PROGRAMMER'S REFERENCE MANUAL

September 1990
Operating System 2.0

HOME PROCESSOR

Operating System 2.0

This release notice contains changes specific to the Home processor. Refer to the appropriate User/Progammer's Manual Update for a list of application specific changes.

1.1 General Enhancements

- **New Machine Configurations**
  Two new machine configurations: BRA and D-Channel are now supported on the Home Processor.

- **Automatic Printer and Remote Port Configuration**
  The HOME.D configuration file has been added which automatically executes default ITL commands to configure the printer and remote ports at bootup. The default configuration can be edited to match the users environment.

- **Application Load Menus**
  The loading menu for WAN applications has been split into a WAN Monitor and a WAN Emulation Applications Menu. The WAN Emulation Applications Menu now includes selections for conformance testing software.

  The BRA D-Channel Emulation Applications Menu has been added which includes selections for both emulation and conformance testing software.

  The APPL_CONF.F file has been added to dynamically create these application loading menus, and must be installed on WD0.

- **Fast Topic Select**
  Type the first letter of the desired topic to advance to that topic. Wraparound scan is implemented from left to right.

- **Copy and Verify Files**
  Filenames are now displayed as they are copied and verified.

- **Pause on Error**
  The *Pause on Error* function key has been added to both the copy and compare functions under the **Files** topic. When highlighted, the copy and compare functions pauses when an error occurs. The user can note the filename and choose whether or not to continue. If the error indicates a problem with remaining files, the copy operation will be aborted.

- **Backup Using Wildcards**
  Wildcards (*) can now be specified to back up a single file or group of files.

Printing Files
When printing multiple files specified by a wildcard (*), filenames are now displayed as they are printed.

Editor
It is no longer necessary to confirm an exit from the editor when no changes have been made to a file.

Test Ports Status Display
The filename is displayed on the Test Ports Status Display when a data recording is in progress, and displayed as 'Suspended' when a data recording is suspended.

Printer and Modem Port Flow Control
CTS/RTS flow control has been added. In addition, XON/XOFF, DTR, and CTS/RTS flow control can be combined to match the user's configuration.

1.2 PRA Enhancements

Channel Setup Menu
In emulation mode, Current Parameters for the selected test channel now displays an ACTIVE status indicating that the test channel is currently connected to the specified timeslot. A SUSPENDED status indicates the test channel is temporarily disconnected due to a change on the Ports Setup Menu.

NOTE
No status is displayed in monitor mode.

System Setup Menu
The *T1 D4 4F/M* framing format has been added which is the same as *T1 D4* framing format except the Fs (signalling channel framing bit) is set to 1 on the transmitter and is ignored by the receiver. This results in a 4 frame multiframe.

NOTE
This requires a special hardware modification and may not be supported on all units.

1.3 General Changes

Preference Menu
This menu has been removed.

Loading Applications
An application, reloaded from an application loading menu, now actually reloads from disk.

OTHERS Key
After exiting from a menu when the *OTHERS* function key was pressed, returning to the same menu results in a display of the original keys not the *OTHERS* set of keys.

✔ **Parallel Printer**
The *Parallel* function key replaces the *Centronics* function key on the Printer Port Setup Menu.

1.4 PRA Changes

✔ **Channel Setup Menu**
Channel Type has been removed from the menu. The channel type is now assigned automatically when an application loads.

The default for *Inverted HDLC* (IHDLC) has been changed to *YES* for T1 framing formats (D4 or ESF), and *NO* for CEPT framing formats.

✔ **Loopback**
The *Transmit Loopback* function has been removed.

✔ All references to BRI and PRI have been changed to BRA and PRA respectively.

✔ **System Setup Menu**
RJ45 Configuration has been changed to *RJ45(DB9) Config.* to include CEPT connectors.

Line Buildout on the System Setup Menu has been changed to *Transmit Equal* to correctly reflect its intended functionality of transmit pulse equalization.

✔ **Layer 1 Error Generation Menu**
Yellow Alarm and Blue Alarm have been changed to Yellow Alarm/RAI and Blue Alarm/AIS to reflect CEPT functionality.

✔ **Remote Alarm Indication**
The 'Remote Alarm Indication' status has been changed to 'RAI' to reflect CEPT functionality.

✔ **Configuration**
Changing PRA configuration (eg. PRA speed between 56 and 64 kbps) unloads the application to prevent unpredictable results when traffic is being received by the application.

1.5 General Problems Fixed

PR **Large Number of Files**
Directory listings, printing, copying, comparing, backup, and restoring backup can now be done on floppy disks and hard disk partitions that contain more than 283 files.

NOTE
A directory listing for a large number of files (more than 500) can take time. The disk LED will flicker indicating activity.

Backup
Pressing the RETURN key instead of the *Execute* function key, when prompted to insert a new blank diskette, no longer terminates the backup operation.

A file, open on another partition for data recording, printing, playback, or editing, no longer closes without warning. Only files opened during backup are closed.

Restore
During a single file restore, an existing file can now be restored on the selected partition.

Copies of backup diskettes created with this and future versions can now be restored.

Workaround:
To restore copies from previous versions:
Format a new floppy diskette.

Create a filesystem on this diskette using the following format for the filesystem name:

`WDx_Backup_#_n"`

Where x can be 0 to 7. Originally, it was the number of the partition from which the backup was created and is used in a full restore to create a new filesystem.

Where n is the number of the diskette in the backup set starting with 0 for the first diskette, 1 for the second diskette, and so on.

Copy the corresponding backup diskette onto this new diskette.

Directory Listing
Changing the filename, several pages into the directory listing, no longer results in a system crash.

Changing the filename, during a long directory listing, now displays file sizes correctly (in Kbytes).

Directory Printing
When printing a directory, a filename containing 14 characters, no longer repeats the 14th character as the 15th character.

Hard Disk Partitioning
WD0 on a 40 MByte drive can now be partitioned to use the entire 40 MBytes.

Comparing Files
Comparing identical small test manager binary files no longer results in a verification failure.

Editor
The 'last modified date' is updated only when the file is saved.

Files which are greater than 409 lines and have been transferred to the PT using RFILEX, cannot be edited.

- **Loading an Application**
  It is no longer necessary to reboot to reload an application after an application has failed to load.

- **Remote Port**
  The Remote (To Modem) port is now configured on machine bootup.

- **Setting Date and Time**
  An error message is now displayed if date and time entries are invalid.

1.6 PRA Problems Fixed

- The test channel 2 receiver is fully functional and is no longer affected by layer 1 transitions on Port B.

- PRA Drop & Insert is now fully operational and reliable.

- The PT is no longer flooded with layer 1 transitions.

- CEPT CRC4 framing format is now fully supported.

  NOTE
  The use of CEPT CRC4 may require a hardware upgrade. An error message will be displayed on the screen if CEPT CRC4 is selected and the hardware cannot support it.

- Timeslot allocation in the PRA Monitor is fixed.

- PRA Drop & Insert functions are fixed.

- PRA receive loopback with regeneration on is fixed.

- Selection of 75 ohm impedance in PRA CEPT modes is now fully operational.

1.7 General Known Problems

- **Configuration Diagram**
  The online connection for the D-Channel on the Configuration Diagram on the Home processor is always on after an application is loaded. Also, the online connection for the B-Channels not only has an effect on the application processors but on the voice and external paths as well.

- A crash will occur if the power source is changed repetitively on a BRA configured tester with D-Channel software and there is continuous traffic with another tester.

FILEX
A disk error will result when downloading more than one file, in one pass, with the receive name not specified. The disk error *'error during closing'* is displayed after the second file is received.

1.8 Known Hardware Problem

- Remote and printer speeds are not totally independent. Changing speed for one can affect the other. However, the following speeds do not cause this problem: 110, 135.5, 300, 600, 1200, 2400, 4800, 9600.

1.9 Known PRA Problems

- In Drop & Insert mode with T1 ESF framing, buffer synchronization might not occur immediately. This loss of buffer synchronization can be identified if the status quickly changes from 'Synchronized' to 'Buffer Overflow' and back to 'Synchronized'.

  Workaround:
  After synchronizing for about one minute, switch to monitor mode and then back to Drop & Insert mode.

- High levels of traffic can result in the inability to change PRA configurations.

  Workaround:
  For those applications with an *Online* function key, turn the application offline before changing the configuration on the Home Processor.

APPLICATION INDEPENDENT

The following enhancements and/or problems fixed are common to all of the following software applications.

SOFTWARE APPLICATIONS	VERSION	
Universal Monitor	2.0	Rev. 0
Universal Simulation	2.0	Rev. 0
X.25 Monitor	2.0	Rev. 0
X.25 Emulation	2.0	Rev. 0
SDLC/SNA Monitor	2.0	Rev. 0
SDLC Emulation	2.0	Rev. 0
BSC 3270 Monitor	2.0	Rev. 0
BSC 3270 Emulation	2.0	Rev. 0
ISDN D–Channel Monitor	2.0	Rev. 0
ISDN D–Channel Emulation	2.0	Rev. 0
SDLC/SNA Verification	2.0	Rev. 0
SNA Network Performance Analysis	2.0	Rev. 0
X.25 Network Performance Analysis	2.0	Rev. 0
X.25 Load Generator	2.0	Rev. 0
X.25/Q Monitor	2.0	Rev. 0
Teletex/Fax Gr. IV Monitor	2.0	Rev. 0
Remote Test Package	2.0	Rev. 0
X.75 Monitor	2.0	Rev. 0
X.75 Emulation	2.0	Rev. 0
ISDN Conformance Testing	2.0	Rev. 0
X.25 Conformance Testing	2.0	Rev. 0

2.1 Enhancements

Configuration File
When a monitor or emulation application is loaded from the Home processor, a corresponding default configuration file is executed which automatically configures the application. These files are uniquely named depending on the application and machine configuration. Refer to the Programmer's Reference Manual for valid filenames.

Example:
The following default configuration file can be edited on AP#1 on a BRA/WAN machine to customize the Bisync Emulation program.
`BSC_EMUL.D1`

Fast Topic Select
Type the first letter of the desired topic to advance to that topic. Wraparound scan is implemented from left to right.

Format Topic
The **Format** topic has been moved next to the **Display** topic.

Search for Timestamp
When the MM:SS:ssss timestamp format is selected, a prompt for MM:SS:ssss is displayed when searching for timestamps.

Clear Capture RAM
The *Clear* function key and the CLEAR_CAPT command have been added to clear the capture RAM buffer anytime.

Trace Display Format
Trace Display Format has been added to the Display Format Menu to control the display format for trace statements, independent of the data format.

WAN Interface Control Leads
Interface Control Leads has been added to the configuration menus to enable and disable the WAN interface control leads. Refer to the appropriate Programmer's Manual for corresponding commands.

Bit Rate
When clocking is provided by the interface, the bit rate can only be measured. When clocking is provided by the tester, the bit rate must be selected.

> **NOTE**
> *For accurate throughput measurement, the bit rate (line speed) must be measured or set to match the actual line speed.*

Display of Invalid Data
When hex and character display formats are selected, invalid data conditions are now reported.

Frame/Block Errors
The STATUS_ERR? command has been added to indicate a frame/block error has been detected in the currently processed frame.

RAM or Disk Playback with Filtered Display
When playing back data with a large sequential series of frames filtered out of the display, notices are displayed to indicate when searching/filtering is progressing and completed.

Test Manager Buffer Commands
The following commands have been added to allocate memory and manipulate text in buffers: ALLOT_BUFFER, FILL_BUFFER, APPEND_TO_BUFFER, and CLEAR_BUFFER.

2.2 Changes

Print On/Off
The *Print On/Off* function keys have been moved from the Display Format Menu to the **Print** topic.

Printing
Printing a data file, source file, or saving RAM to disk operations must now run to completion or be stopped via function key before accessing any other topics.

Playback Source
Using the control shift up and down arrows, now indicates whether the data source is capture RAM or disk.

2.3 Problems Fixed

Response Time Measurement
The start and end data blocks must now be specified to accurately calculate the response time between two frames.

Loading Test Script Error
Loading a test script no longer results in the display of a double error message when a disk error occurred.

Data Recording Suspend/Resume
The Connection Diagram now updates correctly when suspending or resuming a recording.

Error Handling for Buffer Usage
FILE->BUFFER no longer displays an incorrect message when unsuccessful.

An error message is displayed when attempting to send a buffer which has not been filled with the STRING->BUFFER or FILE->BUFFER command.

Save RAM to Disk
If Save RAM to Disk is selected during Playback Disk mode, the data display now changes from disk to RAM to select start and end transfer blocks.

Decode Variables
The REC-POINTER and REC-LENGTH variables now maintain the correct values for the current data event even if the test script or trigger had generated a trace statement.

Received timestamps are no longer altered when an emulation/simulation transmits a frame/block.

Throughput Graph
The throughput graph is now displayed accurately when long or short interval values are changed while the graph display is active.

Playback Disk Errors
The correct error message is now displayed when attempting to play back data from a nonexistent disk drive or partition.

Scrolling through playback RAM in continuous mode, is severely slowed down when running background tasks.

NOTE
This is more noticeable when the display filter is turned on.

PREFACE

Each application program can run a test script that consists of general and protocol specific commands. This manual is intended to provide a programming reference for these general commands and assumes some familiarity with basic programming concepts. Commands are organized according to the major components of the Interactive Test Language (ITL). Information contained in this manual is machine independent.

This manual is not intended to provide basic user instruction, but rather addresses the methods of writing test programs. Refer to the machine specific User Manual for a quick reference to the basic operation of the protocol tester.

IDACOM reserves the right to make any required changes in this manual without prior notice, and the user should contact IDACOM to determine if any changes have been made. No part of this manual may be photocopied, reproduced, or translated without the prior written consent of IDACOM.

IDACOM makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Copyright © **IDACOM** 1990

P/N 6000-1201

IDACOM Electronics Ltd.
A division of Hewlett-Packard

4211 - 95 Street
Edmonton, Alberta
Canada T6E 5R6
Phone: (403) 462-4545
Fax: (403) 462-4869

ITL and R-FILEX are trademarks of IDACOM Electronics Ltd.

TABLE OF CONTENTS

PREFACE

1 INTRODUCTION **1–1**

1.1 Stack Notation 1–2

1.2 Command Format 1–3

2 DATA TYPES **2–1**

2.1 Numeric Entry 2–1

2.2 String Entry 2–1

3 MEMORY ACCESS AND VARIABLES **3–1**

3.1 Access Commands 3–1
32 Bit Operations 3–1
16 Bit Operations 3–2
8 Bit Operations 3–3

3.2 Creating New Variables 3–3

3.3 Filling and Copying Memory 3–5

4 ARITHMETIC OPERATIONS **4–1**

5 BIT MANIPULATION **5–1**

6 PROGRAM CONTROL **6–1**

7 LOGICAL COMPARISONS **7–1**

7.1 Combining Expressions 7–3

7.2 Logical Negation 7–4

7.3 Miscellaneous 7–5

8 STRINGS **8–1**

8.1 String Manipulation 8–1

8.2 Converting Numbers to Strings 8–5

TABLE OF CONTENTS [continued]

8 STRINGS [continued]

8.3 Converting Strings to Numbers 8–6

9 SCREEN DISPLAY 9–1

9.1 Trace Reporting in the Data Window 9–1

9.2 Displaying in the User Window 9–5
Accessing the User Window 9–6
Creating Text 9–6
Cursor Control 9–7
Clearing Text 9–9
Color and Character Sets 9–9

9.3 Displaying in the Test Script Window 9–14

10 PRINTER PORT CONTROL 10–1

10.1 Configuring the Printer Port 10–1
Printer Output Commands 10–3

11 REMOTE PORT CONTROL 11–1

11.1 Configuring the Remote Port 11–1

11.2 ASCII Terminal 11–3
Sending Strings 11–3
IDACOM Logo 11–4
Remote Screen Display 11–4
Remote Directory Listing 11–5
Data Playback 11–5

11.3 File Transfer 11–7
Sending and Receiving Files 11–8

12 TIMESTAMPS 12–1

12.1 Timestamp Conversion 12–3

12.2 Timestamp Arithmetic 12–4

12.3 Copying Timestamps 12–5

12.4 Miscellaneous 12–5

TABLE OF CONTENTS [continued]

13 OPERATING SYSTEM 13–1

13.1 Port Identification 13–1

13.2 Audible Alarms 13–3

13.3 Machine Shutdown 13–4

13.4 Drive Selection 13–4

13.5 File Access 13–4

13.6 Switching Between Processors 13–8

14 COMPILER CONTROL 14–1

14.1 Conditional Compilation 14–1

14.2 Conditional Definition 14–2

15 STACK OPERATIONS 15–1

16 CREATING NEW COMMANDS 16–1

16.1 Pointers to Commands (Vectored Operation) 16–2

16.2 Remote Processor Execution 16–3

17 TEST MANAGER 17–1

17.1 Developing Source Code 17–1

17.2 Finite State Machine Concept 17–3

17.3 Symbol Definitions 17–3

17.4 Introduction to ITL Test Script Structure 17–5
Test Manager Theoretical Example 17–5
Test Script Structural Components 17–6

17.5 Event Recognition 17–7
Layer 1 Events 17–8
Received Frames 17–10
Timeout Detection 17–10
Function Key Detection 17–11
Interprocessor Mail Events 17–12
Wildcard Events 17–14

17.6 General Actions 17–15
Display, Capture, or Record 17–16

TABLE OF CONTENTS [continued]

17 TEST MANAGER [continued]
Audible Alarms 17–18
Input 17–18
Output – Notices and Errors 17–24
Starting or Stopping Timers 17–26
Manipulating Counters 17–28
Mailing to Another Processor 17–31
Protocol Specific Actions 17–33

17.7 Additional ITL Structures 17–33

17.8 Test Scripts 17–39
Loading a Test Script 17–39
Starting a Test Script 17–39
Stopping a Test Script 17–41
Saving a Test Script Binary 17–41

18 CONFIGURATION FILE 18–1

APPENDICES

A TEST SCRIPT COMPATIBILITY A–1

A.1 ?KEYBOARD A–1

A.2 Numerical Value Entry A–2

A.3 Output A–2

A.4 Detecting Cursor Keys A–4

B ERROR RECOVERY B–1

B.1 Description B–1

B.2 Recovery B–2

C CODING CONVENTIONS C–1

C.1 Stack Comments C–1

C.2 Stack Comment Abbreviations C–2

C.3 Program Comments C–2

C.4 Test Manager Constructs C–2

C.5 Spacing and Indentation Guidelines C–3

TABLE OF CONTENTS [continued]

C CODING CONVENTIONS [continued]

C.6 Colon Definitions C-4

D ASCII/EBCDIC/HEX CONVERSION TABLE **D-1**

E COMMAND CROSS REFERENCE LIST **E-1**

INDEX

LIST OF FIGURES

1-1 Sample Stack Notation 1-2
3-1 Copy Direction Using CMOVE 3-5
3-2 Copy Direction Using <CMOVE 3-6
9-1 Example of a Trace Report 9-1
9-2 Statistics Display 9-14
12-1 Data Queuing 12-2
13-1 Port Identifier Variable 13-1
17-1 Development System Environment 17-1
17-2 File Transfer Using R-FILEX 17-2
17-3 IDACOM's SDL Direction Conventions 17-4
17-4 SDL Representation for One State 17-5
B-1 Address Error Screen Display B-1

LIST OF TABLES

1-1 ITL Symbols 1-2
9-1 Color Attributes 9-3
9-2 Monochrome Attributes 9-3
9-3 Color to Monochrome Mapping 9-4
9-4 Monochrome to Color Mapping 9-4
9-5 Character Sets 9-9
13-1 Port Identifier Values 13-2
13-2 Processor Identifier Values 13-8
16-1 Processor Identifier Values 16-3
17-1 TO_DCE Control Lead Identifiers 17-9
17-2 TO_DTE Control Lead Identifiers 17-9
17-3 Mail Commands 17-32
17-4 MAIL_CMD Partners 17-32
17-5 Binary Filename Prefixes 17-42
17-6 Binary Filename Extensions 17-42
18-1 Configuration File Name Extensions 18-1
C-1 ITL Symbols C-2

1
INTRODUCTION

IDACOM has developed a comprehensive set of tools specifically for the development of test scripts. These test scripts control the operation of one of IDACOM's monitor or emulation programs. These tools include:

- a visual editor to prepare source code;
- an on–board compiler which compiles the test script while the monitor or emulation program is running; and
- IDACOM's Interactive Test Language (ITL) which consists of:
  - a set of high–level test manager constructs to provide the structure;
  - a set of events that the test script is programmed to recognize;
  - user–defined action sequences; and
  - primitive constructs that can be used either in test scripts or command mode.

