display.

`message, message1, message2 - (string)`	Additional information added to a fault. Message1 becomes the first line, and message2 becomes the second line of the display.
`err_num(numeric)` `err_msg(string)`	Arguments describing an error raised under special conitions as a fault (e.g. the *io_error* fault)

RAM TEST FAULT CONDITIONS G.3.

Fault Condition Name	*Arguments*

`ram_addr_addr_tied`	`mask`
	`(address bus tied)`
	`access_attempted`
	`addr`
	`data_mask`
	`no_pins`
	`verified`

`ram_addr_data_tied`	`mask`
	`(data bits tied to an address line)`
	`mask_tied`
	`(address bit tied to data bits)`
	`access_attempted`
	`addr`
	`data`
	`data_expected`
	`data_mask`
	`no_pins`

`ram_addr_data_tied_unconfirmed`	`mask`
	`(data bus tied to an address line)`
	`mask_tied`
	`(address bit tied to data bits)`
	`access_attempted`

Fault Condition Name	*Arguments*
	data data_expected data_mask no_pins
ram_addr_fault	mask (address bits faulted) access_attempted addr data_mask no_pins
ram_cell_cell_tied	mask (data bits tied) access_attempted addr data data_expected data_mask no_pins verified
ram_cell_high_tied	mask (data bits tied high) access_attempted addr data data_expected data_mask no_pins
ram_cell_low_tied	mask (data bits tied low) access_attempted addr data data_expected data_mask no_pins
ram_data_data_tied	mask (data bits tied together) access_attempted addr

Fault Condition Name	*Arguments*
	data_expected data_mask no_pins verified
ram_data_fault	access_attempted addr (address of cell unable to modify) data
ram_data_high_tied	mask (data bits tied high) access_attempted addr data data_expected data_mask no_pins
ram_data_incorrect	data_expected data access_attempted addr (address where read/write error occurred)
ram_data_low_tied	mask (data bits tied low) access_attempted addr data data_expected data_mask no_pins
ram_data_retention_fault	access attempted addr data_expected data

ROM TEST FAULT CONDITIONS G.4.

Fault Condition Name	*Arguments*
rom_addr_addr_tied	addr upto mask (addr bits tied together) addrstep data_mask no_pins
rom_addr_fault	addr upto mask (addr bits stuck) addrstep data_mask no_pins
rom_data_data_tied	addr upto mask (data bits tied together) addrstep data_mask no_pins
rom_data_fault	addr upto mask_low (data bits tied low) mask_high (data bits tied high) addrstep data_mask no_pins
rom_data_high_tied_all	addr upto addrstep data_mask no_pins

Fault Condition Name	*Arguments*
rom_data_low_tied_all	addr upto addrstep data_mask no_pins
rom_sig_incorrect	sig sig_expected addr upto

BUS TEST FAULT CONDITIONS G.5.

Fault Condition Name	*Arguments*
bus_addr_high_tied	mask (address bits tied high) addr
bus_addr_low_tied	mask (address bits tied low) addr
bus_addr_tied	mask (address bits tied together) addr
bus_data_high_tied	mask (data bits tied high) addr data
bus_data_low_tied	mask (data bits tied low) addr data

Fault Condition Name	*Arguments*
bus_data_tied	mask (data bits tied together) addr data

MEMORY INTERFACE POD FAULT CONDITIONS G.6.

Fault Condition Name	*Arguments*
m_bus_kernel	message1 message2
m_bus_addr_high	mask
m_bus_addr_low	mask
m_pod_buscycle_clock	mask_stat mask_ctrl mask_misc
m_pod_rom1_cs	message1 message2
m_pod_slow_clock	mask
m_pod_stopped	message1 message2
m_pod_no_reset	mask message1 message2
m_pod_reset_addr	addr_expected mask_high mask_low
m_pod_reset_data	data_expected mask_high mask_low

GENERIC FAULT CONDITIONS G.7.

Fault Condition Name	*Arguments*
generic_fault	message mask_stat mask_ctrl mask_addr mask_data mask_misc

PRIMITIVE FAULT CONDITIONS G.8.

Fault Condition Name	*Arguments*
clkmod_fuse_blown	*<none>*
iomod_dce	iomod_nums
iomod_fuse_blown	iomod_nums
iomod_current_fault	*<none>*
pod_addr_tied	mask (address bits tied together)
pod_breakpoint	*<none>*
pod_ctl_tied	mask (control lines tied)
pod_data_incorrect	data data_expected
pod_data_tied	mask (data lines tied)
pod_forcing_active	mask (forcing lines tied)
pod_interrupt_active	mask (interrupt line active)

Fault Condition Name	*Arguments*
pod_misc_fault	mask (miscellaneous signals faulted)
pod_special	index
pod_timeout_bad_pwr	*<none>*
pod_timeout_enabled_line	mask (enable line causing timeout)
pod_timeout_no_clk	*<none>*
pod_timeout_recovered	*<none>*.
pod_timeout_setup	*<none>*
pod_uut_power	*<none>*
podselftest_failed	code
probe_fuse_blown	*<none>*

I/O FAULT CONDITIONS G.9.