ITL has rich complement of primitive commands. However, because ITL is an *extensible* language, new commands can be defined and used within ITL programs.

The Last In First Out (LIFO) stack as well as the ITL commands are modeled after the FORTH language. Users familiar with this language will quickly recognize many characteristics of FORTH within ITL.

All command definitions in this manual are described in terms of:

- input parameters (values removed from the stack);
- the operation performed on those values; and
- output parameters (results placed onto the stack).

Each command, or word in ITL, accepts zero or more parameters as input and in turn, outputs zero or more results. Both input parameters and output results are stored in a *stack.*

1.1 Stack Notation

The stack notation for a command which neither accepts nor produces parameters is written as (--). The following figure shows the notation for representing input and output parameters.

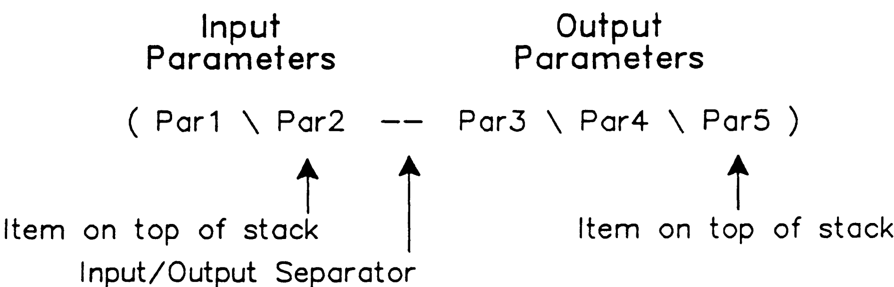


Figure 1–1 Sample Stack Notation

Stack notation can be described verbally by reading ' \ ' as *under*, and ' -- ' as *leaves*. The previous example would then be read as Par_1 under Par_2 leaves Par_3 under Par_4 under Par_5. The stack can be visualized as growing from bottom to top, with the last entry being on top.

Parameters within the stack comment often use a standard set of symbols. These symbols describe the *data type* of each parameter.

Symbol	Description
a	Memory address
b	8 bit byte
c	7 bit ASCII character
n	16 bit signed integer
d	32 bit signed integer
u	32 bit unsigned integer
f	Boolean flag (0=false, non-zero=true)
ff	Boolean false flag (zero)
tf	Boolean true flag (non-zero)
s	String (actual address of a character string which is stored in a count prefixed manner)

Table 1–1 ITL Symbols

1.2 Command Format

The standard format for most commands and variables in this manual is as follows:

COMMAND (in -- out)
(pronunciation)
Where: in = input parameters
out = output parameters

Description

Example: (where applicable)

For example, the *less than* command is described as follows:

< (d_1\\d_2 -- f)
(less than)
Where: d_1, d_2 = values to compare
f = result of comparison

Compares values 'd_1' and 'd_2' and returns a true (1) flag if 'd_1' is less than 'd_2'.

Example:
Check the length of a received frame to see if it is less than 4. If true, print a trace message.

```
L2-LENGTH  @  4  <
IF
      T." Frame is very short"  TCR
ENDIF
```

2
DATA TYPES

ITL uses a single storage type, 32 bit (4 byte) signed integer, to represent all objects that are used by the programmer. Thus, other data types used by the programmer are converted by ITL into a 32 bit signed integer representation for storage on the stack or in memory.

2.1 Numeric Entry

There are four possible bases for numeric entries. Each number base uses a special identifier before the number to specify decimal (base 10), hexadecimal (base 16), octal (base 8), or binary (base 2).

Base	Prefix	Permissible Range
Decimal	NONE	−2,147,483,648 to +2,147,483,647
Hexadecimal	0x 0X	0x00 to 0XFFFFFFFF
Octal	0c 0C	0c0 to 0x37777777
Binary	0c 0C	0b0 to 0b11111111111111111111111111111111

2.2 String Entry

Strings, of either ASCII or hexadecimal characters, can be entered to send data or to match incoming data.

Example:
ASCII strings are enclosed in quotes.

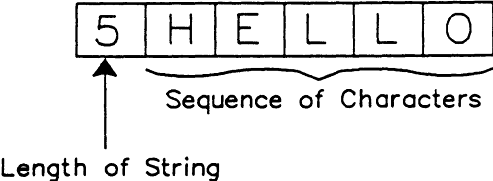
```
" The quick brown fox"
```

Hexadecimal strings are enclosed as follows:

```
X" 3031323A3B"
```

NOTE
There must be at least one space after the opening quotation mark.

Strings are stored in memory as a one byte length field (see below), followed by a sequence of bytes containing ASCII characters.



To reference a string using an entry on the 32 bit stack, ITL puts only the *address* of the string onto the stack. This address (or pointer) points to the first byte, or length field, of the string. For more information, see Section 8.

3
MEMORY ACCESS AND VARIABLES

ITL provides commands to store and recall values to predefined memory locations called variables. Variables are either supplied as part of a protocol application (i.e. COUNTER4, COUNTER10, L2-LENGTH, $MSG-CRVALUE, etc.) or defined by the user with the VARIABLE command.

By default, variables occupy four bytes of memory. However, the same commands which operate on variables can operate on a large memory area (or buffer) consisting of hundreds or thousands of bytes.

3.1 Access Commands

Memory access commands are divided into two classes of operation: read and store, and operate on 8, 16, and 32 bit data sizes.

32 Bit Operations

! (d\a --)
(store)
Stores a 32 bit value at the specified address.

Example:
Put the value '5' into the memory location as defined by COUNTER8.

```
5  COUNTER8  !
```

WARNING
Address 'a' must be an even value. An odd address causes an address violation. Refer to Appendix B for error recovery procedures.

@ (a -- d)
(fetch)
Fetches a 32 bit value 'd' (read) from address 'a'.

Example:
Read the value in the memory location specified by L2-LENGTH and place it on the top of the stack.

```
L2-LENGTH @
```

WARNING
Address 'a' must be an even value. An odd address causes an address violation. Refer to Appendix B for error recovery procedures.

+! (d\a --)
(plus-store)
Increments/decrements a 32 bit value 'd' to the contents of address 'a'.

Example:
Increment the contents of COUNTER4 by 10.
```
10  COUNTER4  +!
```

Decrement the contents of COUNTER4 by 5.
```
-5  COUNTER4  +!
```

16 Bit Operations

W! (n\a --)
('w' store)
Stores a 16 bit value 'n' in address 'a'.

Example:
Store a hex pattern 'AA55' into the first two bytes of 'data-buf'.
```
0XAA55   data-buf    W!
```

WARNING
Address 'a' must be an even value. An odd address causes an address violation. Refer to Appendix B for error recovery procedures.

W@ (a -- n)
('w' fetch)
Fetches a 16 bit value 'n' from address 'a'.

WARNING
Address 'a' must be an even value. An odd address causes an address violation. Refer to Appendix B for error recovery procedures.

NOTE
If the address 'a' is a variable, the most significant 16 bits are accessed.

8 Bit Operations

C! (b\a --)
('c' store)
Stores an 8 bit character value 'b' in address 'a'.

Example:
Store a 1 into the first byte of the data buffer.
```
0X01   data-buf   C!
```

C@ (a -- b)
('c' fetch)
Fetches an 8 bit character value 'b' from address 'a'.

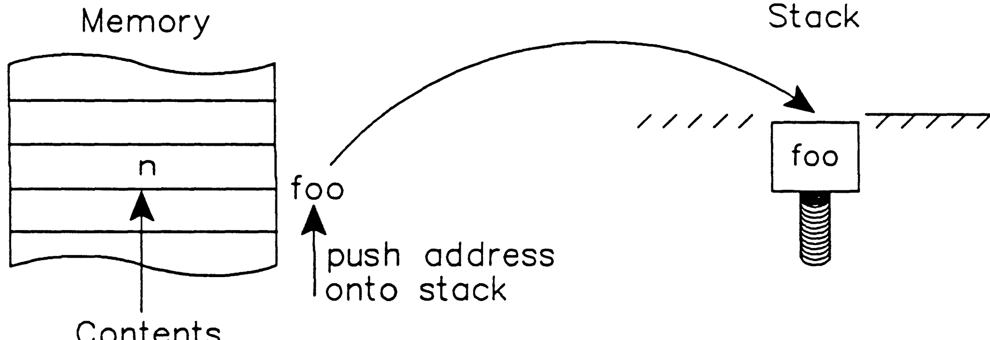
NOTE
If address 'a' is a variable, the most significant 8 bits are accessed.

3.2 Creating New Variables

In addition to the variables supplied by application software, other variables can be defined and named by the user. These variables occupy four bytes of memory (default), but can be expanded to any size.

When executed in a program, a variable places the address of the actual memory location on the stack.

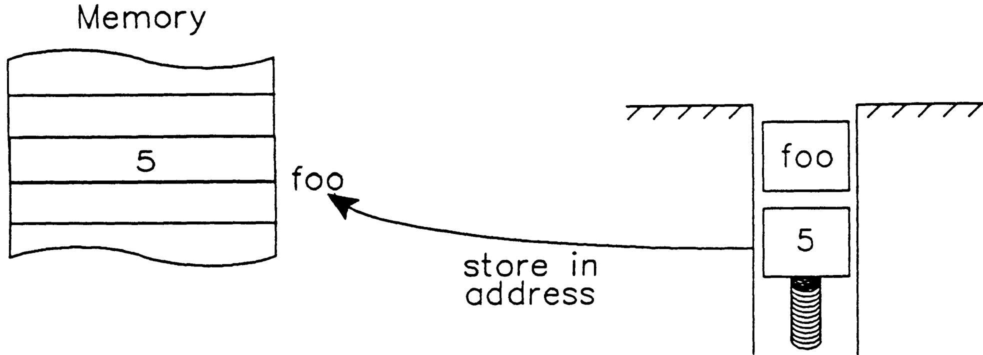
Example:
Assuming the variable 'foo' has been defined, place the address of 'foo' on the top of the stack as shown.
```
foo
```



Therefore, the action of the '!' (store) command takes the address and the value and performs a write operation.

Example:
Change the contents of 'foo' to 5.

```
5 foo !
```



VARIABLE (n --)

Where: n = the initial value

Defines the following word as a 32 bit variable.

Example:
Define a 4 byte variable called 'packets' with the initial value of 15.

```
15 VARIABLE packets
```

When the word 'packets' is encountered in a program, the address of 'packets' is placed on the stack.

NOTE
The initial value must always be specified, even if zero.

ALLOT (d --)

Allocates additional memory space for a variable; used in conjunction with the VARIABLE command. The parameter for ALLOT allocates in excess of the four bytes already reserved by the VARIABLE command.

Example:
Create a variable called 'data-buf' with a total length of 256 bytes (4+252).

```
0 VARIABLE    data-buf    252 ALLOT
```

NOTE
Only the first four bytes are initialized to zero. The contents of the rest of memory is unknown.

3.3 Filling and Copying Memory

FILL (a\d\b --)
Fills 'd' bytes, starting at address 'a', with the 8 bit value 'b'.

Example:
Fill the buffer (defined above) with zeros.

```
data-buf  256  0  FILL
```

FILLW (a\d\n --)
Fills 'd' words, starting at address 'a', with the 16 bit value 'n'.

Example:
Fill the memory area with an alternating 00FF pattern.

```
data-buf    128  0X00FF  FILLW
```

NOTE
Since FILLW uses the number of 16 bit words for the quantity, only one half of the buffer size is required (128 instead of 256).

WARNING
Address 'a' must be an even value. An odd address causes an address violation. Appendix B for error recovery procedures.

CMOVE (a_1\ a_2\n --)
(c move)
Copies the specified number of characters from one memory block starting at address 'a_1' to another memory block starting at address 'a_2'.

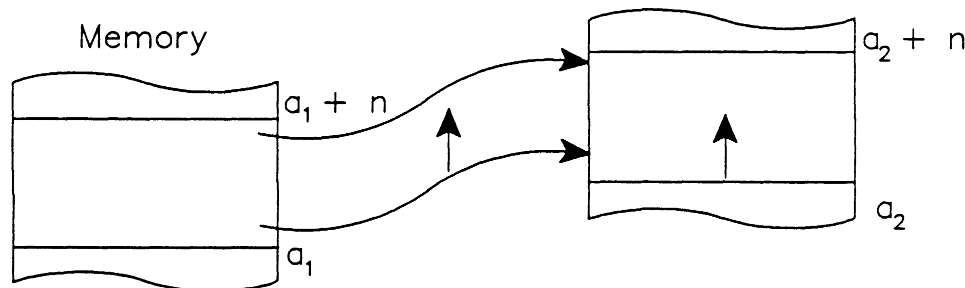


Figure 3–1 Copy Direction Using CMOVE

Example:
Copy the received data from the application software buffer 'L2–POINTER @' to the user's 'data–buf' variable.

```
L2-POINTER  @   data-buf   256   CMOVE
```

WARNING
Do not overwrite the buffer boundaries. In this example, the size of 'data–buf' must be at least 256 bytes.

NOTE
This command is also useful for copying the contents of a string into a buffer (see Section 8).

<CMOVE (a_1\a_2\n --)
Performs a block memory copy starting at the high end of the block defined by a_1 and proceeds downwards. This is used to shift the contents of a buffer upwards within the same buffer, thus preventing problems with memory overlap.

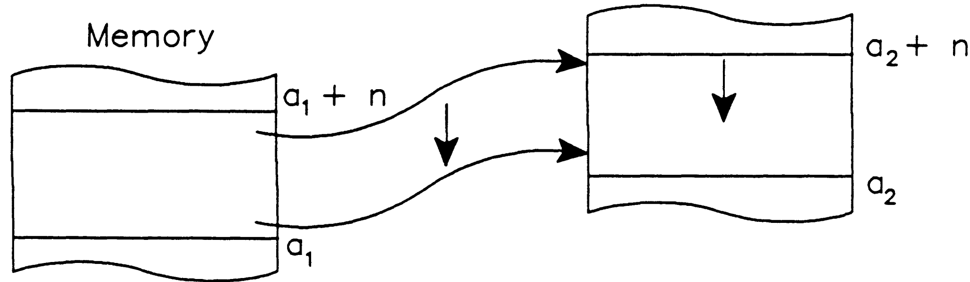


Figure 3–2 Copy Direction Using <CMOVE

Example:
Shift the contents of 'data–buf' 4 bytes upwards.

```
data-buf  data-buf  4+  252  <CMOVE
```

WARNING
Do not overwrite the buffer boundaries. In this example, data–buf must be at least 256 bytes.

4
ARITHMETIC OPERATIONS

ITL provides a standard set of operators to perform arithmetic functions. All of these operators modify values on the stack and in no way affect the contents of memory (i.e. all parameters are taken from the stack and the operator returns the results to the stack).

+ (d_1\\d_2 -- d_3)
(Add)
Where: d_1, d_2, d_3 = signed 32 bit integers

Adds 'd_1' and 'd_2' and leave the sum 'd_3' on the top of the stack.

+! (d\a --)
(plus-store)
Increments/decrements a 32 bit value 'd' to the contents of address 'a'.

Example:
Increment the contents of COUNTER4 by 10.
```
10  COUNTER4  +!
```

Decrement the contents of COUNTER4 by 5.
```
-5  COUNTER4  +!
```

- (d_1\\d_2 -- d_3)
(Subtract)
Where: d_1, d_2, d_3 = signed 32 bit integers

Subtracts 'd_2' from 'd_1' and leaves the difference 'd_3' on the top of the stack.

Example:
Leave the number '4' on the top of the stack.
```
7  3  -
```

***** (n_1\\n_2 -- d)
(Signed Multiply)
Where: n_1, n_2 = 16 bit signed integers
d = 32 bit signed product of n_1 and n_2

Multiplies 'n_1' by 'n_2' together and leaves the product 'd' on the top of the stack.

M* ($d_1\backslash d_2$ -- d_3)
(32 Bit Signed Multiply)
Where: d_1, d_2 = 32 bit signed integers
d_3 = 32 bit signed product of d_1 and d_2

Performs 32 bit x 32 bit multiplications, however, the result remains a 32 bit product.

U* ($u_1\backslash u_2$ -- u_3)
(Unsigned Multiply)
Where: u_1, u_2 = 16 bit unsigned integers
u_3 = 32 bit unsigned product of u_1 and u_2

Performs an unsigned 16 bit multiplication, resulting in a 32 bit product.

/ ($d_1\backslash n_2$ -- n_3)
(Signed Divide)
Where: d_1 = 32 bit signed dividend
n_2 = 16 bit signed divisor
n_3 = 16 bit signed quotient

Divides 'd_1' by 'n_2', producing quotient 'n_3'. This command should not be used for 32 bit division.

M/ ($d_1\backslash d_2$ -- $d_3\backslash d_4$)
(32 Bit Signed Divide)
Where: d_1 = 32 bit signed dividend
d_2 = 32 bit signed divisor
d_3 = 32 bit signed remainder
d_4 = 32 bit signed quotient

Divides 'd' by 'n_1' and produces two 32 bit values: the remainder and the quotient. The sign of the remainder is the same as that of 'd'.

U/ ($u_1\backslash u_2$ -- $u_3\backslash u_4$)
(Unsigned Divide)
Performs the equivalent unsigned operation of M/.

MOD ($n_1\backslash n_2$ -- d_3)
Where: n_1, n_2 = 16 bit signed integers
d_3 = 32 bit signed remainder of ($n_1 \div n_2$)

Leaves the remainder of 'n_1' divided by 'n_2'. This remainder has the same sign as 'n_1'.

ABS (d -- u)
Where: d = 32 bit signed integer
u = 32 bit unsigned integer

Returns the absolute value of 'd'.

To improve program performance, a number of common functions have been specially defined.

Addition	Multiplication/Division
1+	
2+	
3+	2*
4+	2/
6+	4*
8+	4/
10+	
12+	
14+	
16+	
18+	
20+	
22+	
24+	

5
BIT MANIPULATION

Bit operations, using 32 bit operands, are performed using standard logical operators.

These operations can be used to mask out bits in a variable to isolate specific fields. All of these operators modify values on the stack and in no way affect the contents of memory (i.e. all parameters are taken from the stack and the operator returns the result to the stack).

OR ($d_1 \backslash d_2$ -- d_3)
(logical OR)
Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise logical OR on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:
```
0xE45A 0x957C OR
```

1110	0100	0101	1010	(E45A)
1001	0101	0111	1100	(957C)

OR

1111	0101	0111	1110	(F57E)

AND ($d_1 \backslash d_2$ -- d_3)

(logical AND)

Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise logical AND on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:

```
0xE45A 0x957C AND
```

1110	0100	0101	1010	(E45A)
1001	0101	0111	1100	(957C)

AND

1000	0100	0101	1000	(8458)

XOR ($d_1 \backslash d_2$ -- d_3)

(exclusive OR)

Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise exclusive OR on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:

```
0xE45A 0x957C XOR
```

1110	0100	0101	1010	(E45A)
1001	0101	0111	1100	(957C)

XOR

0111	0001	0010	0110	(7126)

<< (d_1 -- d_2)

(logical shift left)

Shifts input parameter 'd_1' one bit to the left (towards the MSB – most significant bit) and leaves the result on the top of the stack. Overflow bits past the MSB are discarded. This instruction is equivalent to an unsigned multiply by 2.

Example:

Leave a value of 0b0010 on the stack.

```
0b0001 <<
```

>> (d_1 -- d_2)

(logical shift right)

Shifts input parameter 'n_1' one bit to the right (towards the LSB – least significant bit) and leaves the result on the top of the stack. Overflow bits past the LSB are discarded. This instruction is equivalent to an unsigned divide by 2.

Example:

Leave a value of 0b0010 on the stack.

```
0b0101 >>
```

<<# (d_1\b -- d_2)

(variable shift left)

Where: d_1= number to be shifted
b = number of positions to shift
d_2 = result

Shifts a 32 bit value 'd_1' to the left a specified number of bits. Overflow bits past the MSB are discarded.

Example:

Shift the first parameter to the left by 5 bits producing 0b01100100000.

```
0b00110010  5  <<#
```

>># (d_1\b -- d_2)

(variable shift right)

Where: d_1 = number to be shifted
b = number of bits to shift
d_2 = result

Shifts a 32 bit value 'd_1' to the right a specified number of bits. Overflow bits past the LSB are discarded.

Example:

Shift the first parameter to the right by 5 bits producing 0b00000001.

```
0b00110010  5  >>#
```

6
PROGRAM CONTROL

Several ITL primitives exist which dynamically modify the execution flow of a running program. They can be used to perform different command sequences based on the result of a current calculation or variable contents.

Generally, two categories of program control commands are available:
- selection-oriented (IF/ENDIF, DOCASE/CASE/ENDCASE) and,
- looping (DO/LOOP, REPEAT/UNTIL etc.).

IF ... ENDIF (f --)

Conditionally executes any commands between the IF and ENDIF depending on the value of the input flag 'f'.

The enclosed commands are executed for any value of 'f' not equal to zero. If 'f' is equal to zero (false), then program execution branches to the first statement after the ENDIF.

See Section 7 for valid expressions before an IF statement.

Example:
Check the value of COUNTER2, and if equal to eight, set COUNTER1 to equal 4 and produce an audible beep.

```
COUNTER2  @  8 =          ( Fetch counter 2 and compare to 8 )
IF
      4 COUNTER1  !         ( If true, set counter 1 and beep )
      BEEP
ENDIF
```

IF ... ELSE ... ENDIF (f --)

Allows two action paths to be followed: one if the comparison is true, another if the comparison is false.

Example:
Show the logic for implementing a modulo 8 sequence number scheme. If the current value of NR (the received sequence number) is equal to seven, then set it to zero; if not, then increment its value by one.

```
NR  @  7 =
IF
    0  NR  !                  ( Set sequence no. to 0 )
ELSE
    1  NR  +!                 ( Increment sequence no. )
ENDIF
```

Both IF/ENDIF and IF/ELSE/ENDIF statements can be intermixed and nested. Nesting allows decisions within decisions so that more powerful control structures can be built up.

Example:
Check the contents of the FRAME-TYPE variable, and if equal to an RR frame, increment the sequence number according to the modulo 8 procedure. If the frame is not an RR, the ELSE portion of the outer IF/ELSE/ENDIF structure is executed and a warning message plus a beep is produced.

```
FRAME-TYPE  @  R*RR =
IF
    NR  @  7 =
    IF
        0  NR  !              ( Set sequence no. to 0 )
    ELSE
        1  NR  !              ( Increment sequence no. )
    ENDIF
ELSE
    T." RR not received"  TCR ( Output a trace )
    BEEP
ENDIF
```

DOCASE ... CASE/ORCASE ... ENDCASE (d --)
Where: d = the value to use for selection criteria.

Selects one of many possible code segments to execute based on an input value.

Compares the input parameter 'd' with each item listed until a match is found. Once a match is found, the commands enclosed in the brackets are executed. Once the action sequence has completed, execution branches to the first statement following 'ENDCASE'.

Example:
ORCASE can be used when two values execute the same actions. Action 3 is executed if the input value, d, matches either d_4 or d_5. In the event that none of the items match, the DOCASE construct normally exits without taking any action. 'CASE DUP' causes the execution of the default action if none of the previous conditions are met.

```
DOCASE
      CASE d2     { ...action 1... }
      CASE d3     { ...action 2... }

      CASE d4
      ORCASE d5   { ...action 3... }

      CASE DUP    { ...default action... }
ENDCASE
```

Example:
This example is taken from a large test program for BSC 3270 testing and collects statistics on polling activity. Upon reception of a general poll, a different counter (1 to 6) is incremented depending on the contents of the 'LogicalUnit' variable. If the poll is to a station other that 0 through 5, then counter seven is incremented.

```
LogicalUnit  @
DOCASE
    CASE 0    { 1 COUNTER1  +! }
    CASE 1    { 1 COUNTER2  +! }
    CASE 2    { 1 COUNTER3  +! }
    CASE 3    { 1 COUNTER4  +! }
    CASE 4    { 1 COUNTER5  +! }
    CASE 5    { 1 COUNTER6  +! }
    CASE DUP  { 1 COUNTER7  +! }   ( default case )
ENDCASE
```

DO ... LOOP (d_1\d_2 --)

Where: d_1 = the ending value of the index
d_2 = the starting value of the index

Repeats a given section of ITL commands a specified number of times. The body of the DO/LOOP executes once if 'd_2' is greater than 'd_1'.

Example:
Produce ten beeps in succession.

```
10  0
DO
      BEEP
LOOP
```

I (-- d)

(loop index)
Where: d = the current loop index

Obtains the current count of the loop index inside the body of a DO/LOOP.

Example:

```
10  0
DO
      W." index="  I  W.  WCR
LOOP
```

The following output is produced:

```
index=0
index=1
index=2
index=3
index=4
index=5
index=6
index=7
index=8
index=9
```

NOTE
The body of the loop is executed ten times starting at 'd_2' and ending at (d_1−1)

Advanced test programs, using nested DO/LOOP constructs, can also use the 'J' command to access the second index of a DO/LOOP.

J (-- d)
(outer loop index)
Where: d = outer loop index

Obtains the current loop index of the outer DO/LOOP when nesting is used.

Example:

```
23  20
DO
      5  0
      DO
          W." J="  J  W.          ( Print value of outer index )
          W."     "               ( Leave some blank space )
          W." I="  I  W.          ( Print value of inner index )
          WCR                     ( .. and a carriage return )
      LOOP
LOOP
```

The following output is produced:

```
J=20   I=0
J=20   I=1
J=20   I=2
J=20   I=3
J=20   I=4
J=21   I=0
J=21   I=1
J=21   I=2
J=21   I=3
J=21   I=4
J=22   I=0
J=22   I=1
J=22   I=2
J=22   I=3
J=22   I=4
```

DO ... d_3 +LOOP (d_1\d_2 --) - DO
(d_3 --) - +LOOP

Where: d_1 = the ending value of the index
d_2 = the starting value of the index
d_3 = increment/decrement value

Increments/decrements the value of 'd_3' other than one to the current loop index upon each iteration.

Example:
Count backwards by 20 from 100 and output to the screen.

```
0  100
DO
   W." I="  I  W.  WCR
-20                                    ( Count backwards by 20 from 100 )
+LOOP
```

The following output is produced:

```
I=100
I= 80
I= 60
I= 40
I= 20
I=  0
```

LEAVE (--)

Forces the premature termination of a DO/LOOP. This is implemented by setting the value of the loop limit to the current value of the index.

NOTE
Execution continues within the body of the loop until either LOOP or +LOOP is reached. This means that the termination of the DO/LOOP structure is deferred until the next evaluation of the index and limit.

Example:
Check each byte in the received data frame until an 'FF' is found. When found, print its position and exit the loop.

```
REC-LENGTH @ 0
DO
      REC-POINTER @ I + @                ( Index into frame )
      C@                                 ( Get data byte )
      0xFF =
      IF
          T." Found 'FF' at position" I T.
          TCR
          LEAVE
      ENDIF
LOOP
```

BEGIN ... AGAIN (--)

(unconditional looping)

Iteratively executes the enclosed block of commands continuously.

Example:

Increment counter number one and print its new value endlessly.

```
BEGIN
      1 COUNTER1  +!
      W." Counter="  COUNTER1  @  W.  WCR
AGAIN
```

WARNING

Because there is no programmed way to exit from a BEGIN/AGAIN loop, this structure should not be used within test scripts. This command sequence is of more interest to protocol application developers. If an infinite BEGIN/AGAIN loop is inadvertently started, press CTRL/SHIFT/f8 and enter MENU.

BEGIN ... f UNTIL

(conditional looping)

Where: f = a flag to determine if looping should terminate.

Iteratively executes the enclosed commands as long as 'f' is false (equal to zero). Once 'f' becomes true (or non-zero), the BEGIN-UNTIL loop terminates, and execution continues starting with the next statement following the UNTIL.

Example:

```
0  COUNTER1  !
BEGIN
      COUNTER1  @  W.  WCR
      1  COUNTER1  +!
COUNTER1  @  5 =
UNTIL
```

The following output is produced:

```
0
1
2
3
4
```

BEGIN ... f WHILE ... REPEAT

(conditional looping)

Where: f = a flag to determine if looping should continue

Executes the test in the middle of the loop. The body of the loop is split into two parts: a pre-test block and a post-test block.

Any commands contained within the pre-test block are always executed at least once. Any commands in the post-test block are executed if the expression resulting in 'f' is non-zero.

Example:
Transmit X.25 data packets until the packet window closes. The status of the packet window is determined by executing an application command which returns a one if the window is still open.

```
BEGIN
      WINDOW?                           ( Is the X.25 packet window open? )
WHILE
      DATA                              ( Yes, send a data packet )
REPEAT
```

If the window is open, the DATA command queues a data packet to the automatic layer 2 for transmission.

7
LOGICAL COMPARISONS

Logical comparison operators can be used to decide the path of program execution as well as test values for certain bounds. These operators can be grouped together into expressions when combined with the 'OR' and 'AND' operators to widen the possibility for complex or multiple testing and comparison.

Most comparison operators return a boolean true/false flag. As explained in the previous section on program control, true is any non-zero value while false is a zero value. These true and false values are used by the program control commands (IF/ENDIF, BEGIN/UNTIL, etc.) to modify the behavior of programs.

= (d_1\\d_2 -- f)

(equality)

Where: d_1, d_2 = 32 bit numeric value to compare
f = result of comparison

Returns true if 'd_1' and 'd_2' are equal.

Example:
Compare the value in COUNTER4 to number five.

```
COUNTER4 @  5=
IF
      T." values are equal"  TCR
ENDIF
```

> (d_1\\d_2 -- f)

(greater than)

Where: d_1, d_2 = values to compare
f = result of comparison

Returns true if 'd_1' is greater than 'd_2'.

Example:
Test if the value of the MTEI* variable is greater than 63. If so, execute the first trace statement.

```
MTEI* @  63  >
IF
      T." Manual TEI frame received"  TCR
ELSE
      T." Automatic TEI frame received"  TCR
ENDIF
```

< (d_1\d_2 -- f)
(less than)
Where: d_1, d_2 = values to compare
f = result of comparison

Returns true if 'd_1' is less than 'd_2'.

Example:
Check the length of a received frame to see if it is less than 4. If true, print a trace message.

```
L2-LENGTH  @  4  <
IF
      T." Frame is very short"  TCR
ENDIF
```

BETWEEN? (d_1\$d_2$\$d_3$ -- f)
Where: d_1 = value to compare
d_2 = lower bound
d_3 = upper bound
f = result of comparison

Returns true if 'd_1' falls between or is equal to the range parameters 'd_2' and 'd_3'.

WARNING
The boundary value 'd_2' must be less than 'd_3'. If not, BETWEEN? always returns false regardless of the input value 'd_1'.

Example:
Check the length of the received frame and, depending on what range the length falls into, increment a different counter. This code fragment forms the core of many statistics applications.

```
REC-LENGTH  @
DUP  1    10  BETWEEN?  IF   1  COUNTER1  +!  ENDIF
DUP  11   20  BETWEEN?  IF   1  COUNTER2  +!  ENDIF
     21   30  BETWEEN?  IF   1  COUNTER3  +!  ENDIF
```

7.1 Combining Expressions

Comparison expressions can be combined into more complicated forms by using the AND and OR operators.

Example:
Execute the action clause if both an ISDN SETUP message is received and its call reference is equal to one.

```
M#SETUP  ?L3_MSG
$MSG-CRVALUE  @  1 =
AND
ACTION{
      ........
}ACTION
```

Example:
Check if both the contents of the received SAPI variable and the TEI are equal to 0 and 127, respectively.

```
MSAPI*  @  0 =
MTEI*  @  127 =  AND
IF
      T." Broadcast frame received"  TCR
ENDIF
```

Example:
Check if at least one comparison is true.

```
MSAPI*  @  0 =
MSAPI*  @  16 =  OR
MSAPI*  @  63 =  OR
IF
      T." Valid SAPI received"  TCR
ENDIF
```

7.2 Logical Negation

Because boolean values (true and false) are represented by a value of non-zero for true and zero for false, the sense of an expression can be inverted using the 0= command.

0= (d -- f)
(fast zero equality)
Where: d = value to compare to zero
f = result of comparison

Returns true if a value of 0 is received.

NOTE
0= is equivalent to the 'NOT' operator used in other languages.

Example:
Print a trace message if a 0 SAPI frame is received.

```
MSAPI*  @  0=
IF
   T." Signalling Frame received"  TCR
ENDIF
```

Example:
Use the same construct to detect inequality.

```
MSAPI*  @  0 =
MSAPI*  @  16 = OR
MSAPI*  @  63 = OR
0=
IF
      T." Invalid SAPI received"  TCR
ENDIF
```

Example:
Execute the trace statement if COUNTER5 *is not equal* to 255.

```
COUNTER5  @  255 =  0=
IF
      T." Wrong value received"  TCR
ENDIF
```

7.3 Miscellaneous

There are two commands that are particularly useful for range checking:

MAX (d_1\d_2 -- d_3)
(check maximum)
Where: d_1, d_2 = values to compare
d_3 = the larger of either d_1 or d_2

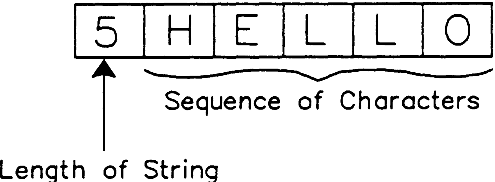
Returns the larger 'd_3' of two input values 'd_1' and 'd_2'; signed comparisons are used.

MIN (d_1\d_2 -- d_3)
(check minimum)
Where: d_1, d_2 = values to compare
d_3 = the smaller of either d_1 or d_2

Returns the smaller 'd_3' of two input values 'd_1' and 'd_2'; signed comparisons are used.

8
STRINGS

Strings are stored in memory as a one byte length field (see below), followed by a sequence of bytes containing ASCII characters. The one byte length field means that all strings have a maximum length of 255 characters.



Any references to the string returns an address pointing to the first byte of the string (i.e. the length byte).

Memory space for the string is reserved when the string is initially defined. Since memory surrounding the allocated string is reserved for other objects, the first time the string is defined fixes the maximum possible length of the string. This means that the length of a string cannot be expanded after it has been defined, nor should the length of the string be decreased.

8.1 String Manipulation

Strings can be defined with either ASCII or hexadecimal characters.

" xxxxx" (-- a)
(quote space)

Contains the address of a count-prefixed string of a length less than or equal to 80 characters if input directly from the keyboard, and 255 characters if created within an ITL program. All characters after the space character, following the opening quotation mark and before the closing quotation mark, are included in the string. If used within an ITL test program, the compiler reserves sufficient memory space to store the complete string and its length.

Example:
Transmit the ASCII contents of the string.

```
" This is a test message"  SEND
```

X" xxxxx" (-- a)
(X quote space)

Contains the address of a count-prefixed string of hexadecimal digits of a length less than or equal to 80 characters if input directly from the keyboard, and 255 characters if created within an ITL program. Valid hex characters are '0' through '9' and 'A' through 'F'. Lower case letters are also allowed.

Example:
Send a frame containing the specified hex data.

```
X" 0802000025" SENDF
```

NOTE
X" xxxxx" is left justified, eg. X" 123" SENDF (use 12 bits) has the same result as X" 1230" SENDF.