Fault Condition Name	*Arguments*
io_error	err_num err_msg

ARGUMENTS USED WITH BUILT-IN TESTS G.10.

The following built-in tests raise the primitive fault conditions (listed in the previous section). In addition to the arguments required as indicated by each primitive fault condition, each built-in test adds the arguments listed in the following table:

Built-in Test	*Arguments*
rampaddr	access_attempted addr
rampdata	access_attempted addr data
read	access_attempted addr
readblock	access_attempted addr
readstatus	*(Does not raise fault conditions)*
rotate	access_attempted addr data
toggleaddr	access_attempted addr
togglecontrol	access_attempted ctl
toggledata	access_attempted data addr
write	access_attempted data addr

Built-in Test	*Arguments*
writeblock	access_attempted data addr
writecontrol	access_attempted ctl
writefill	access_attempted data addr

Appendix H
Generating Built-in Fault Messages

OVERVIEW H.1.

This appendix relates the text of TL/1 fault messages to the arguments used in the TL/1 *fault* command. With this information, you can produce the same fault messages as those produced by the TL/1 default handler.

Fault messages are one or two lines long, and each line can have up to three variations, depending on the absence or presence of particular arguments in the handler. For example, the *pod_forcing_active* fault condition can produce these different messages:

```
1.    forcing signal <name> <pin> is active

2.    forcing signal <name> <pin> is active
      attempted to <action> at <address>

3.    forcing signal <name> <pin> is active
      attempted to <action> control <control>
```

Message 1 would be displayed if the following statement appeared in the program:

```
fault pod_forcing_active mask mask
```

Message 2 would be displayed if the following statement appeared in the program:

```
fault pod_forcing_active mask mask, access_attempted
 "read", addr $8000
```

Finally, message 3 would be displayed if the following statement appeared in the program:

```
fault pod_forcing_active mask mask, access_attempted
 "read", ctl $62
```

For the statements above, 62 is the argument value used for the argument named ctl, "read" is the argument value used for the argument named access_attempted, 8000 is the argument value used for the argument named addr, and mask is the variable name used to provide the argument value for the argument named mask.

Symbols H.1.1.

The following symbols and names are used in the tables that follow:

s	The argument is a string.
n	The argument is numeric.
x	The argument is included in the handler.
*	The message is always used for this fault condition.
1	The first alternative for the fault message.
2	The second alternative for the fault message.
3	The third alternative for the fault message.
4	The fourth alternative for the fault message.
5	The fifth alternative for the fault message.

Message Variables H.1.2.

`<action>`	One of the following actions:
	`write`
	`read`

`<name>`	A signal name.
`<pin>`	A pin number or alphanumeric designation.
`<data>`	A hexadecimal data value.
`<pod-special message>`	A pod-dependent fault message.
`<number>`	A decimal number.
`<address>`	A UUT address.
`<control>`	A control word.

Argument Names H.1.3.

`mask, mask_tied, mask_low, mask_high (string), mask-stat, mask-ctrl, mask-data, mask-addr, mask-misc`	- A 64-character string of faulted bits, where 0 = not faulted, and 1 = faulted.
`access_attempted (string)`	- UUT access when a fault condition occurred, (e.g. "read" or "write").
`addr (numeric)`	- Address at which a fault condition occurred or the low address of an address range.
`upto (numeric)`	- High address in an address range.
`ctl (numeric)`	- Value written to control lines of the UUT.

`message, message1, message2 (string)`	-	Message written to the display.
`verified (string)`	-	Parameter indicating that a fault condition has been verified, where verified = "confirmed" or "not confirmed."
`addr_expected (numeric)`	-	Address expected during a test.
`data_expected (numeric)`	-	Data expected during a test.
`code (numeric)`	-	Pod self-test error code.
`index (numeric)`	-	Index into a table of pod special error messages.
`data (numeric)`	-	Data read from or written to the UUT.
`sig (numeric)`	-	Faulty signature read in a ROM test.
`sig_expected (numeric)`	-	Signature expected in a ROM test.
`test_type (string)`	-	The type of test being executed.
`iomod_nums (numeric)`	-	A bit mask for the faulty I/O module number (the least-significant bit is I/O module 1). Example: If iomod_nums is 3, it refers to both I/O module 1 and I/O module 2.
`data_mask (numeric)`	-	Mask of valid data bits, where 0 = invalid bit, 1 = valid bit.
`addrstep (numeric)`	-	Address increment.
`no_pins (numeric)`	-	Pin numbers are not to appear in message text, where (0 = pins, 1 = no pins.)