COUNT (a_1 -- a_2\b)
Where: a_1 = pointer to count-prefixed string
a_2 = pointer to first character of the string (a_1 + 1)
b = length of string

Converts a count-prefixed string to an address/length pair. Many commands used by protocol applications require a buffer address and a length as parameters.

Example:
Copy the contents of the string into a defined buffer called data-keep. COUNT is used to change the address of the string into two numbers: a pointer to the first character in the string and the length of the string.

```
0  VARIABLE data-keep  100  ALLOT          ( declare a buffer )
" This is a test string"  COUNT            ( obtain src-addr and len )
data-keep                                  ( now have src-addr len dest-addr )
SWAP                                       ( now have src dest len )
CMOVE
```

Example:
Use the T.TYPE command to print a trace report.

```
" This is another test"  COUNT  T.TYPE  TCR
```

CMOVE (a_1\a_2\n --)
(c move)
Copies the specified number of characters from one memory block starting at address 'a_1' to another memory block starting at address 'a_2' (see Figure 3-1).

This command is useful for copying the contents of a string into a buffer. For an example of this function, see the example under COUNT.

$! (string1\string2 --)
Copies string1 into string2.

Example 1:
Initialize the string DNA-A to the ASCII string 012345. The first quotation mark must be followed by one space. The space is not part of the string contents.

```
" 012345" DNA-A $!
```

Example 2:
Copy the contents of the string DNA-A to DNA-B.

```
DNA-A DNA-B $!
```

+$ (string\address\count --)
Appends *count* bytes from *address* to *string* and increments the count in *string* accordingly.

Example:
Add the characters 01 to the string DNA-A.

```
DNA-A " 01" COUNT +$
```

$= (string1\string2 -- flag)
Returns true if string1 and string2 are an exact match.

Example:
```
DNA-A DNA-B $=
IF                  < code which should be executed if strings are identical >
ELSE                < code which should be executed if strings are not identical >
ENDIF
```

?*SEARCH (a_1\n\s -- a_2\f)
(question star search)
Where: a_1 = address of search area
n = size of search area
s = count-prefixed string to search for
a_2 = pointer within the search area where s was found
f = true/false flag

Returns true if a user-defined string is found anywhere within a buffer area of specified length. If the search was successful, address 'a_2' points to the first byte *after* the desired string.

NOTE
Single character wildcards can be specified within the string by using the '?' character or hex 3F.

Example:
Search for the string login within the received data buffer of a protocol application. If the string is found, a text message is printed together with the address of where login occurs within the buffer.

```
L2-POINTER  @  L2-LENGTH  @  " login"  ?*SEARCH
IF
      W." Match was found at position"  W.  WCR
ELSE
      W." No match was found"  WCR
      DROP                                        ( get rid of address a2 )
ENDIF
```

Wildcards are used by substituting '?' with particular characters in the key string(s). For example, '?ogin' matches a sequence beginning with any character and terminating with 'ogin'. This is useful for searching for either Login or login within the data buffer.

?MATCH (a\n\s -- f)
(question match)
Where: a = address of match area
n = size of match area
s = count-prefixed string to match
f = true/false flag

Returns true if a user-defined string is found starting at the first byte within a buffer area of specified length.

NOTE
Single character wildcards can be specified within the string by using the '?' character or hex 3F.

CONV_STR (string\code -- flag)
Returns true if the specified string is converted from one code to another. The four conversion codes are:

- ASCII-TO-EBCDIC
- EBCDIC-TO-ASCII
- ASCII-TO-HEX
- HEX-TO-HEX

Example:
Convert the ASCII string "G-day mate" to EBCDIC.

```
" G-day mate" ASCII-TO-EBCDIC CONV_STR
```

8.2 Converting Numbers to Strings

A 32 bit signed number can be converted into an ASCII string representing one of four number bases for output to the screen, printer, or remote port by using the <# . . . #> construct.

ITL supports four number bases for converting numbers to strings:

- Decimal (default)
- Hexadecimal
- Octal
- Binary

BASE (-- a)
Contains a pointer to the current numerical base.

Example:
```
BASE @
```

Valid values after the @ include:

- 2 – binary
- 8 – octal
- 10 – decimal
- 16 – hexadecimal

DECIMAL (--)
Sets the current numerical base to decimal (default).

HEX (--)
Sets the current numerical base to hexadecimal.

OCTAL (--)
Sets the current numerical base to octal.

BIN (--)
Sets the current numerical base to binary.

#>STR (value\string address\base --)
(number to string)
Converts a value to ASCII representation in the specified base.

Example:
Attempt a conversion to binary.
```
21 prompt 2 #>STR
```

Attempt a conversion to octal.
```
21 prompt 8 #>STR
```

Attempt a conversion to decimal.
```
21 prompt 10 #>STR
```

Attempt a conversion to hex.
```
21 prompt 16 #>STR
```

WARNING
Sufficient memory must be reserved for storing the ASCII representation of the value (i.e. If the value to be converted is 65535 and the base is 2, a minimum of 17 bytes must be reserved).

8.3 Converting Strings to Numbers

STR># (address\n -- 0 | converted number\1)
(string to number)
Converts an address containing a count prefixed string of ASCII characters to any base. If the conversion was successful, the converted value and a true flag is returned.

Example:
Attempt a conversion to binary.
```
prompt 2 STR>#
```

Attempt a conversion to octal.
```
prompt 8 STR>#
```

Attempt a conversion to decimal.
```
prompt 10 STR>#
```

Attempt a conversion to hex.
```
prompt 16 STR>#
```

9
SCREEN DISPLAY

This section describes the commands used to output to the User, Data, or Test Script Window.

9.1 Trace Reporting in the Data Window

A trace report is a user–generated message or comment inserted into the Data Window that can be captured to RAM or disk as well as displayed on the screen.

Trace report messages contain a unique block number, the time of creation, and user–defined text. Figure 9–1 shows an example of a trace report consisting of a two line display in the Data Window.

```
MM:SS         Source    Character  ( ASCII )  Display of Frames
20:32.8581  TRACE REPORT   BLOCK NUMBER 4
This is a trace report.
>
```

Figure 9–1 Example of a Trace Report

NOTE
Only the first 38 characters of a trace statement are displayed when split display format is selected.

The Data Window can be accessed for background display under the **Background** topic or by using the following command:

SHOW_DATA (--)
Displays the Data Window.

The following commands can be used to create text in trace reports:

T." string" (--)
Inserts a character string into the trace report buffer. Maximum length of this string is 255 characters. Refer to the TCOLOR command for mixing both text and numerical information in trace reports.

Example:
Create the trace report shown in Figure 9–1.

```
T." This is a trace report." TCR
```

NOTE
T." must be followed by a single space before the string and concluded with a " (quote).

T. (n --)
Prints a value, followed by a single space, to the trace report buffer. Valid values are 0 through 4294967295 (hex FFFFFFFF).

Example:
Print the numeric value '41', followed by a space, into the trace report buffer.

```
41 T.
```

T.H (n --)
Prints a 4 byte hex value, followed by a single space, to the trace report buffer. Valid values are 0 through 4294967295 (hex FFFFFFFF).

Example:
Print the hex value '29', followed by a space, into the trace report buffer.

```
41 T.H
```

T.TYPE (string\count --)
Inserts a string of 'count' characters into the trace report buffer.

Example:
Insert 'ABC' into the trace buffer.

```
" ABC" COUNT T.TYPE
```

TEMIT (character --)
Inserts a single character into the trace report buffer.

Example:
Insert the letter 'A' in the trace buffer.

```
0X41 TEMIT
```

NOTE
Hex 41 is converted to ASCII character 'A'.

TCR (--)

Produces a carriage return in the trace report. When TCR is called, a timestamp for the trace buffer is created and the trace report is put in the display, capture RAM, and/or disk recording.

NOTE

The number of character outputs to the screen is limited by the size of the Data Window.

TCOLOR (attribute --)

Changes the color or highlighting of trace reports.

Attributes	Foreground	Background
WHI_FG	White	Black
RED_FG	Red	Black
GRN_FG	Green	Black
BLU_FG	Blue	Black
YEL_FG	Yellow	Black
MAG_FG	Magenta	Black
CYA_FG	Cyan	Black
RED_BG	Black	Red
GRN_BG	Black	Green
BLU_BG	White	Blue
YEL_BG	Black	Yellow
MAG_BG	Black	Magenta
CYA_BG	Black	Cyan
BLK_BG	White	Black
WHI_BG	Black	White
REV_VIDEO	Reverse	Reverse

Table 9–1 Color Attributes

These attributes can be OR'ed together to independently specify the foreground and background color, however, the ensuing combination might produce unpredictable results.

Example:
Print text with a red foreground and yellow background.

```
YEL_BG RED_FG OR TCOLOR
```

The following attributes can be used to modify the highlighting on units with a monochrome CRT.

Attributes	Description
DIM	Dim foreground with black background
BRITE	Bright foreground with black background
DIM INVERSE	Black foreground with dim background
BRITE INVERSE	Black foreground with bright background

Table 9–2 Monochrome Attributes

If a test script has been developed for a unit with a color CRT, it can still be used on a unit with a monochrome CRT. The following table lists the mapping of colors to highlighting attributes on a monochrome unit.

Color Attributes	Monochrome Attributes
WHI_FG	BRITE
RED_FG	BRITE
GRN_FG	DIM INVERSE
BLU_FG	DIM
YEL_FG	BRITE INVERSE
MAG_FG	DIM
CYA_FG	DIM
RED_BG	BRITE INVERSE
GRN_BG	DIM
BLU_BG	BRITE INVERSE
YEL_BG	BRITE
MAG_BG	DIM INVERSE
CYA_BG	DIM INVERSE
BLK_BG	BRITE
WHI_BG	BRITE INVERSE
REV_VIDEO	BRITE INVERSE

Table 9–3 Color to Monochrome Mapping

Likewise, a test script developed for a unit with a monochrome CRT can be used on a unit with a color CRT. The following list maps monochrome highlights to color.

Monochrome Attributes	Color Attributes
DIM	CYA_FG
BRITE	WHI_FG
DIM INVERSE	GRN_FG
BRITE INVERSE	YEL_FG

Table 9–4 Monochrome to Color Mapping

Trace reports can be turned on (default) or off in test scripts using the following commands:

RTRACE (flag --)
YES RTRACE turns the display trace reports on. NO RTRACE turns them off.

CTRACE (flag --)
YES CTRACE turns the RAM capture trace reports on. NO CTRACE turns them off.

DTRACE (flag --)

YES DTRACE turns the disk recording trace statements on. NO DTRACE turns them off.

NOTE
The appropriate filter must be activated.

Example:

```
2 STATE{
     TIMEOUT
     ACTION{
         BLU_BG TCOLOR                   ( Highlight trace in blue background )
         YES RTRACE                      ( Turn display trace reports on )
         YES CTRACE                      ( Send trace reports to capture RAM )
         YES DTRACE                      ( Send trace reports to disk recording )
         BEEP                            ( Generate audible alarm )
         T." Timer"
         TIMER-NUMBER @ T.
         T." has occurred."              ( Display - the timer that has expired )
         T."  Going to STATE 3"          ( Display - going to STATE 3 )
         TCR                             ( Output display )
     }ACTION
  }STATE
```

NOTE
Trace reports can also be selected from the Filter Setup Menu.

9.2 Displaying in the User Window

A sixteen line window is reserved for user-created displays, statistics, graphs, etc. This window completely overlays the Data Window making only one window visible at one time.

There are two distinct conditions of the User Window:

- Visible
  The User Window completely overlays the Data Window. No monitored data is shown - only information which has been created by writing to the User Window.
- Open
  Output to the User Window can only occur when the window is open. Once open, output commands direct output to the current cursor position within the User Window. It is not necessary for the window to be visible to insert data.

Accessing the User Window

The User Window can be selected for display under the **Background** topic or by using the following commands.

SHOW_USER (--)
Displays the User Window.

OPEN_USER (--)
Opens the User Window to create and position text in the User Window.

POP_USER (--)
Displays and opens the User Window. This command is equivalent to SHOW_USER followed by OPEN_USER.

CLOSE_WINDOW (--)
Closes the currently open window. When used with OPEN_USER or POP_USER, the User Window is closed and any further window commands go to the Command Window.

Creating Text

Text can be created using commands similar to those in trace reporting. Unlike trace reporting, there is no timestamp and no block number. The POP_USER command must be called first to make the text visible.

W." string" (--)
Displays a character string (maximum length is 255 characters).

Example:
Create the text in the title in Figure 9-2.

```
W." Statistics Display After One Minute"
```

NOTE
W." must be followed by a single space, then the desired string, and concluded with a " (quote).

W. (n --)
Displays a value followed by a single space. Valid values are hex 0 through hex FFFFFFFF.

Example:
Print '20' followed by a space.

```
20 W.
```

W.H (n --)
Displays a 4 byte hex value followed by a single space. Valid values are hex 0 through hex FFFFFFFF.

Example:
Print hex '29' followed by a space.
```
41 W.H
```

W.TYPE (string\count --)
Displays a string of 'count' characters.

Example:
Display the ASCII letters 'ABC'.
```
" ABC" COUNT W.TYPE
```

W.TYPE_A (string\count\color --)
Displays a string of the length specified with the attribute specified (see Section 9.1, under TCOLOR, for a list of attributes).

Example:
Display the letters 'ABC' with a blue background in the currently active window.
```
" ABC" COUNT BLU_BG W.TYPE_A
```

WEMIT (character --)
Displays a character.

Example:
Display the letter 'A' (hex value 41).
```
0X41 WEMIT
```

WCR (--)
Prints a carriage return.

Cursor Control

The defaults for cursor control are:
- the cursor is not displayed; and
- scrolling is in the downwards direction (i.e. after 16 lines of text are displayed, any subsequent lines push the display up).

The following commands can be used to position text or control the cursor.

CURSOR_ON (--)
Turns on the display of the cursor which indicates the position of the next displayed character.

CURSOR_OFF (--)
Turns off the display of the cursor.

THERE (row\column --)
Positions the text to the specified row and column specified. This command can be used to align text (see Figure 9–2).

The User Window has 16 rows and 80 columns available. Valid values are 0 through 15 for row position and 0 through 79 for column position.

Example:

```
0 20 THERE                               ( Position the title in Row 0, Column 20 )
W." Statistics Display After 1 Minute"   ( Create the text )
```

WHERE (-- row\column)
Returns the row and column position of the cursor.

Example:
Position the cursor at row 0 and column 3, print three letters, and return the position of the cursor (row 0 and column 6).

```
0 3 THERE
W." abc"
WHERE
```

WSCROLL (--)
Converts the currently selected window into a scrolling window. Text is moved upwards when the cursor reaches the bottom of the window.

WRAP (--)
Converts the currently selected window into a wraparound window. New text is inserted in the top of the window when the cursor reaches the bottom of the window.

WUP (--)
Sets the direction of cursor movement to upwards. If WCR is called, the next line of text is above the current line.

WDOWN (--)
Sets the direction of cursor movement to downwards. If WCR is called, the next line of text is below the current line.

Clearing Text

CLEAR_ROW (row --)
Clears text from the specified row. The cursor is placed at column 0 of the specified row. The color and character attributes remain and any future text entry in this row is in the original color and character set unless the attributes have been changed. Any background color remains visible as a solid color bar.

CLEAR_TEXT (--)
Removes all text from the currently active window. The cursor is placed at row 0 and column 0. The attributes remain as in CLEAR_ROW.

AUTO_CLEAR_ON (--)
Removes text in the line following a WCR (default). The attributes remain as in CLEAR_ROW.

AUTO_CLEAR_OFF (--)
Opposite of AUTO_CLEAR_ON. Text is not removed in the line following a WCR.

CLEAN_WINDOW (--)
Erases the text and color of the current window.

Color and Character Sets

PAINT (color --)
Changes the color of the entire active window leaving the text unchanged (see Section 9.1, under TCOLOR, for a list of valid color attributes).

Example:
Paint the window with a blue background.

```
BLU_BG PAINT
```

PAINT can also be used to change the character set. The default color for character sets is white foreground on a black background. Attributes which can be used to change character sets are:

Attributes	Description
ASCII (default)	Returns value for ASCII character set
EBCDIC	Returns value for EBCDIC character set
HEXSET	Returns value for hex character set
TTX	Returns value for teletex character set
JS8	Returns value for JIS8 character set

Table 9–5 Character Sets

A combination of change in color attribute and character set can be accomplished using the OR command.

Example:
Change the color attribute to a red background and the text to hex character set.
```
RED_BG HEXSET OR PAINT
```

PAINT_ROW (row\color --)
Paints the specified row. Color and character set attributes can be specified (refer to the PAINT command).

PAINT_FIELD (row\column\length\color--)
Paints an area of the specified length starting at the location given in the row and column numbers. PAINT_FIELD paints to a maximum of one screen row. Color and character set attributes can be specified (refer to the PAINT command).

Example:
Change the character set to hex in row 1 starting at column 5. The length specified in the example is greater than the number of columns remaining in row 1. The paint ends at column 79 and does not spill over into row 2.

```
1 5 90 HEXSET PAINT_FIELD
```

The following example is from an ISDN test script. Look for frames and keep statistics on the different frame types received for one minute. At the end of one minute, the user is prompted to press a function key to see the results. The screen created in state 2 is shown in Figure 9-2.

```
TCLR

0 STATE{
      ?WAKEUP
      ACTION{
          0 COUNTER1 !            ( Initialize counters )
          0 COUNTER2 !
          0 COUNTER3 !
          0 COUNTER4 !
          0 COUNTER5 !
          0 COUNTER6 !
          0 COUNTER7 !
          0 COUNTER8 !
          0 COUNTER9 !
          0 COUNTER10 !
          0 COUNTER11 !
          0 COUNTER12 !
          21 600 START_TIMER       ( Start timer 21 for 1 minute )
          SHOW_DATA                ( Show the Data Window )
          1 NEW_STATE
      }ACTION
  }STATE
```

```
1 STATE{
     R#I ?RX_FRAME              ( Check event field for I Frame )
     ACTION{
         1 COUNTER1 +!          ( Increment counter )
     }ACTION

     R#RR ?RX_FRAME             ( Check event field for RR Frame )
     ACTION{
         1 COUNTER2 +!          ( Increment counter )
     }ACTION

     R#RNR ?RX_FRAME            ( Check event field for RNR Frame )
     ACTION{
         1 COUNTER3 +!          ( Increment counter )
     }ACTION

     R#REJ ?RX_FRAME            ( Check event field for REJ Frame )
     ACTION{
         1 COUNTER4 +!          ( Increment counter )
     }ACTION

     R#SABM ?RX_FRAME           ( Check event field for SABM Frame )
     ACTION{
         1 COUNTER5 +!          ( Increment counter )
     }ACTION

     R#SABME ?RX_FRAME            ( Check event field for SABME Frame )
     ACTION{
         1 COUNTER6 +!            ( Increment counter )
     }ACTION

     R#DISC ?RX_FRAME             ( Check event field for DISC Frame )
     ACTION{
         1 COUNTER7 +!            ( Increment counter )
     }ACTION

     R#UA ?RX_FRAME               ( Check event field for UA Frame )
     ACTION{
         1 COUNTER8 +!            ( Increment counter )
     }ACTION

     R#DM ?RX_FRAME               ( Check event field for DM Frame )
     ACTION{
         1 COUNTER9 +!            ( Increment counter )
     }ACTION

     R#FRMR ?RX_FRAME             ( Check event field for FRMR Frame )
     ACTION{
         1 COUNTER10 +!           ( Increment counter )
     }ACTION
```

```
    R#UI ?RX_FRAME               ( Check event field for UI Frame )
    ACTION{
        1 COUNTER11 +!           ( Increment counter )
    }ACTION

    R#XID ?RX_FRAME              ( Check event field for XID Frame )
    ACTION{
        1 COUNTER12 +!           ( Increment counter )
    }ACTION

    21 ?TIMER                    ( Check for expiration of the T21 timer )
    ACTION{
        BEEP_ON                  ( Give audible alert )
        " Press UF1 function key to see results"
        W.NOTICE                 ( Inform user with notice )
        BEEP_OFF                 ( Turn alarm off )
        2 NEW_STATE              ( Go to next state )
    }ACTION
  }STATE

2 STATE{
    UF1 ?KEY
    ACTION{
        POP_USER                 ( Open the User Window )
        CLEAR_TEXT WHI_FG PAINT ( Clear screen text and color )
        0  20 THERE W." Statistics Display after 1 minute"
        2  30 THERE W." I     =" COUNTER1  @ W.
        3  30 THERE W." RR    =" COUNTER2  @ W.
        4  30 THERE W." RNR   =" COUNTER3  @ W.
        5  30 THERE W." REJ   =" COUNTER4  @ W.
        6  30 THERE W." SABM  =" COUNTER5  @ W.
        7  30 THERE W." SABME =" COUNTER6  @ W.
        8  30 THERE W." DISC  =" COUNTER7  @ W.
        9  30 THERE W." UA    =" COUNTER8  @ W.
        10 30 THERE W." DM    =" COUNTER9  @ W.
        11 30 THERE W." FRMR  =" COUNTER10 @ W.
        12 30 THERE W." UI    =" COUNTER11 @ W.
        13 30 THERE W." XID   =" COUNTER12 @ W.
        CLOSE_WINDOW
        TM_STOP
    }ACTION
  }STATE
```

An alternate implementation of state 1 is:

```
1 STATE{
      EVENT-TYPE @ FRAME# =
      ACTION{
          FRAME-ID @
          DOCASE
              CASE R#I { 1 COUNTER1 +! }
              CASE R#RR { 1 COUNTER2 +! }
                   .
                   .
                   .
              CASE R#XID 1 COUNTER12 +!
          ENDCASE
          1 SWAP !
      }ACTION

      21 ?TIMER                  ( Check for expiration of the T21 timer )
      ACTION{
          BEEP_ON                ( Give audible alert )
          " Press UF1 function key to see results"
          W.NOTICE               ( Inform user with notice )
          BEEP_OFF               ( Turn alarm off )
          2 NEW_STATE            ( Go to next state )
      }ACTION
  }STATE
```

For either method, the User Window is displayed as shown in Figure 9-2.

```
D Channel  :  D-Chan Monitor       Live Data          1988-09-29 09:31:34
                Statistics Display after 1 minute

                          I     = 0
                          RR    = 0
                          RNR   = 0
                          REJ   = 0
                          SABM  = 0
                          SABME = 0
                          DISC  = 0
                          UA    = 0
                          DM    = 0
                          FRMR  = 0
                          UI    = 0
                          XID   = 0
```

Figure 9-2 Statistics Display

9.3 Displaying in the Test Script Window

A three line window is reserved for user-created displays, statistics, graphs, etc. This window overlays the lower three lines of the Data Window.

There are two distinct conditions of the Test Script Window:

- Visible
  The Test Script Window overlays the lower three lines of the Data Window. No monitored data is shown – only information which has been created by writing the Test Script Window.
- Open
  Output to the User Window can only occur when the window is open. Once open, output commands direct output to the current cursor position within the Test Script Window. It is not necessary for the window to be visible to insert data.

Text can be created using the same commands available for the User Window. Refer to Section 9.2 for a description of these commands.

The Test Script Window can be selected for display under the **TestScript** topic or by using the following commands:

SHOW_TEST (--)
Displays the Test Script Window.

OPEN_TEST (--)
Opens the Test Script Window to create and position text in the Test Script Window.

DROP_TEST (--)
Closes the Test Script Window.

10
PRINTER PORT CONTROL

Information can be sent to an attached printer in almost the same way as to a CRT device.

NOTE
Before using any printer commands, ensure that the printer is connected on–line, and that the tester has been properly configured for baud rate, handshaking, etc.

10.1 Configuring the Printer Port

When the menu system software is loaded, the default configuration source file (HOME.D) is executed which automatically configures the printer port. Refer to Section 18.

I/F_TYPE (port \ printer type --)
Where: port = PRINTER
printer type = V28 (serial printer)
PARALLEL (parallel printer)

Sets the printer port to serial (default) or parallel.

PRINTER (-- value)
Identifies the printer port for configuration commands.

TX_SPEED (port \ speed --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the transmitter baud rate for the specified asynchronous port. Valid speed values are 50 through 19200 (default value is 1200).

RX_SPEED (port \ speed --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the receiver baud rate for the specified asynchronous port. Valid speed values are 50 through 19200 (default value is 1200).

TX_CONTROL (port \ control type --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the transmitter control mode for the specified asynchronous port. Valid control modes are:

OFF	Flow control is off (default)
DTR	Reads DTR for printer; asserts DTR for modem
XOF	XON / XOFF control
CTS_FLOW	CTS/RTS control (CTS for modem; RTS for printer)

NOTE
Combinations of DTR, XON/XOFF, and CTS/RTS are allowed.

Example:
Configure the printer to use DTR and XON/XOFF control.
```
PRINTER DTR XOF OR TX_CONTROL
```

PRINTER_EOL (end of line condition --)
Determines the sequence generated for an end of line condition. Valid conditions are:

LFCR	Line feed is generated followed by a carriage return (default)
NONE_EOL	End of line is not generated automatically
CR_EOL	A carriage return is generated

PRINTER_MODE (mode --)
Determines the format of printer output. Valid modes are:

CHAR_DISPLAY	Outputs all unprintable characters in the form <AB>, where AB are the two nibbles represented in hex notation (default)
RAW_DISPLAY	Outputs all characters without any range checking
HEX_DISPLAY	Outputs all characters in hex format <AB>

CHARS/LINE (port / # of characters --)
Where: port = PRINTER
# of characters = a decimal value

Sets the number of characters per line (default 80) and generates an end of line sequence after the specified number of characters.

LINES/PAGE (port / # of lines --)
Where: port = PRINTER
# of lines = a decimal value

Sets the number of lines per page and generates a form feed after the specified number of lines.

CONFIG (port --)
Configures the specified port.

Example:
Configure the printer port for a serial printer with 9600 baud rate, XON/XOFF flow control, hex output, 132 characters per line and 40 lines per page.

```
PRINTER >                       ( Pinter port to return stack )
R V28 I/F_TYPE                  ( Configure for a serial printer )
R 9600 TX_SPEED                 ( Set transmitter speed )
R 9600 RX_SPEED                 ( Set receiver speed )
R XOF TX_CONTROL                ( XON/XOFF control )
HEX_DISPLAY PRINTER_MODE        ( Hex output )
R 132 CHARS/LINE                ( 132 characters/line )
R 40 LINES/PAGE                 ( 40 lines per page )
R> PRINTER CONFIG
```

Printer Output Commands

The printer output commands are similar to the trace report commands. Text is collected in a print buffer until a PCR command is issued. The maximum size of this buffer is 136 characters and the default number of characters printed on one line is 80.

P." string" (--)
Inserts a character string into the print buffer.

NOTE
P." must be followed by a single space before the string and terminated with a " (quote).

P. (n --)
Prints a value, followed by a single space, to the print buffer.

P.H (n --)
Prints a 4 byte hex value, followed by a single space, to the print buffer.

P.TYPE (string\count --)
Inserts a string of 'count' characters into the print buffer.

Example:
Insert the string 'Job complete' in the print buffer.
```
" Job complete" COUNT T.TYPE
```

PCR (--)
Forces a carriage return. When PCR is called the contents of the print buffer are sent to the printer.

PEMIT (character --)
Inserts a single character into the print buffer.

PRINT_SCREEN (--)
Transmits an image of the current CRT screen to the attached printer.

PRT-L (-- address)

Contains the number of characters per line. Once this number has been reached, the tester automatically issues a PCR.

Example:
Set the value in PRT-L to the same number of characters/line as selected on the Home processor.

```
60 PRT-L !
```

11
REMOTE PORT CONTROL

IDACOM's testers have four methods of remote control:

- ITL command entry from a terminal or PC
- Remote Test Package (RTP)
- R-FILEX™
- FILEX

This section of the manual applies only to ITL commands entered remotely from a terminal or PC and cannot be used with the other three methods of remote control.

The remote control/modem port on the back of the tester is used to transmit and receive asynchronous data on the Home processor or application processor. This port supports asynchronous serial RS-232C modems. To use an application processor, load an application and switch to that processor.

NOTE
The tester behaves like a terminal. If remote control is performed by a DTE device, a null modem is required.

11.1 Configuring the Remote Port

When the menu system software is loaded, the default configuration source file (HOME.D) is executed which automatically configures the remote port. Refer to Section 18.

REMOTE (-- value)
Identifies the remote port for configuration commands.

TX_SPEED (port \ speed --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the transmitter baud rate for the specified asynchronous port. Valid speed values are 50 through 19200 (default is 1200).

RX_SPEED (port \ speed --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the receiver baud rate for the specified asynchronous port. Valid speed values are 50 through 19200 (default is 1200).

TX_CONTROL (port \ control type --)
Where: port = REMOTE (modem port)
PRINTER (printer port)

Sets the transmitter control mode for the specified asynchronous port (REMOTE indicates to modem port or PRINTER indicates to printer port). Valid control modes are:

OFF	Flow control is off (default)
DTR	Reads DTR for printer; asserts DTR for modem
XOF	XON / XOFF control
#CTS_FLOW	CTS/RTS control (CTS for modem; RTS for printer)

NOTE
Combinations of DTR, XON/XOFF, and CTS/RTS are allowed.

Example:
Configure the remote port to use DTR and XON/XOFF control.

```
REMOTE DTR XOF OR TX_CONTROL
```

CONFIG (port --)
Configures the specified port.

Example:
Configure the remote port for 9600 baud rate with XON/XOFF flow control.

```
REMOTE 9600 TX_SPEED      ( Set Transmitter speed )
REMOTE 9600 RX_SPEED      ( Set receiver speed )
REMOTE XOF TX_CONTROL     ( XON/XOFF control )
REMOTE CONFIG
```

TURN_ON (port \ lead id --)
Where: port = REMOTE (modem port)
RMT_DTR (data terminal ready)
RMT_RS (request to send)

Turns on the specified lead on the remote port.

Example:
Turn on the request to send leads.

```
REMOTE RMT_RS TURN_ON
```

TURN_OFF (port \ lead id --)
Where: port = REMOTE (modem port)
RMT_DTR (data terminal ready)
RMT_RS (request to send)

Turns off the specified lead on the remote port.

11.2 ASCII Terminal

The remote control line is configured to use async, ASCII, 8 bits, no parity, and 1200 baud. The tester does not echo received characters. The terminal should be set to half duplex for local display.

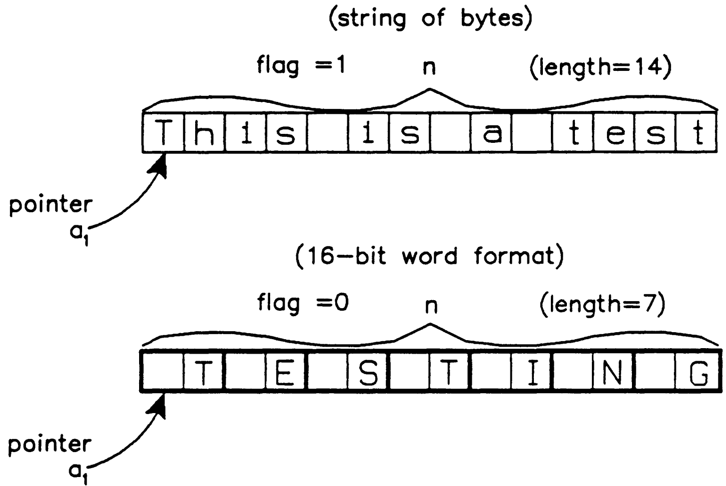
Commands on the ASCII terminal are entered, followed by a carriage return. The program processes the input, and sends the command line display to the remote site followed by the prompt 'ROK' and the carriage return and linefeed characters. The application processes the remote commands the same way as if they had been entered on the local keyboard. Either 'R-OK' or '???' and a carriage return and linefeed are sent to the terminal.

Sending Strings

Text can be transmitted to the remote port from either the Command Window or a test script (on an application processor).

REMOTE_OUT (a_1\n\flag -- f)

Transmits the string 'a_1' of length 'n' out the remote port. The 'flag' indicates the data format as either one byte (flag=1) or two byte (flag=0) characters. A 0 is returned if the transmission was successful.



WARNING

The status is returned on the computational stack. If never used, the stack grows by one each time REMOTE_OUT is called.

REMOTE_OUT_W (a_1\n\flag --)

Transmits the string 'a_1' of length 'n' out the remote port and waits for transmission to complete. The 'flag' indicates the data format as either one byte (flag=1) or two byte (flag=0) characters. This command is used when internal flow control is desired.

RCR (--)

Transmits a carriage return and line feed to the remote port.

Example:
Send the text 'abc' via the remote port.

```
" abc"  COUNT  YES  REMOTE_OUT  RCR
" abc"  COUNT  YES  REMOTE_OUT_W  RCR
```

IDACOM Logo

The initial display of the IDACOM logo, requiring the user to press ↵ (RETURN), can be disabled.

UNLOCK_LOGO (--)

Displays the IDACOM logo at boot-up and directly enters the Home menu.

LOCK_LOGO (--)

Displays the IDACOM logo at boot-up. The user must press ↵ (RETURN), to continue (default).

Remote Screen Display

RLINE (b --)

Where: b = the row number

Transmits the specified line on the screen to the remote port. Valid rows are 0 through 22, where 0 is the top line and 22 the bottom line.

RSCREEN (--)

Transmits the entire 23 line display to the remote port.

WARNING

Special characters have been used in creating the screen display. Thus, the display at the remote end might not match exactly.

Remote Directory Listing

A listing of the directory of the selected drive on the remote machine can be obtained in short or long form.

RMT_DIR (--)
Produces a short form listing of the selected directory.

RMT_DIRL (--)
Produces a complete (long) form listing of the selected directory.

Data Playback

Data files or capture RAM data from an application processor can be played back via the remote port. The RMT_ON command must be executed first.

RMT_ON (--)
Enables data playback via the remote.

RMT_OFF (--)
Disables data playback via the remote.

Example:
Obtain data playback from the capture RAM.

```
FROM_CAPT                    ( Capture RAM is source )
HALT                         ( Go to off-line mode )
```

Obtain data playback from disk.

```
PLAYBACK
```

To change the default disk drive and playback title, the PLAYBACK command must be followed immediately by the filename.

Example:
Playback a file on drive 0 with the filename of DATA1.

```
PLAYBACK  DR0:DATA1
```

The following commands control display scrolling.

TOP (--)
Positions the display at the beginning of the playback source

BOTTOM (--)
Positions the display at the end of the playback source.

FORWARD or **F** (--)
Scrolls one line forward on the screen.

BACKWARD or **B** (--)
Scrolls one line backward on the screen.

SCRN_FWD or **FF** (--)
Scrolls one page forward on the screen.

SCRN_BACK or **BB** (--)
Scrolls one page backward on the screen.

NOTE
Reports are displayed correctly when scrolling forward but appear reversed when scrolling backward.

The entire contents of a data file or capture RAM can be sent via the remote.

TRANSFER (--)
Transfers data from the selected data source.

QUIT_TRA (--)
Abruptly terminates the transfer of data from capture RAM to disk.

See the appropriate Programmer's Manual for more information on these commands.

FULL_DUPLEX (--)
Sets the communication mode to full duplex (default). Keyboard input is not locally echoed to the terminal screen. The host must be set up to echo the keyboard input back to the terminal.

HALF_DUPLEX (--)
Sets the communication mode to half duplex. Keyboard input is locally echoed to the Terminal Emulator screen.

EOL=CR (--)
Transmits a carriage return character when ↵ (RETURN) is pressed.

EOL=CRLF (--)
Transmits both a carriage return and a linefeed character (default) when ↵ (RETURN) is pressed.

11.3 File Transfer

DEST_DRIVE (--)
Sets the current device as the destination drive to store files received over the remote port (default is DR0).

Example:
```
WD3 DEST_DRIVE
```

CRC_CORRECTION (--)
Uses the CRC error correction scheme (default) during receive file transfers.

CHECKSUM_CORRECTION (--)
Uses the CHECKSUM error correction scheme during receive file transfers.

TRANSLATE_ON (--)
Translates files during file transfer. When transmitting files from the tester, end-of-line markers are added and IDACOM character attributes are removed. When receiving files on the tester, end-of-line markers are replaced with blank character padding and character attribute bytes are added.

NOTE
Translate should only be used when transferring ASCII files between a tester and another computer/tester.

TRANSLATE_OFF (--)
Files are not translated during file transfer (default).

CONTROL-Z=EOF (--)
Uses the control-Z character as the end-of-file marker for a receive file transfer (default). All characters after control-Z are dropped.

CONTROL-Z<>EOF (--)
The control-Z character is not recognized as the end-of-file marker for a receive file transfer.

RCV-TIMEOUT (-- address)
Contains the time, in tenths of seconds, the tester waits for another computer/tester to transmit a data packet during a receive file transfer (default is 8 seconds).

Example:
Set the timeout for a receive file transfer to 10 seconds.
```
100 RCV-TIMEOUT !
```

SEND-TIMEOUT (-- address)
Contains the time, in tenths of seconds, the tester waits for another computer/tester the acknowledge after transmitting a data packet. (default is 80 seconds).

Example:
Set the timeout for a send file transfer to 60 seconds.
```
600 SEND-TIMEOUT !
```

T-LIMIT (-- address)
Contains the number of times to retransmit files after receiving no acknowledgement (default is 10).