`err_num(numeric)` `err_msg(string)`	-	Arguments describing an error raised under special conditions as a fault (e.g. the io_error fault).

HOW TO READ THE FAULT-MESSAGE TABLES H.2.

In the tables that follow, the fault conditions are organized alphabetically, the *Type* column shows whether the arguments for that fault condition are string or numeric variables. Columns *1, 2, 3, 4* or *5* contain an *x* if the related argument is needed to produce the fault message associated with that column.

For example, in the *bus_addr_tied* fault condition (next page), the arguments *mask, access_attempted,* and *addr* are needed in the fault condition handler if you want MESSAGE 2 to be displayed. If you prefer MESSAGE 3, use the *mask, access_attempted, addr,* and *data* arguments. To get MESSAGE 1, include the *mask* argument in your handler.

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
bus_addr_high_tied					
mask	s	x	x		
access_attempted	s		x		
addr	n		x		

MESSAGE 1: addr line <name> <pin> stuck high
MESSAGE 2: attempted to <action> at <address>

bus_addr_low_tied					
mask	s	x	x		
access_attempted	s		x		
addr	n		x		

MESSAGE 1: addr line <name> <pin> stuck low
MESSAGE 2: attempted to <action> at <address>

bus_addr_tied					
mask	s	x	x	x	
access_attempted	s		x	x	
addr	n		x	x	
data	n			x	

MESSAGE 1: addr line <name> <pin> tied
MESSAGE 2: attempted to <action> at <address>
MESSAGE 3: attempted to <action> data <data> at <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
bus_data_high_tied					
mask	s	x			
MESSAGE 1: data line <name> <pin> stuck high					
bus_data_low_tied					
mask	s	x	x		
MESSAGE 1: data line <name> <pin> stuck low					
bus_data_tied					
mask	s	x			
MESSAGE 1: data line <name> <pin> tied					
clkmod_fuse_blown		*			
MESSAGE 1: clock module fuse blown					

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
generic_fault					
message	s	x			
mask_stat	s	x			
mask_ctrl	s	x			
mask_addr	s	x			
mask_data	s	x			
mask_misc	s	x			

MESSAGE 1: <message>
status line <name> <pin>
control line <name> <pin>
address line <name> <pin>
data line <name> <pin>
line <name> <pin>

io_error					
err_num	n	x			
err_msg	s	x			

MESSAGE 1: I/O error <err_num>: <err_msg>

iomod_current_fault		*			

MESSAGE 1: I/O module overcurrent fault

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
iomod_dce					
iomod_nums	n	x			
access_attempted	s		x		
addr	n		x		

MESSAGE 1: compare condition reached in I/O module <number>
MESSAGE 2: attempted to <action> at <address>

iomod_fuse_blown					
iomod_nums	n	x			

MESSAGE 1: I/O module <number> fuse blown

m_bus_addr_high					
mask	s	x			

MESSAGE 1: addr line <name> <pin> was high - expected low

m_bus_addr_low					
mask	s	x			

MESSAGE 1: addr line <name> <pin> was low - expected high

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
m_bus_kernel					
message1	s	x			
message2	s	x			

MESSAGE 1: kernel fault
<message1>
<message2>

m_pod_buscycle_clock					
mask_stat	s	x			
mask_ctrl	s	x			
mask_misc	s	x			

MESSAGE 1: pod buscycle CLK BAD
check status line <name> <pin>
check control line <name> <pin>
check line <name> <pin>

m_pod_no_reset					
mask	s	x			
message1	s	x			
message2	s	x			

MESSAGE 1: no µP reset detected on <name> <pin>
<message1>
<message2>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
m_pod_reset_addr					
addr_expected	n	x			
mask_high	s	x			
mask_low	s	x			

MESSAGE 1: BAD reset address: <address> expected
<name> <pin> was high
<name> <pin> was low

m_pod_reset_data					
data_expected	n	x			
mask_high	s	x			
mask_low	s	x			

MESSAGE 1: BAD reset data: <data> expected
<name> <pin> was high
<name> <pin> was low

m_pod_rom1_cs					
message1	s	x			
message2	s	x			

MESSAGE 1: ROM1 CS or OE is stuck invalid
<message1>
<message2>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
m_pod_slow_clock					
mask	s	x			

MESSAGE 1: UTT clock <name> <pin> slow or stuck

m_pod_stopped					
message1	s	x			
message2	s	x			

MESSAGE 1: microprocessor stopped or bad
<message1>
<message2>

pod_addr_tied					
mask	s	x	x	x	
access_attempted	s		x	x	
addr	n		x	x	
data	n			x	