Example:
Set the number of retries to 20 attempts.

```
20 T-LIMIT !
```

Sending and Receiving Files

Disk files can be transmitted from the tester through the remote modem port to a remote computer. The remote computer must use the XMODEM protocol to receive files. The source file is unaffected by the transfer.

Files received over the remote port from a remote computer can be stored on a tester.

SEND_FILEX (filename --)
Transmits the specified file over the remote port using the XMODEM protocol.

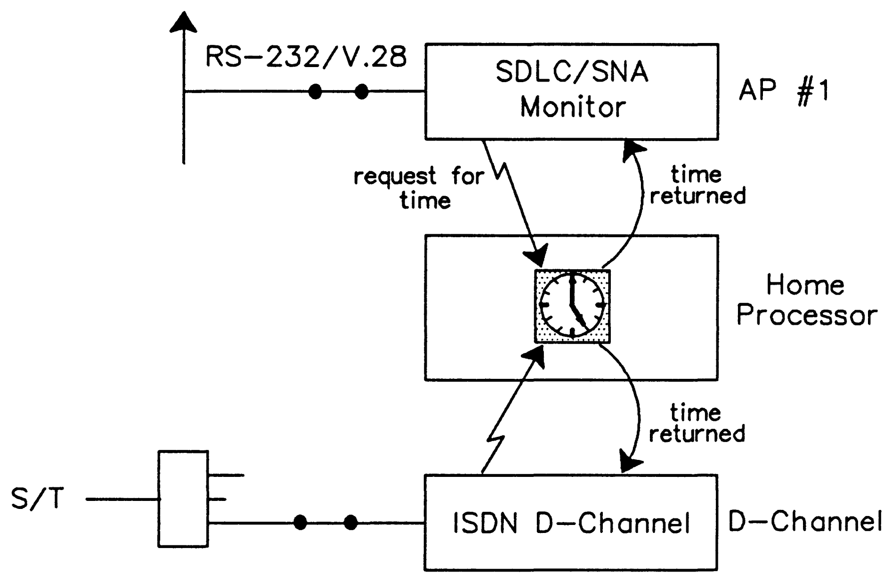
RECEIVE_FILEX (filename --)
Specifies the filename in which to store received files.

WARNING
If the specified file already exists on the destination drive, the old file will be overwritten.

NOTE
If the destination filename is not specified, a filename will automatically be assigned by the tester. For files transferred between two testers (with TRANSLATE_OFF), the source filename is used as the destination filename. Other files are assigned the filename 'USER.nn', where 'nn' is a unique and sequential number.

12
TIMESTAMPS

All IDACOM testers record the exact starting and ending time of each received frame, lead change, and trace message. These timestamps are derived from a central timebase on the Home processor and upon receipt of a frame, the application processor requests the time from the central timebase as illustrated below.



All received data is stored into a buffer by the operating system. A pointer to this buffer is placed into a queue between the operating system and the application program.

When the first byte of a frame is received, the time is recorded in the buffer. As data is received by the operating system it is stored into the buffer and the time of the last character is recorded.

NOTE
By default, recording of start of frame timestamps is on for all applications except SS#7.

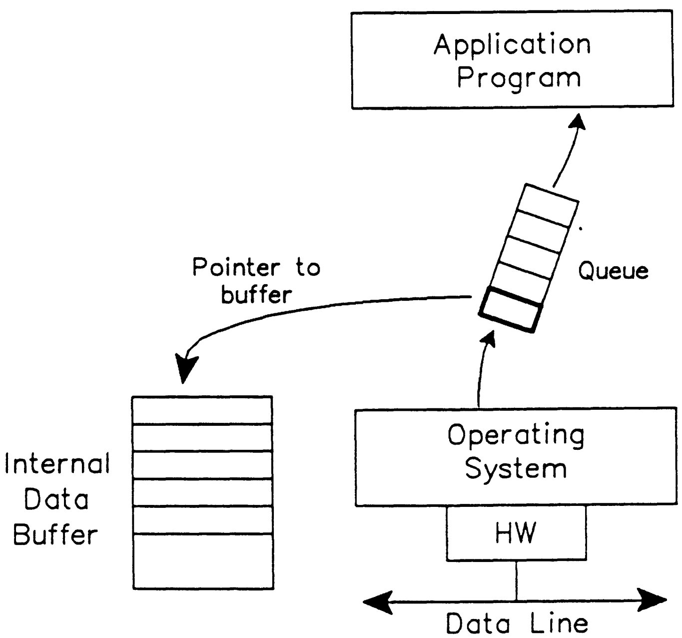


Figure 12–1 Data Queuing

Upon reception of a frame, the stored information includes:

- port identifier;
- start of frame timestamp;
- data length;
- data;
- data status; and
- end of frame timestamp.

NOTE
Additional information stored in the buffer depends upon the event type.

The application program stores this entire buffer into RAM or disk during recording and therefore, all information is available for analysis during playback. Each time a frame, lead change, or trace statement is received, or the cursor is placed over it during playback, the application software decodes all the data within the buffer. This includes setting protocol and layer 1 variables such as START–TIME.

Since all timestamping information comes from a centralized source, there is a high degree of time correlation between data streams on different interface ports.

Internally, timestamps are represented as a 48 bit/6 byte value which equal the number of 0.5 microsecond increments from a base value. The timebase increments the master clock once every half microsecond.

Because the stack used in ITL can only manipulate 32 bit values at one time, all timestamp manipulation is done using pointers to timestamps which are stored in memory.

12.1 Timestamp Conversion

The following commands convert between 48 bit timestamp format and other formats.

GET_TSTAMP_MICRO (a -- $n_1\backslash n_2\backslash n_3\backslash n_4\backslash n_5\backslash n_6\backslash n_7$)

Where: a = address of timestamp
n_1 = microseconds
n_2 = seconds
n_3 = minutes
n_4 = hours
n_5 = day
n_6 = month
n_7 = year

Converts a 48 bit value into seven values on the stack which correspond to the real calendar time.

GET_TSTAMP_MILLI (a -- $n_1\backslash n_2\backslash n_3\backslash n_4\backslash n_5\backslash n_6\backslash n_7$)

Where: a = address of timestamp
n_1 = milliseconds
n_2 = seconds
n_3 = minutes
n_4 = hours
n_5 = day
n_6 = month
n_7 = year

Converts a 48 bit value into seven values on the stack which correspond to the real calendar time.

Example:
Print the full date of the last character of the last received frame. i.e. 1989 5 15 11 45 55 76 (15 May 1989, 11:45:55:076)

```
END-TIME GET_TSTAMP_MILLI
T. T. T. T. T. T. T. TCR
```

12.2 Timestamp Arithmetic

TSTAMP_ADD ($a_1 \backslash a_2 \backslash a_3$ --)
Where: a_1 = address of first timestamp
a_2 = address of second timestamp
a_3 = address to put sum

Adds two timestamps in addresses 'a_1' and 'a_2' and stores the result into the memory location pointed to by 'a_3'.

WARNING
Pointer 'a_3' must point to memory space which has been previously reserved.

Example:
Sum the timestamps in the END-TIME and START-TIME variables and place the answer into result.

```
0  VARIABLE  result  2  ALLOT                    ( Reserve 6 bytes for answer )
END-TIME  START-TIME  result  TSTAMP_ADD
```

TSTAMP_SUB ($a_1 \backslash a_2 \backslash a_3$ -- f)
Where: a_1 = address of first timestamp
a_2 = address of second timestamp
a_3 = address of difference $a_1 - a_2$
f = success flag

Subtracts timestamps 'a_2' from 'a_1' and forms the difference 'a_3' and stores the result into the memory location pointed to by 'a_3'. TSTAMP_SUB returns 0 if $(a_1 - a_2) \geq 0$. If $(a_1 - a_2) < 0$, the subtraction is not performed and −1 is returned.

WARNING
Pointer 'a_3' must point to memory space which has been previously reserved.

Example:
Subtract the timestamps in the END-TIME and START-TIME variables and place the answer into result.

```
0  VARIABLE  result  2  ALLOT                  ( Reserve 6 bytes for answer )
END-TIME  START-TIME   result  TSTAMP_SUB
```

12.3 Copying Timestamps

Timestamps are sequences of six bytes stored in memory and can be manipulated just as any other memory data is manipulated.

To copy a timestamp, all six bytes must be moved from their current location to the destination location. Use the CMOVE command to copy a timestamp as described in Section 3.3.

Example:
Reserve space to hold the timestamp and then copy the timestamp from the area pointed to by END-TIME to the newly declared ts-save variable.

```
0  VARIABLE   ts-save  2  ALLOT                  ( Reserve six bytes for timestamp )
END-TIME  ts-save  6  CMOVE                      ( Move value of timestamp)
```

12.4 Miscellaneous

GET_TS (a --)
Where: a = pointer to 6 byte space

Stores the current time of day, using the 48 bit timestamp format, at the memory address as specified by the pointer 'a_1'. This command is useful when the current time must be compared to the time of a received event.

Example:
Reserve space for the current time of day and then use the GET_TS command to store the current time of day into the reserved memory area. Compare to determine which timestamp is greater.

```
0  VARIABLE  time  2  ALLOT                ( Reserve space )
time GET_TS
END-TIME  time  TSTAMP_COMP
-1 =
IF
      T." End time is less than the current time"  TCR
ENDIF
```

TSTAMP_COMP (a_1\a_2 -- n)
Where: a_1, a_2 = pointers to timestamps for comparison
n = result of comparison

Compares the timestamps pointed to by 'a_1' and 'a_2' and returns the result 'n'.

- -1 if $timestamp_1 < timestamp_2$
- 0 if $timestamp_1 = timestamp_2$
- +1 if $timestamp_1 > timestamp_2$

13
OPERATING SYSTEM

This section describes operating system commands which might be of interest to ITL programmers.

WARNING
The commands in this section might cause the application to crash or cause unpredictable results if used improperly.

13.1 Port Identification

Events (frames, lead changes) received at the physical interface are passed by the operating system to the application program. The application extracts information for each received frame/lead and stores it in a set of variables, one of which is a port identifier variable (PORT-ID). The lower two bytes of the port identifier variable contain a one-byte processor identifier and a one-byte line side indicator.

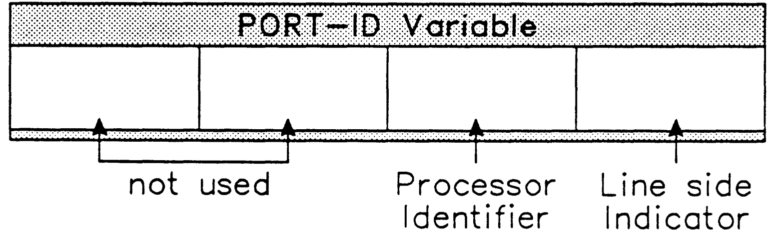


Figure 13-1 Port Identifier Variable

NOTE
The PORT_ID variable is not available in SS#7. Refer to the SS#7 Programmer's Manual.

The contents of the processor ID vary depending on the machine configuration. Table 13–1 shows the relationship between machine configuration, application processors, and port ID value.

	D–Chan	WAN	PRA	PRA/WAN	BRA	BRA/WAN	BRA/BRA	PRA/BRA/ WAN	WAN/WAN
AP #1		0x00	0x02	0x00	0x00	0x00	0x00	0x00	0x00
AP #2			0x03	0x02	0x01	0x01	0x01	0x01	0x01
AP #3	0x07			0x03	0x07	0x07	0x07	0x07	
AP #4							0x02	0x02	
AP #5							0x03	0x03	
AP #6							0x04		

Table 13–1 Port Identifier Values

For line side identification, two system constants are defined.

TO_DCE_RX (-- d)
Indicates received data was destined for the DCE (or NT in the case of ISDN) if contained in the line side indicator byte of the PORT–ID variable.

TO_DTE_RX (-- d)
Indicates received data was destined for the DTE (or TE in the case of ISDN) if contained in the line side indicator byte of the PORT–ID variable.

NOTE
The PORT–ID variable is set by the application each time a frame is received or the cursor is placed over it during playback. See the appropriate Programmer's Manual for a description of PORT–ID.

For a test script to identify which port and side a particular frame came from, it is necessary to mask either the processor ID or the side ID and then perform a comparison test on the result.

Example:
Identify which side of the line any type of frame was received on using the Universal Simulation/Monitor application and print a trace report in the Data Window.

```
" ?"  ?RECEIVED
ACTION{
    PORT-ID  @  0xFF  AND         ( Get variable, mask upper byte )
    TO_DCE_RX  =
    IF
        T." Frame received from the TO_DCE side"  TCR
    ELSE
        T." Frame received from the TO_DTE side"  TCR
    ENDIF
}ACTION
```

Example:
Print the processor ID value for each received frame.

```
" ?" ?RECEIVED
ACTION{
    T." Frame from port"
    PORT-ID @ 8 >>#          ( Shift value )
    0xFF AND                 ( Mask value )
    T.                       ( Print value )
    TCR
}ACTION
```

13.2 Audible Alarms

The beeper operates with a variable frequency tone and an adjustable duration. When the BEEP or BEEP_ON commands are executed, the frequency and duration are read from the BEEP_TONE and BEEP_DUR variables.

BEEP (--)
Generates a single beep.

BEEP_ON (--)
Generates a continuous sequence of beeps.

BEEP_OFF (--)
Stops the continuous sequence of beeps.

BEEP_TONE (-- a)
Contains the value used to set the tone of the beeper.

Example:
Set the frequency to 30 units.

```
30  BEEP_TONE  !
```

BEEP_DUR (-- a)
Contains the value, in tenths of seconds, used to set the duration of the beeper tone (i.e. On/Off timer).

Example:
Set the duration to 500 msec.

```
5 BEEP_DUR !
```

TONE (n --)
Sets the tone of the beeper directly in hardware without affecting the value saved in BEEP_TONE.

13.3 Machine Shutdown

SHUTDOWN (--)
Halts all activity on all application processors, returns the hard disk head to track zero (parked position), and resets all peripheral devices to their inactive state.

REBOOT (--)
Reboots the tester by reloading the operating system and initial Home processor software. This is equivalent to pressing the reset button with the exception that self test procedures are not performed.

13.4 Drive Selection

IDACOM testers come with a variety of options for disk storage. Units can have one or two 3.5 inch floppy drives (depending on the model and configuration) and, optionally, a 10, 20, or 40 Mb hard disk drive.

The hard disk, regardless of total size, can be partitioned into up to eight *logical partitions.* These partitions appear to the user as separate devices or directories.

Disk operations can have the actual drive explicitly specified (as part of the filename) or the operations can use the currently selected drive or drive partition.

The following commands select which drive or hard disk partition is used with subsequent disk operations.

DR0, DR1 (--)
Selects either floppy drive 0 or 1

WD0, WD1, WD2, WD3
WD4, WD5, WD6, WD7 (--)
Selects the corresponding hard disk partition.

These commands are available on all machine configurations. However, if a particular device or partition does not exist, the message 'Non-existent I/O device' is displayed and the command fails.

13.5 File Access

Disk files can be used to store information specific to a user's application. These files, called 'user' files, are organized as one or more 'blocks' of 512 bytes.

The file size can be specified upon creation. If not specified, a file is created which occupies the largest contiguous free space on the disk.

The contents of a file can only be accessed if the file is either opened or created. Once done, the file is subsequently referenced by a file descriptor (an integer assigned by the operating system to keep track of each open file).

Read and write operations require the file descriptor to identify the file. Data to read/write must be stored in a buffer of at least 512 bytes.

The success or failure of all file operations is stored as a status number in FDSTATUS.

FDSTATUS (--status)

Returns the status of the previous disk command; 0 indicates no error, and a negative number is an error indication. These values are:

Status Code	Description
-3	File does not exist
-4	Unrecoverable physical I/O error
-5	Nonexistent I/O device
-6	Bad file descriptor
-7	File access prohibited
-8	File already exists
-9	Error in DIO
-10	File name error
-11	End of file
-12	No space in filesystem for another file
-13	Too many files open
-14	Cannot create a new filesystem - existing filesystem has open files
-15	Floppy disk not loaded
-16	No RTE file system
-17	Disk is write protected
-18	Can't copy file to itself
-19	Bad source device
-20	Bad destination device
-21	No match for wildcards
-22	Compared files are different
-23	File system is write protected

CREATE_FILE (filename\access\size\type -- file desc)

Where: filename = string containing the name of the file

access = the access mode (one of the following)

- RE - read only
- WR - write only
- REWR - read/write

size = the number of 512 byte blocks. If 0 is used, the maximum contiguous free space is used.

type = one of the following constants:

- DATA_FILE (data file)
- SRC_FILE (source file)
- USR_FILE (user file)

file desc = the file descriptor (if the operation fails the returned value is undefined)

Creates a 512 byte (1 block) file on the currently selected device.

OPEN_FILE (filename\access -- file desc)
Where: filename = the name of the file
access = the access mode
file desc = the file descriptor (if the operation fails the returned value is undefined)

Opens an existing file for subsequent read/write operation.

CLOSE_FILE (truncate flag\file desc --)
Where: truncate flag =
- TRUNCATE – truncates the file at the number of written blocks.
- NO_TRUNCATE – maintains the file length as specified in the CREATE_FILE command.

file desc = the file descriptor

Closes a file and ensures all data is properly written to the disk.

WRITE_BLOCKS (file desc\buffer address\no. of blocks --)
Where: file desc = the file descriptor
buffer address = the starting address of the memory to be written to the file
no. of blocks = the specified number of blocks

Writes the specified number of blocks from memory to the indicated file.

READ_BLOCKS (file desc\buffer address\no. of blocks --)
Where: file desc = the file descriptor
buffer address = the starting address of the memory block to copy into
no. of blocks = the specified number of blocks

Reads the specified number of blocks.

SEEK (file desc\offset --)
Where: file desc = the file descriptor
offset = the number of blocks from the start of the file

Positions a number of blocks from the start of the specified file.

CLOSE (--)
Closes and truncates to the number of written blocks the last opened file.

RENAME (old\new --)
Where: old = a string containing the name of the file
new = a string containing the new name of the file

Changes the name of a file. If the drive is not specified in the string, the currently selected device is used.

Example:
Change the name of the files: TST.F and JUNK to TST.F.OLD and JUNK2, respectively.

```
" TST.F"   "TST.F.OLD"   RENAME
" DR0:JUNK"   " DR0:JUNK2"   RENAME
```

NOTE
The specified drive must be the same for both the old and new names (i.e. " DR0:JUNK" " WD0:JUNK.OLD" RENAME is illegal).

REMOVE (filename --)
Removes the specified file. This command does not remove system files or files on a protected file system.

Example 1:
Create a user file containing the text 'ABC'.

```
0 VARIABLE USER_BLOCK 508 ALLOT        ( Allot a 512 byte buffer )
" ABC" COUNT USER_BLOCK
SWAP CMOVE                             ( Move text to buffer )
" MYFILE" REWR
USR_FILE CREATE_FILE                   ( Create file )
FDSTATUS 0=                            ( Was there a disk error )
IF                                     ( No )
    USER_BLOCK 1 WRITE_BLOCKS          ( Write one block )
    TRUNCATE SWAP CLOSE_FILE           ( close the file )
ELSE                                   ( Disk error occurred )
    DROP                               ( Remove file descriptor from stack )
ENDIF
```

NOTE
One block is 512 bytes in length.

Example 2:
Open and read 'MYFILE'.

```
" MYFILE" RE OPEN_FILE                 ( Open the file )
DROP                                   ( Remove file descriptor from stack )
FDSTATUS 0=                            ( Was there a disk error )
IF
    USER_BLOCK 1 READ_BLOCK            ( Read one block )
    CLOSE                              ( Close the file )
ENDIF
```

13.6 Switching Between Processors

SWITCH (processor id --)

Transfers complete control of keyboard, remote port, and display to the designated processor. The following table shows the relationship between processor id, application processors, and machine configuration.

	D-Chan	WAN	PRA	PRA/WAN	BRA	BRA/WAN	BRA/BRA	PRA/BRA/ WAN	WAN/WAN
AP #1		FEF_1	FEF_3	FEF_1	FEF_1	FEF_1	FEF_1	FEF_1	FEF_1
AP #2			FEF_4	FEF_3	FEF_2	FEF_2	FEF_2	FEF_2	FEF_2
AP #3	FEF_7			FEF_4	FEF_7	FEF_7	FEF_7	FEF_7	
AP #4							FEF_3	FEF_3	
AP #5							FEF_4	FEF_4	
AP #6							FEF_5		
Home	MFTH								

Table 13-2 Processor Identifier Values

Example:
Switch to AP #2 on a WAN/WAN machine.

```
FEF_2 SWITCH
```

All application programs provide the following commands to switch to other processors.

MAIN (--)
Switches to the Home processor.

CPU1 (--)
Switches to FEF_1 processor.

CPU2 (--)
Switches to FEF_2 processor.

CPU3 (--)
Switches to FEF_3 processor.

CPU4 (--)
Switches to FEF_4 processor.

CPU5 (--)
Switches to FEF_5 processor.

CPU7 (--)
Switches to FEF_7 processor.

FTH (-- processor id)
Returns the processor id of the current application processor.

Example:
Switch to the *other* application processor on a WAN/WAN machine.

```
FTH FEF_1 =
IF
      FEF_2 SWITCH
ELSE
      FEF_1 SWITCH
ENDIF
```

14
COMPILER CONTROL

There are a number of program control statements which affect the execution of the ITL compiler as opposed to the run-time execution of the program itself.

These commands are useful for eliminating duplicate definitions of commands and variables to be used in a program.

14.1 Conditional Compilation

```
#IF  (expression)  #IS_TRUE
      ...
#ELSE
      ...
#ENDIF
```

This structure allows the evaluation of expressions during compile-time, and based on the outcome of those expressions, the execution flow of the compiler can be modified. The expression must return a single true/false flag on the top of the stack.

The #IF command must be combined with either the #IS_TRUE or #IS_FALSE commands in order to complete expression evaluation.

Example:

```
#IF  FTH  FEF_1 = #IS_TRUE                    ( are we using AP #1 now? )
      1  1  CPU2_MAIL                         ( yes )
#ELSE
      1  1  CPU1_MAIL                         ( no )
#ENDIF
```

14.2 Conditional Definition

```
#IFDEF   command
     ...
#ELSE
     ...
#ENDIF

#IFNOTDEF   command
      ...
#ELSE
     ...
#ENDIF
```

These structures allow the definition of commands that have not yet been defined, or the prevention of multiply defined commands and/or variables.

Example:
Check if the variable 'result' exists; if not previously defined, proceed to execute the enclosed code. The enclosed code defines the variable.

```
#IFNOTDEF   result
      0  VARIABLE  result
#ENDIF
```

15
STACK OPERATIONS

As previously discussed, the ITL language is based upon the primitives and computational model of FORTH. Like FORTH, ITL has two Last In First Out (LIFO) stacks which are integral to its operation.

The computational stack is used to transfer all parameters in and out of ITL commands. When passing parameters to a command, parameters are entered into the stack for later processing by the command.

Additionally, the return stack can be used for occasional storage of temporary values during program execution. Several parameters can be pushed onto the return stack, however, it is extremely important that all values placed onto the return stack be removed before the current code segment finishes.

STACK (--)
Displays all values, currently on the computational stack, in the Command Window.

DUP (d_1 -- d_1\d_1)
Duplicates the value on the top of the stack.

DROP (d_1 --)
Deletes the value on the top of the stack.

2DUP (d_1\$d_2$ -- d_1\$d_2$\$d_1$\$d_2$)
Duplicates the pair of values on the top of the stack.

2DROP (d_1\d_2 --)
Deletes two values on the top of the stack.

3DUP (d_1\$d_2$\$d_3$ -- d_1\$d_2$\$d_3$\$d_1$\$d_2$\$d_3$)
Duplicates the top three values on the top of the stack.

3DROP (d_1\$d_2$\$d_3$ --)
Deletes the top three values on the top of the stack.

?DUP (d_1 -- d_1\d_1 | 0)
Duplicates the value on the top of the stack if it is non-zero; otherwise returns 0.

SWAP (d_1\$d_2$ -- d_2\$d_1$)
Switches the places of the first two entries on the stack.

ROT (d_1\$d_2$\$d_3$ -- d_2\$d_3$\$d_1$)
Reorders the first three elements on the stack such that the third element 'd_1' is brought to the top of the stack.

OVER (d_1\$d_2$ -- d_1\$d_2$\$d_1$)
Copies the second element on the stack to the top of the stack.

PICK (n -- copy of nth stack entry)
Copies the nth stack entry to the top of the stack.

Example:
Assuming the values on the stack are: 1 2 3 4, copy the bottom value (1) to the top of the stack.

```
3 PICK
```

The stack would then be: 1 2 3 4 1.

>R (n --)
(Push R)
Pushes the value 'n' onto the return stack for temporary storage. The value is removed from the computational stack.

R (-- n)
Copies the top value on the return stack to the main stack, leaves the return stack unaffected.

R> (-- n)
(Pop R)
Moves the top value on the return stack to the main stack.

WARNING
Any values placed on the return stack using '>R' must be removed from the return stack using 'R>' before the end of the ACTION{ }ACTION or 'colon-definition' clause.

Improper use of the return stack can cause system failure and/or program malfunction. See Appendix B for error recovery.

16
CREATING NEW COMMANDS

New commands are created via a 'colon-definition' using the following syntax:

```
: new_word
      existing commands ...
;
```

A new command is defined by entering a ':' (colon) followed by a space and the name of the command (maximum of 31 characters).

Following that, a sequence of previously defined commands separated by spaces can be entered. Finally, the definition is terminated with a ';' (semicolon).

Example:
Define the SEND_DACTPU command (which transmits information in the SDLC/SNA application program).

```
: SEND_DACTPU
      X" 2D00000004D96B80001202"  SEND_BTU
;
```

This command can be used in any action clause where the full expression is valid. That is, the following clauses are equivalent forms of the same action.

```
ACTION{
      X" 2D00000004D96B80001202"  SEND_BTU
}ACTION
```

and

```
ACTION{
      SEND_DACTPU
}ACTION
```

16.1 Pointers to Commands (Vectored Operation)

Many programming languages allow pointer variables which point to executable functions. This is also the case with ITL.

In ITL, a pointer to a function (or command) is called a DOER. This pointer must first be defined and then assigned to point at some existing command or sequence of ITL commands.

DOER 'command name'
Defines a pointer to a command.

Example:
```
DOER PRINT
```

NOTE
The pointer is not assigned to any sequence of ITL commands.

MAKE name
ITL commands ...
;AND
Assigns a sequence of ITL commands to a pointer.

WARNING
If a MAKE is defined within a colon definition, the terminator ';AND' must be used. Outside colon definitions and state definitions the terminator ';' must be used.

Example:
Define a new pointer command 'PRINT' which sends information to both the printer and screen when executed within a colon definition.

```
( s-- )
MAKE PRINT
      DUP
      COUNT W.TYPE  WCR      ( Output to screen )
      COUNT P.TYPE  PCR      ( Output to printer )
;AND
```

Execute the new command.
```
" This is a test" PRINT
```

Since vectored operation implies the ability to change the pointer, this can be done here as well.

Example:
Disable the printing as enabled above.
```
MAKE PRINT DROP ;       ( Where DROP deletes the input string from the stack )
```

16.2 Remote Processor Execution

FEF_DO_WORD (s\n --)
Where: s = the command sequence entered in string format
n = the application processor designator

Executes commands on another application processor without switching to that processor. The designator used for the processor depends on the interface configuration of the machine. Consult the following table to obtain the proper designator.

Example:
Execute an SABM command on AP #1.

```
" SABM" FEF_1 FEF_DO_WORD
```

Command strings containing quotation marks must be delimited with a CTRL Q.

Example:
Send a string on AP #1.

```
" d1Hello Thered1 SEND"  FEF_1  FEF_DO_WORD
```

	D-Chan	WAN	PRA	PRA/WAN	BRA	BRA/WAN	BRA/BRA	PRA/BRA/ WAN	WAN/WAN
AP #1		FEF_1	FEF_3	FEF_1	FEF_1	FEF_1	FEF_1	FEF_1	FEF_1
AP #2			FEF_4	FEF_3	FEF_2	FEF_2	FEF_2	FEF_2	FEF_2
AP #3	FEF_7			FEF_4	FEF_7	FEF_7	FEF_7	FEF_7	
AP #4							FEF_3	FEF_3	
AP #5							FEF_4	FEF_4	
AP #6							FEF_5		
Home	MFTH								

Table 16-1 Processor Identifier Values

17
TEST MANAGER

17.1 Developing Source Code

Source code can be prepared using the onboard visual editor or under a UNIX® environment and then transferred to the tester by using IDACOM's R-FILEX™ utility. PC's can also be used, with IDACOM's FILEX utility (XMODEM compatible), to transfer the files.

Workstation Environment Characteristics

Networked LAN
- Ethernet
- Token Ring

Windowed User Interface

Powerful Editing Capability

Hierarchical filesystem

Access control and protection

Source code development tools and revision control

Sun

Apollo

Vax

R-FILEX™ Protocol

IDACOM
Protocol Tester

XMODEM Protocol

PC

PC Environment Characteristics

Economical

Good expansion possibility

Flexible

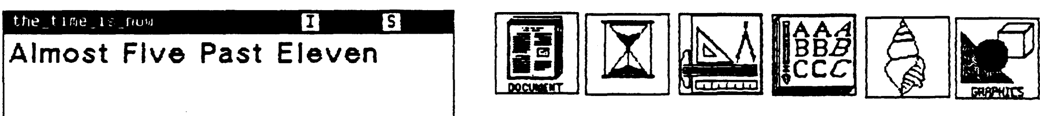
Accessible from common PC application software

Figure 17-1 Development System Environment

Test scripts can be saved as source code to be compiled each time the test script is loaded, or in binary format. The loading operation automatically determines whether compilation is required.

The maximum size of the compiled test script is approximately 50 Kbytes. In order to create large test scripts, multiple source files can be compiled sequentially.

The following figure illustrates a typical file transfer on a UNIX® system using R-FILEX™.



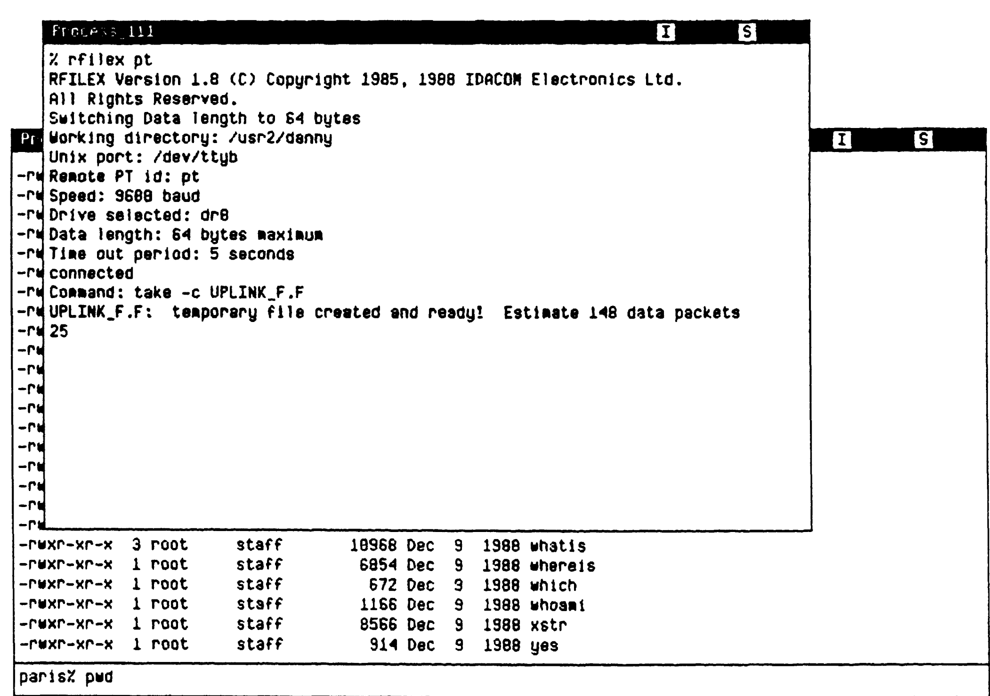


Figure 17–2 File Transfer Using R–FILEX

Source files are compiled into object code which is stored in memory. The size of memory available for the test script object code varies between application programs.

TM.RAM (--)

Prints the size of memory available for test programs.

Example:

```
TM.RAM
Test Manager DICTIONARY Used= 34  Free= 50334  Largest File= 98 blocks
```

17.2 Finite State Machine Concept

Test scripts are built as finite state machines. An ITL program contains one or more states each of which wait for one or more events to occur. Event recognition within a particular state leads to a set of actions which in turn could lead to the transition to another state.

The components of both the action and event clauses of the test program are built up using the ITL commands. These commands are divided into a *protocol specific* set of operations, where the power of the emulation or monitor software is utilized, and a *common command* set which can be used with all application software.

Both types of commands can be intermixed in a single program to detect incoming events and form the corresponding action.

The program remains in a state until the NEW_STATE command is executed and a different state is entered.

17.3 Symbol Definitions

The symbols used to represent SDL diagrams are defined by the CCITT Recommendations Z.101 through Z.104.

A **state** is a condition in which the action of the test manager is suspended awaiting the occurrence of an incoming event.

An **input** is an incoming event which is recognized by the test manager.

A **task** is any action following an event recognition that is neither a decision or an output.

An **output** is an action that results in the generation of an output to another entity.

A **decision** is an action which asks a question which results in the choice of following one of two paths.

The direction shown by the graphical description of inputs and outputs are valuable when representing a layered protocol state machine. Figure 17-3 shows IDACOM's SDL direction conventions.

An input received from a lower layer

An input received from an upper layer

An output going to an upper layer

An output going to a lower layer

Figure 17-3 IDACOM's SDL Direction Conventions

17.4 Introduction to ITL Test Script Structure

The high-level constructs used in building test scripts are illustrated in the following theoretical example. The SDL diagram for this example is shown in Figure 17-4.

Test Manager Theoretical Example

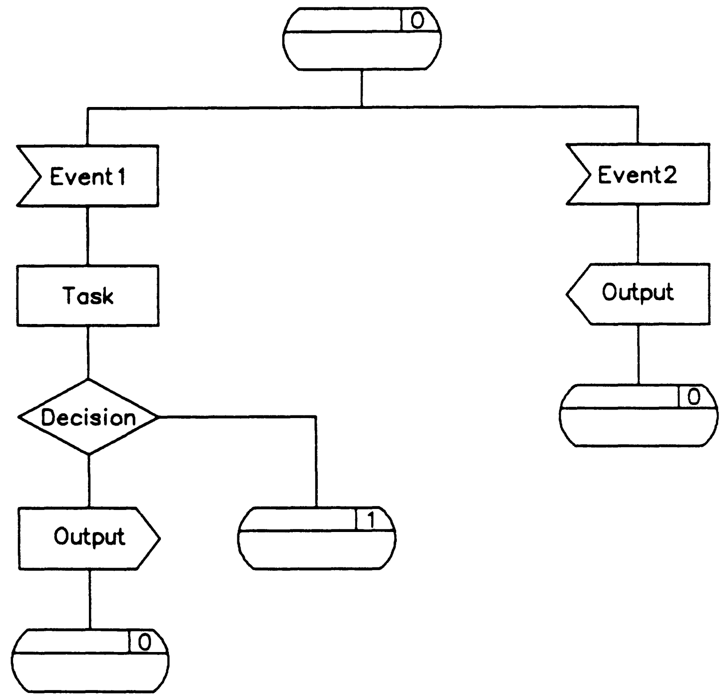


Figure 17-4 SDL Representation for One State

```
TCLR

0 STATE_INIT{
      actions
  }STATE_INIT

0 STATE{
      EVENT1
      ACTION{
          Task
          Decision
          IF
              1 NEW_STATE
          ELSE
              Output
          ENDIF
      }ACTION
```

```
    EVENT2
    ACTION{
        Output
    }ACTION
}STATE
```

Test Script Structural Components

The previous theoretical example utilizes some of the ITL structural components. These are described below. For additional structures, see Section 17.7.

TCLR (--)
Initializes the test manager and clears any existing test suites already in memory. The current state is set to 0. All test scripts should start with TCLR.

WARNING
When compiling multiple test scripts, TCLR must be used only in the first test script.

STATE_INIT{ }STATE_INIT (n --)
Brackets the execution sequence performed prior to entering a state. The initialization logic for a state is executed independently of how it was called.

This initialization procedure can be used for any state but is not compulsory. STATE_INIT{ must be preceded by the number of the state being initialized, eg. 0 STATE_INIT{. Valid values are 0 through 255.

If the ACTION{ }ACTION sequence does not result in a change of state, the STATE_INIT{ }STATE_INIT will not be re-executed.

STATE{ }STATE (n --)
Brackets a state definition. STATE{ must be preceded by the number of the state. Valid values are 0 through 255. State 0 must be defined within an ITL program. If not, the test manager will not run the script. If multiple states are defined with the same number in the test script, the test manager will use the latest definition.