MESSAGE 1: addr line <name> <pin> not drivable
MESSAGE 2: attempted to <action> at <address>
MESSAGE 3: attempted to <action> data <data> at <address>

pod_breakpoint		*			
access_attempted	s		x		
addr	n		x		

MESSAGE 1: breakpoint reached
MESSAGE 2: attempted to <action> at <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
pod_ctl_tied					
mask	s	x	x	x	x
access_attempted	s		x	x	x
ctl	n				x
addr	n		x	x	x
data	n			x	

```
MESSAGE 1:      control line <name> <pin> not drivable
MESSAGE 2:      attempted to <action> at <address>
MESSAGE 3:      attempted to <action> data <data> at
                 <address>
MESSAGE 4:      attempted to <action> control <control>
```

pod_data_incorrect					
data	n	x			
data_expected	n	x			
access_attempted	s		x		
addr	n		x		

```
MESSAGE 1:      read incorrect data <data> expected <data>
MESSAGE 2:      attempted to <action> at <address>
```

pod_data_tied					
mask	s	x	x	x	
access_attempted	s		x	x	
addr	n		x	x	
data	n			x	

```
MESSAGE 1:      data line <name> <pin> not drivable
MESSAGE 2:      attempted to <action> at <address>
MESSAGE 3:      attempted to <action> data <data> at
                 <address>
```

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*				
		1	*2*	*3*	*4*	
pod_forcing_active						
mask	s	x	x	x		
access_attempted	s		x	x		
ctl	n			x		
addr	n		x			

MESSAGE 1: forcing signal <name> <pin> is active
MESSAGE 2: attempted to <action> at <address>
MESSAGE 3: attempted to <action> control <control>

pod_interrupt_active						
mask	s	x	x	x		
access_attempted	s		x	x		
ctl	n			x		
addr	n		x			

MESSAGE 1: interrupt <name> <pin> is active
MESSAGE 2: attempted to <action> at <address>
MESSAGE 3: attempted to <action> control <control>

pod_misc_fault							
mask	s	x	x	x	x	x	x
access_attempted	s		x	x	x	x	x
ctl	n					x	
data	n		x				
addr	n		x				x

MESSAGE 1: <name> fault <pin>
MESSAGE 2: attempted to <action> data <data> at <address>
MESSAGE 3: attempted to <action>
MESSAGE 4: attempted to <action> control <control>
MESSAGE 5: attempted to <action> at <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
pod_special					
index	n	x			
access_attempted	s		x		
addr	n		x		

MESSAGE 1: <pod-special message>
MESSAGE 2: attempted to <action> at <address>

pod_timeout_bad_pwr		*			

MESSAGE 1: pod timeout bad UUT power supply

pod_timeout_enabled_line					
mask	s	x			

MESSAGE 1: enabled line <name> <pin> causes timeout

pod_timeout_no_clk		*			

MESSAGE 1: pod timeout bad UUT clock

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
pod_timeout_recovered		*			
access_attempted	s		x	x	x
ctl	n				x
addr	n			x	

MESSAGE 1: pod timeout recovered
MESSAGE 2: attempted to <action>
MESSAGE 3: attempted to <action> at <address>
MESSAGE 4: attempted to <action> <control>

pod_timeout_setup		*				

MESSAGE 1: setup causes pod timeout

MESSAGE 1: bad UUT power supply

podselftest_failed						
code	n	x				

MESSAGE 1: pod selftest code = <number>

probe_fuse_blown		*				

MESSAGE 1: probe fuse blown

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
ram_addr_addr_tied					
verified	s	x	x		
mask	s	x	x	x	
no_pins	n		x		
access_attempted	s			x	
addr	n			x	

MESSAGE 1: addr line <name> <pin> tied to addr line <name> <pin>
MESSAGE 2: addr line <name> may be tied to addr line <name>
MESSAGE 3: attempted to <action> at <address>

ram_addr_data_tied					
mask	s	x	x		
mask_tied	s	x	x		
no_pins	n	x			
access_attempted	s		x		
addr	n		x		
data_expected	n		x		
data	n		x		

MESSAGE 1: addr line <name> tied to data line <name>
MESSAGE 2: attempted to <action> data <data> at <address> read <data>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
ram_addr_data_tied_unconfirmed					
mask	s	x	x		
mask_tied	s	x	x		
no_pins	n	x			
access_attempted	s		x		
addr	n		x		
data_expected	n		x		
data	n		x		

MESSAGE 1: addr line <name> may be tied to data line <name>
MESSAGE 2: attempted to <action> data <data> at <address> read <data>

ram_addr_fault					
mask	s	x	x		
no_pins	n	x			
access_attempted	s		x		
addr	n		x		

MESSAGE 1: addr line <name> stuck or open
MESSAGE 2: attempted to <action> at <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
ram_cell_cell_tied					
verified	s		x		
mask	s	x	x	x	
no_pins	n	x	x		
access_attempted	s			x	
addr	n			x	
data_expected	n			x	
data	n			x	

MESSAGE 1: memory cell for <name> coupled to memory cell for <name>
MESSAGE 2: memory cell for <name> may be coupled to memory cell for <name>
MESSAGE 3: attempted to <action> data <data> at <address> read <data>

ram_cell_high_tied					
mask	s	x	x		
no_pins	n	x			
access_attempted	s		x		
addr	n		x		
data_expected	n		x		
data	n		x		

MESSAGE 1: memory cell for <name> stuck high
MESSAGE 2: attempted to <action> data <data> at <address> read <data>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
ram_cell_low_tied					
mask	s	x	x		
no_pins	n	x			
access_attempted	s		x		
addr	n		x		
data_expected	n		x		
data	n		x		

MESSAGE 1: memory cell for <name> stuck low
MESSAGE 2: attempted to <action> data <data> at <address> read <data>

ram_data_data_tied					
verified	s			x	
mask	s	x	x	x	x
no_pins	n		x	x	
access_attempted	s				x
addr	n				x
data_expected	n				x
data	n				x

MESSAGE 1: data line <name> <pin> tied to data line <name> pod <pin>
MESSAGE 2: data line <name> tied to data line <name>
MESSAGE 3: data line <name> may be tied to data line <name>
MESSAGE 4: attempted to <action> data <data> at <address> read <data>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*				
		1	*2*	*3*	*4*	*5*
ram_data_fault		*				
access_attempted	s		x			
addr	n		x			
data	n		x			

MESSAGE 1: cannot modify RAM data
MESSAGE 2: attempted to <action> at <address> read <data>

ram_data_high_tied						
mask	s	x	x	x		
no_pins	n		x			
access_attempted	s			x		
addr	n			x		
data_expected	n			x		
data	n			x		

MESSAGE 1: data line <name> <pin> stuck high
MESSAGE 2: data line <name> stuck high
MESSAGE 3: attempted to <action> data <data> at <address> read <data>

ram_data_incorrect						
data	n	x				
data_expected	n	x				
access_attempted	s		x			
addr	n		x			

MESSAGE 1: read incorrect data <data> expected <data>
MESSAGE 2: attempted to <action> at <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
ram_data_low_tied					
mask	s	x	x	x	
no_pins	n		x		
access_attempted	s			x	
addr	n			x	
data_expected	n			x	
data	n			x	

MESSAGE 1: data line <name> <pin> stuck low
MESSAGE 2: data line <name> stuck low
MESSAGE 3: attempted to <action> data <data> at <address> read <data>

ram_data_retention_fault						
access_attempted	s			x		
addr	n			x		
data_expected	n			x		
data	n			x		

MESSAGE 1: RAM data retention fault (bad refresh?)
MESSAGE 2: attempted to <action> data <data> at <address> read <data>

rom_addr_addr_tied						
mask	s	x	x			
no_pins	n	x				
addr	n		x			
upto	n		x			

MESSAGE 1: addr line <name> tied to addr line <name>
MESSAGE 2: testing from addr <address> to <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
rom_addr_fault					
mask	s	x	x		
no_pins	n	x			
addr	n		x		
upto	n		x		

MESSAGE 1: addr line <name> stuck
MESSAGE 2: testing from addr <address> to <address>

rom_data_data_tied					
mask	s	x	x		
no_pins	n	x			
addr	n		x		
upto	n		x		

MESSAGE 1: data line <name> tied to data line <name>
MESSAGE 2: testing from addr <address> to <address>

rom_data_fault					
mask_low	s	x			
mask_high	s	x			
no_pins	n	x			
addr	n				
upto	n				

MESSAGE 1: data line <name> stuck low
or data line <name> stuck high
MESSAGE 2: testing from addr <address> to <address>

Fault Condition and Arguments	*Type*	*Fault Message Alternatives*			
		1	*2*	*3*	*4*
rom_data_high_tied_all		*			
addr	n		x		
upto	n		x		

MESSAGE 1: all ROM data bits stuck high
MESSAGE 2: testing from addr <address> to <address>

rom_data_low_tied_all		*			
addr	n		x		
upto	n		x		

MESSAGE 1: all ROM data bits stuck low
MESSAGE 2: testing from addr <address> to <address>

rom_sig_incorrect					
sig	n	x			
sig_expected	n	x			
addr	n		x		
upto	n		x		

MESSAGE 1: read incorrect sig <data> expected <data>
MESSAGE 2: testing from addr <address> to <address>

unknown_fault		*			
access_attempted	s		x		
addr	n		x		
data	n		x		

MESSAGE 1: unknown or intermittent fault occurred
MESSAGE 2: attempted to <action> data <data> at <address>

Appendix I
Pod-Related Information

POD CALIBRATION AND OFFSETS I.1.

Calibration is the process by which the internal delay lines in the I/O module and probe are adjusted to correctly align (in time) the clock and signals to be sampled. To calibrate an I/O module or probe to a pod for a particular pod sync mode, you are prompted to probe a signal on the UUT. The specified reference edge on that signal is found by adjusting the delay lines in the I/O module or probe relative to the internal PodSync signal. The appropriate delay, labeled "tcal" in Figure I-1, may vary from one pod to another and from one sync mode to another. If calibration is not performed, then a default setting is used for the tcal value. When calibration is performed, the measured value for tcal replaces the default value.

Once the reference edge is found, then an offset is applied to that edge to determine just where in time the I/O module or probe will latch data.

The following is an example for an imaginary "xyz" pod showing how the offset data is listed in this appendix and how this data would apply to real waveforms. Pod calibration and offset data for the "xyz" pod would appear in this appendix as follows:

Sync Mode	*UUT Signal*	*Edge of Signal*	*Offset from Edge*
ADDR	~ALE	rising	-24 ns

In the imaginary "xyz" pod, the reference edge for address sync is the rising edge of the ~ALE (address latch enable) signal. Furthermore, the offset data shows that a valid address is best captured when sampled 24 nanoseconds before the rising edge of ~ALE. (A positive offset would have indicated that the address should be latched after the reference edge.)

The waveforms corresponding to the above example are shown in Figure I-1. The ~ symbol indicates that both the ALE signal and the PodSync signal are active low.

As a result of the calibration process, the offset value is set to the default value for the sync mode in use. If other offsets are required, the TL/1 setoffset command can be used. See the setoffset command and the getoffset command in the "TL/1 Alphabetical Reference" section of this manual.

POD INFORMATION FOR 9100A/9105A USERS I.2.

In addition to the pod information in Fluke pod manuals, the *Supplemental Pod Information for 9100A/9105A Users Manual* provides the following additional information for each pod:

- **Address space options:** Shows the parameter names, parameter values, and all legal combinations of parameter values. Address space options are accessed through the OPTIONS key at the operator's keypad and display or through the TL/1 getspace command.
- **Pod-specific set-up information:** Shows the parameters that are available through the SETUP MENU key on the operator's keypad and display or through the TL/1 podsetup command. These parameters are used to set-up the pod for a specific UUT.

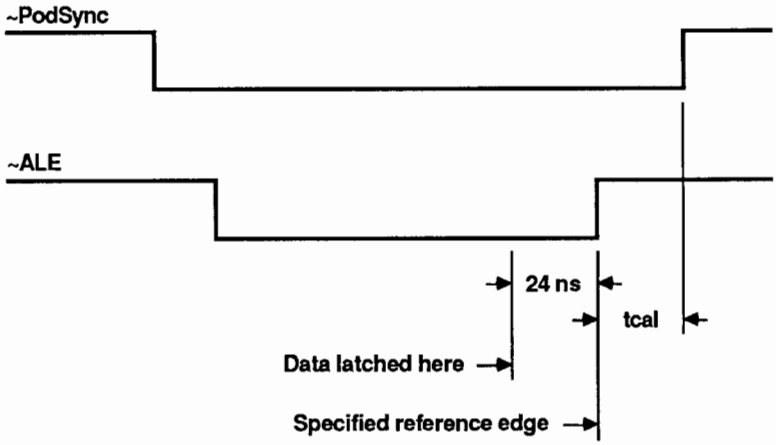


Figure I-1: Calibration and Offset Example Waveforms

- **TL/1 support programs:** Shows a list of the TL/1 programs that are available in the pod library. These programs provide convenient interfacing with any special functions built into the pod.
- **Pod sync calibration and offset data:** Shows the UUT signal, active (reference) edge of the signal, and the default offset from the specified reference edge for each sync mode.

SUMMARY OF 80186 POD SUPPLEMENTAL INFORMATION I.3.

The following information is included as one example of the information available in *The Supplemental Pod Information for 9100A/9105A Users Manual*. This information is particularly helpful for use with the *getspace* command, for pod-specific setup information when using pod-specific TL/1 support programs, and for adjusting pod sync calibration.

Address Space Options:

Parameter Names:	*mode*	*space*	*size*
Parameter Values:	NORMAL	MEMORY	WORD
	NORMAL	MEMORY	BYTE
	NORMAL	I/O	WORD
	NORMAL	I/O	BYTE
	DMA	MEMORY	WORD
	DMA	MEMORY	BYTE
	DMA	I/O	WORD
	DMA	I/O	BYTE

Pod-Specific Setup Information:

```
POD_CTL - Pod control addresses
        STDBY_AD - Standby read address
        RESET - Reset
```

```
SEG_REG - Segment registers
        EXTRA - Extra segment register
        STACK - Stack segment register
        CODE - Code segment register
        DATA - Data segment register

ERR_MASK - Error masks
        SUMMARY - Error summary mask
        CTRL_DR - Control drivability error mask
        ACT_FRC - Forcing signal error mask
        ACT_INT - Active interrupt signal mask
        SEG_DR - Segment drivability error mask
        ADDR_DR - Address drivability error mask
        DATA_DR - Data drivability error mask
        INTA_TO - Interrupt acknowledge and timer out
                  error mask
        CHIP_SEL - Chip select error mask

CS_REG - Chip select registers
        MPCS - MPCS register
        MMCS - MMCS register
        PACS - PACS register
        LMCS - LMCS register
        UMCS - UMCS register

DMA_CH0 - DMA channel 0
        CTRL_WD - Control word
        TRNS_CNT - Transfer count
        DP_UPPR - Destination pointer (upper four bits)
        DP_LWR - Destination pointer (lower sixteen bits)
        SP_UPPR - Source pointer (upper four bits)
        SP_LWR - Source pointer (lower sixteen bits)

DMA_CH1 - DMA channel 1
        CTRL_WD - Control word
        TRNS_CNT - Transfer count
        DP_UPPR - Destination pointer (upper four bits)
        DP_LWR - Destination pointer (lower sixteen bits)
        SP_UPPR - Source pointer (upper four bits)
        SP_LWR - Source pointer (lower sixteen bits)

TIMER0 - Timer 0
        M/C_WD - Mode/Control word register
        MAX_CNTA - Max count A register
        MAX_CNTB - Max count B register
        COUNT - Count register
```

```
TIMER1 - Timer 1
        M/C_WD - Mode/Control word register
        MAX_CNTA - Max count A register
        MAX_CNTB - Max count B register
        COUNT - Count register

TIMER2 - Timer 2
        M/C_WD - Mode/Control word register
        MAX_CNTA - Max count A register
        COUNT - Count register

INT_CNTR - Interrupt controller registers
        INT3 - INT3 control register
        INT2 - INT2 control register
        INT1 - INT1 control register
        INT0 - INT0 control register
        DMA1 - DMA1 control register
        DMA0 - DMA0 control register
        TMR_CTL - TIMER control register
        INT_STAT - Interrupt control status register
        INT_REQ - Interrupt request register
        IN_SERV - In-service register
        PRI_MSK - Priority mask register
        MASK - Mask register
        EOI - EOI register
        INT_VECT - Interrupt vector register
```

TL/1 Support Programs:

```
QWK_RD  Quick looping read
QWK_WR  Quick looping write
QWK_ROM Quick ROM test
QWK_RAM Quick RAM test
QWK_RAMP Quick ramp
QWK_FILL Quick fill
```

Pod Sync Calibration Data:

Sync Mode	*UUT Signal*	*Edge of Signal*	*Offset from Edge*
ADDR	ALE	falling	0 ns
DATA	~DEN	rising	-30 ns
INTA	~DEN	rising	-50 ns

Appendix J
9100A/9105A Error Numbers

INTRODUCTION J.1.

The 9100A/9105A associates an error number with each possible error that the instrument can encounter during operation. This association is system-wide. A given error has the same error number in all facets of system operation.

Also associated with each error is an error message, which is a string describing the error. For some errors, this error message string is modified to reflect certain error parameters (for example, if a file cannot be opened because it does not exist, the error message will include the name of the file).

Normally, error numbers are invisible to the operator or programmer, with only the error message displayed when an error is encountered. However, some errors can be dealt with in a TL/1 program, where it is more convenient to check error numbers than message strings when particular errors are handled differently.

ERROR NUMBERS J.2.

Figure J-1 contains only those errors that are relevant to TL/1 programming. The number of errors which can be generated by the 9100A/9105A is much larger than this list; however, errors not in this table cannot be intercepted by TL/1

programs. Also, error numbers from 900 through 999 are reserved for user-defined errors, which are sometimes needed in programs which deal with both system-specific and application-specific errors.

Error messages which are modified to reflect the actual error parameter(s) are shown with the error parameter(s) in italics.

The error numbers that appear in the following table are the 'err_num' argument to the io_error fault. This fault is raised by various I/O statements, including print and input.

Error Number	Error
200	Path Table full
201	Bad Path Number
203	Bad Mode
207	Out of Memory
211	End of File
214	File Not Accessible
215	Bad Path Name
216	Path Name Not Found
218	Creating Existing File
220	Phone hangup occurred (modem)
237	Out of Memory
238	Directory not empty
241	I/O error - bad disk sector number
242	Disk is Write Protected

Figure J-1: List of Error Numbers

Error Number	Error
243	I/O error -- bad CRC verify
244	Read Error
245	Write Error
246	No disk in drive
247	Unreadable or unformatted disk
248	Disk Full
249	Incompatible Disk Type
250	I/O Device Busy
251	Disk ID error
252	File record is busy (locked out)
253	Non-sharable file busy
255	Device is format protected; cannot format
300	IEEE-488 output buffer not empty
301	Operation requires an IEEE-488 address list
302	Operation requires system controller capability
303	Operation requires controller-in-charge capability

Figure J-1: List of Error Numbers (cont.)

Error Number	Error
304	IEEE-488 interface hardware is busy
306	Not configured for parallel poll
307	IEEE-488 transaction timed out
308	Operation requires controller capability
309	No listeners on the IEEE-488 bus
310	Cannot go to local in local lockout
4000	I/O attempted before any channels opened.
4001	Invalid BAUD Setting
4002	Invalid Parity Setting
4003	Invalid Number of Parity Bits
4004	Invalid Number of Stop Bits
4009	*file* is write protected
4010	*file* does not exist
4011	Text files cannot be opened in update mode
4012	There is no Video Interface
4013	There is no Operator Display
4014	Ports and files cannot have windows

Figure J-1: List of Error Numbers (cont.)

Appendix K
9100 Series
Software Error Report Form

9100 SERIES SOFTWARE ERROR REPORT FORM

We would like to thank you for taking the time to let us know about any bugs you encounter while using the 9100A or the 9105A. This information will help us in our goal of providing the best possible products for our customers.

We suggest that you retain this form as an original and use a photocopy for reporting a bug.

Name _____ Date _____

Company _____

Department _____ Mail Stop _____

Address _____

City/State/Zip _____

Country _____

Phone ( ) _____

Model: ☐ 9100A
☐ 9105A

Serial Number: _____

Software Version: _____

Pod In Use: _____

Description of Problem: _____

_____

_____

_____

_____

_____

_____

_____

(Continued on next page)

What sequence of steps led to the problem? _____

_____

_____

_____

_____

_____

_____

What was displayed on the operator's display when the problem was encountered?

Line 1: _____

Line 2: _____

Line 3: _____

Was the problem intermittent? ☐ Yes ☐ No

Were you able to work around the problem? If so, how?

_____

_____

_____

_____

_____

Please return the completed form to:

John Fluke Mfg. Co., Inc.
MR&D Sales Support
M/S 251E
Box C9090
Everett, WA 98206

Index

NOTE

TL/1 commands are not listed in this index. They are located alphabetically in Section 3.

Address Space Options, I-4
Annunciator control, B-3
Argument names, G-2, H-3
Arguments used with built-in tests, G-12
Arithmetic Operators, 2-12
Arrays, 2-11
ASCII Codes, A-1

Beeper control, B-4
Bit mask operators, 2-15
Bit shifting operators, 2-15
Block statements, 2-22
Bus test faults, G-8

Calibration, I-1
Case sensitive, 2-2, 2-3
Case, 2-2
Comment, 2-22
Conditional expressions, 2-17
Control codes, B-1
Cursor control sequences, B-2

Data types, 2-8
Device list, 2-6
Device names, 2-4
Display attributes, B-2
Display characters for the monitor, B-5
Display mode sequences, B-2
Double-quote character, 2-10

Editing control, B-3
Erasing, B-1
Error Numbers
    9100A/9105A, J-1
Error table, J-1
Esc key, B-1

Fault condition
    arguments, G-2, H-6
    handling, G-1
    messages, H-6
    raising, H-1
Fault message tables, H-6
File and directory names, 2-2, 2-3
Floating-point, 2-9
Functions, 2-20
    I/O module and probe, 2-20
    pod, 2-20
    special, 2-20

Generating built-in fault messages, H-1
Generic faults, G-10

Handling built-in fault conditions, G-1
How to read fault message tables, H-5

I/O module clip/pin mapping, E-1

Keypad mapping to TL/1 output, C-1

Logical operators, 2-13

Memory interface pod faults, G-9
Message variables, H-3
Monitor display, B-1

Name conventions, 2-2
Non-printing characters, 2-10
Numeric type, 2-8

Numeric type, 2-8
Numeric values, 2-8

Operator's keypad mapping, C-1
Operators, 2-11
Order of evaluation, 2-16
Organization of manual, 1-2
Other built-in test faults, G-12

Parentheses, 2-17
Pin names, 2-7
Pin numbers, 2-7
Pod calibration and offsets, I-1
Pod related information, I-2
Pod supplemental information, I-4
Pod sync calibration data, I-6
Pod-specific set-up information, I-2
Pod-specific setup information, I-4
Primitive fault conditions, G-10
Programmer's keyboard mapping to TL/1 input, D-1

Ram test faults, G-4
Ref pins, 2-7
Reference designator names, 2-7
Relational Operators, 2-12
ROM test faults, G-7

Simple statements, 2-22
Single quote character, 2-2
Software error report form, J-6
Special display characters, B-4
String
    operators, 2-14
Symbols, H-2
    used in displaying IC's, B-5

Tab stops, B-3
TL/1
    language conventions, 2-1
    reserved words, F-1
    statement conventions, 2-22
    support programs, I-6
Type conversion operators, 2-21