Example:

```
1 STATE{
    ...             ( Event - action sequences )
  }STATE
```

ACTION{ }ACTION (--)
Brackets the set of tasks, decisions, and outputs which execute once the expected event is received by the test manager. There must be at least one action defined for each expected event. The action is executed when the flag is true (non-zero).

NEW_STATE (n --)

Executes the initialization logic of the specified state (providing STAT_INIT{ }STAT_INIT is defined) and establishes the state to be executed for the next event. Any remaining action code for the current state is then executed. It must be preceded with a valid state number and be inside the ACTION{ }ACTION brackets. This command is not mandatory if no state change is desired.

Example:

```
ACTION{
      1 NEW_STATE
}ACTION
```

TM_STOP (--)

Stops the execution of the test script. The test suite remains in memory and can be re-executed until another test script is loaded or TCLR is called.

17.5 Event Recognition

During test script execution, any event received by the test manager is evaluated to determine if the accompanying action should be performed. If the evaluation does not return a true value, the following events are evaluated in a sequential manner. Once an event evaluates true, the remaining events in that particular state are not examined.

In state 0 of the theoretical example (shown on page 17-5), EVENT1 is examined first. If the received event matches EVENT1, the actions defined are performed and the test manager does not look for EVENT2. If the received event does not match EVENT1, the test manager examines EVENT2. If the received event matches neither EVENT1 or EVENT2, the test manager performs no actions. No further processing takes place in that state until another event is received.

All received events are passed to the test manager via a FIFO queue. The test manager handles one event at a time on an equal priority basis.

To provide more flexibility, events can be logically OR'ed or AND'ed with other conditions or manipulated in other ways.

Example:

```
EVENT1 EVENT2 OR
ACTION{
    ...
}ACTION
```

Example:

```
EVENT1 EVENT2 AND
ACTION{
    ...
}ACTION
```

Events can be categorized as follows:

- Layer 1 Events
- Received Frames
- Timeout Detection
- Function Key Detection
- Interprocessor Mail Events
- Wildcard Events

Layer 1 Events

All applications running on WAN interfaces have a common set of commands. Layer 1 events for ISDN and SS#7 are specific to the respective protocols (see the appropriate Programmer's Manuals).

Individual or all interface leads can be enabled or disabled. Leads must be enabled for test manager detection.

ENABLE_LEAD (lead identifier --)
Enables the specified lead. Refer to Tables 17-1 and 17-2 for a list of supported leads.

Example:
Enable the request to send lead.

```
IRS ENABLE_LEAD
```

DISABLE_LEAD (lead identifier --)
Disables (default) the specified lead. Refer to Tables 17-1 and 17-2 for a list of supported leads.

Example:
Disable the clear to send lead.

```
ICS DISABLE_LEAD
```

ALL_LEADS (-- lead identifier)
Enables/disables leads supported on the currently selected WAN interface. ALL_LEADS must be used with ENABLE_LEAD or DISABLE_LEAD.

Example 1:
Enable leads for the current interface.

```
ALL_LEADS ENABLE_LEAD
```

Example 2:
Disable all leads for the current interface.

```
ALL_LEADS DISABLE_LEAD
```

DISABLE function key

NOTE
Only the leads in the following tables indicated with '', are enabled or disabled. Use ENABLE_LEAD or DISABLE_LEAD to enable/disable individual leads.*

TO_DCE Control Lead Identifiers				
Mnemonic	**V.28/RS-232C**	**V.35**	**V.36**	**V.11**
IRS	Request to Send *	Request to Send *	Request to Send *	
ITR	Data Terminal Ready *	Data Terminal Ready *	Data Terminal Ready *	
ISR	Data Signal Rate Select *		Data Signal Rate Select *	
ISS	Select Standby			
ILL	Local Loopback		Local Loopback *	
IC				Control *
IRL			Remote Loopback *	

Table 17-1 TO_DCE Control Lead Identifiers

TO_DTE Control Lead Identifiers				
Mnemonic	**V.28/RS-232C**	**V.35**	**V.36**	**V.11**
ICS	Clear to Send *	Clear to Send *	Clear to Send *	
IDM	Data Set Ready *	Data Set Ready *	Data Set Ready *	
IRR	Carrier Detect *	Carrier Detect *	Data Channel Received Line Signal Detector *	
ISQ	Signal Quality *			
IIC	Ring Indicate *	Ring Indicate *	Calling Indicator *	
ITM	Test Indicator		Test Indicator *	
II				Indicate *

Table 17-2 TO_DTE Control Lead Identifiers

Received Frames

The test manager provides recognition of protocol specific frames. Recognition of the frame can include anchored or unanchored comparisons of user-specified octets, frames with CRC errors, and/or aborted frames.

Any frames received by the monitor or emulation are decoded and the decoded information is stored in various communication variables (see the appropriate Programmer's Manual). The decoded information is used by the test manager to identify a particular event.

Timeout Detection

There are 128 user programmable timers available. Specific timers are reserved for use with the test manager in each program (see the appropriate Programmer's Manual). The remaining timers are used in the applications and should not be started or stopped in a test script.

TIMEOUT (-- f)
Returns true if any timer has expired.

Example:
In State 8, look for the expiration of any timer. The action is to display a trace statement.

```
8 STATE{
      TIMEOUT        ( Check for timeout of any timer )
      ACTION{
          T." A Timer has expired." TCR
      }ACTION
  }STATE
```

TIMER-NUMBER (-- a)
Contains the number of the expired timer. Valid values are 1 through 128.

Example:
In State 8, the test manager looks for the expiration of any timer. The action displays a trace statement listing the specific timer.

```
8 STATE{
      TIMEOUT                    ( Check for timeout of any timer )
      ACTION{
          T." Timer"
          TIMER-NUMBER @         ( Get timer # )
          T.                     ( Display timer # )
          T." has expired." TCR
      }ACTION
  }STATE
```

?TIMER (n -- f)

Returns true if the specified timer has expired. Valid timer numbers are those reserved in each application package.

Example:
In state 8, look for the expiration of timer 21. The action is to display a trace statement.

```
8 STATE{
      21 ?TIMER          ( Check for timeout of timer 21 )
      ACTION{
          T." Timer 21 has expired." TCR
      }ACTION
  }STATE
```

?WAKEUP (-- f)

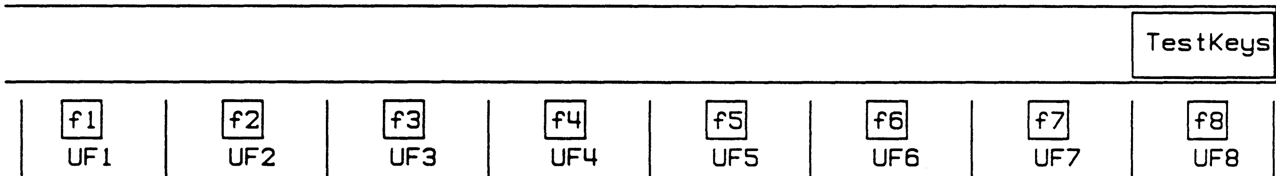
Returns true if the wakeup timer has expired. The wakeup timer can be used to initiate action sequences immediately upon the test manager starting. Timer 34 is started for 100 milliseconds when the test manager is started after a WAKEUP_ON command has been issued. The default is WAKEUP_OFF. Refer to the 'Starting a Test Script' section on page 17-39 for further description of these commands.

Example:
When the wakeup timer expires, prompt the user to press a function key and go to state 1.

```
0 STATE{
      ?WAKEUP        ( Check for timeout of wakeup timer )
      ACTION{
          T." To start the test, press UF1." TCR
          1 NEW_STATE
      }ACTION
  }STATE
```

Function Key Detection

Eight keyboard function keys are available in the test manager when running test scripts.



The following rules apply:

- Shifted control function keys and shifted function keys are never passed to the test manager.
- Control function keys are always passed to the test manager.
- Unshifted function keys are passed to the test manager only when the **TestKeys** topic is selected.

?KEY (n --)

Detects the specified function key when the **TestKeys** topic is selected.

Example:

```
UF1 ?KEY
```

Valid parameters for ?KEY are:

- UF1
- UF2
- UF3
- UF4
- UF5
- UF6
- UF7
- UF8

Example:

In State 0, look for function key 1. The action is to start timer 21 and go to state 1.

```
TCLR
0 STATE_INIT{
      " Press UF1 to start test." W.NOTICE
  }STATE_INIT

0 STATE{
      UF1 ?KEY              ( Start the test? )
      ACTION{               ( Yes )
          21 1 START_TIMER  ( Start timer 21 to expire in 1/10 second )
          1 NEW_STATE       ( Go to state 1 )
      }ACTION
  }STATE
```

Interprocessor Mail Events

Messages (mail) can be transmitted from a test script to another running test script on a different interface port.

This functionality is useful for the synchronization of test scripts running on different ports. Information or parameters can also be passed from one test script to another. The test manager recognizes the reception of this mail and the test script is responsible for decoding the mailed message.

?MAIL (-- f)

When mail is received from another processor, returns true and executes the ACTION{ }ACTION sequence.

Examples of interprocessor mail include the starting and stopping of X.25 B-Channel data under the control of the D-Channel in ISDN. Another example is performing transit delay measurements across a protocol converter.

Parameters within a mailed message, can be extracted with the EXTRACT_FTH_DATA command. Situations requiring only signalling (i.e. synchronization) do not require the use of this command.

For an explanation of the CPU1_MAIL command, refer to the 'Mailing to Another Processor' section on page 17-31.

EXTRACT_FTH_DATA (-- d_n\...\d_1\n)

Unpacks the received message buffer and returns a variable number of items where the top number on the stack indicates the number of 32 bit values in the mail message. These values could include constants, variables, or addresses. The maximum number of items is 12.

Example:
Print the values in any received messages.

```
TCLR

0 STATE{
      ?MAIL                    ( Check event field for mail )
      ACTION{
          EXTRACT_FTH_DATA
          DUP 1 >               ( Any remaining? )
          IF                    ( Yes - print them )
              1 - 0
              DO
                  T.            ( Print all values on the stack )
              LOOP
              TCR
          ELSE
              DROP              ( Drop count )
          ENDIF
      }ACTION
  }STATE
```

Wildcard Events

Wildcard events can be used for a "don't care" situation when the test script is not looking for a specific event.

OTHER_EVENT (-- f)
Returns true for any incoming event.

NOTE
OTHER_EVENT must be the last event listed within the action sequence.

The actual type of received event is in the EVENT-TYPE variable. Possible values for EVENT-TYPE vary according to specific application and protocol (see the appropriate Programmer's Manual).

Within an action which uses OTHER_EVENT, the IF statement can be used to perform actions based on the actual event type.

Example:
This example comes from an ISDN test script.

```
OTHER_EVENT
ACTION{
      EVENT-TYPE @ TIME-OUT#
      IF
          T." Timer expired" TCR
      ENDIF
}ACTION
```

Example:
Look for the *UF1* function key (which stops the test) and then for any other event but an incoming frame. The following example is taken from an ISDN test script.

```
TCLR

5 STATE{
      UF1 ?KEY
      ACTION{
          T." Test has stopped." TCR               ( Display trace )
          TM_STOP                                  ( Stop test manager )
      }ACTION
```

```
        OTHER_EVENT
        ACTION{
            EVENT-TYPE @ DUP                        ( Fetch the event )
            FRAME# =                                ( Is it a frame? )
            IF                                      ( Yes, clean up the stack )
                DROP
            ELSE                                    ( No, display trace )
                T." Event is"
                DOCASE
                    CASE TIME-OUT#
                    {
                        T." a timeout of timer"     ( Display timeout )
                        TIMER-NUMBER @              ( Fetch timer number )
                        T.                          ( Display timer number )
                    }
                    CASE FUNCTION-KEY#
                    {
                        T." a function key"   ( Display event is function key )
                    }
                    CASE COMMAND_IND
                    {
                        T." a mail"           ( Display event is a mail )
                    }
                    CASE DUP
                    {
                        T." undefined"        ( Display event is undefined )
                    }
                ENDCASE
                TCR
            ENDIF
        }ACTION
    }STATE
```

17.6 General Actions

Actions can be executed when an event is recognized. The following actions can be performed by either the monitor or the emulation:

- Starting or stopping data display, capture to RAM, or recording to disk
- Alarm generation, eg. beeping or displaying highlighted messages
- Trace reporting in the Data Window
- Reporting in the User Window
- Output to printer or remote ports
- User input
- User output
- Starting or stopping timers
- Manipulating counters
- Mailing to another processor

The following actions can only be performed by the emulation:

- Layer 1 actions, i.e. activating/deactivating the S/T Bus for ISDN or turning on and off control leads on a WAN Interface
- Generating protocol specific action (transmitting frames or messages)

Display, Capture, or Record

NOTE
The following commands are not common to all applications.

REP_ON (--)
Turns on data display.

REP_OFF (--)
Turns off data display.

CAPT_ON (--)
Saves live data in capture RAM.

CAPT_OFF (--)
Live data is not saved in capture RAM.

=TITLE (filename--)
Specifies the name of the file to be opened for disk recording or disk playback.

Example:
Obtain playback data from disk

```
HALT               ( Place the monitor in playback  mode )
FROM_DISK          ( Identify a disk file as data source )
" X25DAT" =TITLE   ( Create a title for next data file to be opened )
PLAYBACK           ( Playback data )
```

RECORD (--)
Opens a data recording file. Default values are as chosen in the Recording Menu. When used in the Command Window, the filename can be specified as part of the command.

Example 1:

```
RECORD DATA1
```

Example 2:
Specify the 'DATA1' filename to open for recording.

```
" DATA1" =TITLE   ( Change title of file to DATA1 )
RECORD            ( Open the recording )
```

Example 3:
Specify drive DR0 and the 'DATA1' filename to open for recording.

```
" DR0:DATA1" =TITLE ( File on drive 0 named DATA1 )
RECORD              ( Open the recording )
```

NOTE
When RECORD is used in a test script, the filename must be specified with =TITLE.

WARNING
Because of the relatively long time required to open a disk file (especially on a floppy drive), the RECORD command should not be used within time critical portions of a test script.

DISK_OFF (--)
Live data is not recorded to disk. The current disk recording is closed. This must be done prior to opening a different file with RECORD. For multi-processor recording (same file open on more than one processor), data from the current processor is no longer recorded.

The file is closed when DISK_OFF is issued on the last processor recording to disk.

PLAYBACK (--)
Opens a data recording file for playback. When used in the Command Window, the filename can be specified as part of the command.

Example:
```
PLAYBACK DATA1
```

NOTE
When PLAYBACK is used in a test script, the filename must be specified with =TITLE.

STATE_ON (--)
Generates a report line for every change of state in the automatic protocol state machine of the emulation or in the test manager.

STATE_OFF (--)
Disables the reporting of protocol and test manager state changes.

Example:
When Timer 21 expires, turn on the display, capture data to RAM, and open a disk recording on drive 0 with the title DATA2. When Timer 22 expires, turn off the display, cease capture of data, close the disk recording, and enter a new state.

```
4 STATE[
      21 ?TIMER              ( Check event field for timeout of timer 21 )
      ACTION[
          REP_ON             ( Turn on the data display )
          CAPT_ON            ( Start capturing data )
          " DR0:DATA2" =TITLE  ( Select drive 0 with titlename DATA2 )
          RECORD             ( Open recording )
      ]ACTION
```

```
    22 ?TIMER               ( Check event field for timeout of timer 22 )
    ACTION{
        REP_OFF             ( Turn off the data display )
        CAPT_OFF            ( Stop capturing data )
        DISK_OFF            ( Close disk recording )
        5 NEW_STATE         ( Go to State 5 )
    }ACTION
}STATE
```

Audible Alarms

There are three commands available to alter audible alarms.

BEEP (--)
Generates a single beep.

BEEP_ON (--)
Generates a continuous sequence of beeps.

BEEP_OFF (--)
Stops the continuous sequence of beeps.

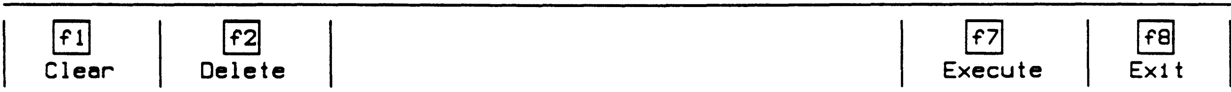
Input

Test scripts can accept input from the user in the following ways.

- text
- numerical
- function key

For text and numerical input, the PROMPT command displays a user-defined message and waits for input. Full type checking is provided for all inputs.

PROMPT" string" action **END_PROMPT**
Defines a keyboard input prompt. The prompt function keys appear on the screen. There must be one space after the first quotation mark before the text.



Example 1:
Create a prompt that asks the user to enter a number from 1 to 100 and store this value in the COUNTER variable. The text in the prompt must be entered between the quotation marks in the PROMPT" string".

```
PROMPT" Enter number of repeats (1-100):"   ( Display the prompt )
      prompt                                ( Get the keyboard entry )
      10 STR>#                              ( Convert it to decimal )
```

The keyboard input is temporarily stored in the 'prompt' variable as an ASCII string with the count of the number of characters entered stored in the first byte.

WARNING
'prompt' can be called only once in each PROMPT" string" action END_PROMPT sequence.

Translate the keyboard entry to a number in the decimal base using the STR>># command (pronounced string to number). This command must be preceded by 'prompt' and the number 10 which represents decimal base. If the conversion is successful, STR>># returns the converted value and a success flag. If the conversion fails, a false flag is returned.

The entire code for this example is:

```
ACTION{
      PROMPT" Enter number of repeats (1-100):"   ( Display the prompt )
          prompt                                  ( Get the keyboard entry )
          10 STR>#                                ( Convert it to decimal )
          IF                                      ( Conversion is successful )
              DUP 1 100 BETWEEN?                  ( Check the range )
              IF
                  COUNTER !                       ( Store it in COUNTER )
                  1 NEW_STATE
              ELSE                                ( Not between 1 and 100 )
                  DROP
                  " Invalid entry"
                  W.ERROR
              ENDIF
          ELSE                                    ( Invalid entry - non-numeric
                                                characters )
              " Invalid entry"
              W.ERROR
          ENDIF
      END_PROMPT
}ACTION
```

WARNING
The PROMPT" string" action END_PROMPT sequence must be the last command called in an ACTION{ }ACTION sequence of the test manager. Otherwise, any following commands are executed without waiting for the keyboard input in answer to the PROMPT.

Example 2:
The following test script is an example from X.25. These are separate prompts.

Decide how many channels to configure and ask for the LCN number, called and calling addresses. Assign these values starting at CH1.

NOTE
The last of the commands including the NEW_STATE command for each ACTION{ }ACTION sequence is contained in the PROMPT" action.

```
TCLR
WAKEUP_ON
#IFNOTDEF #LCNS       ( Define the variable and colon definition once only )
      0 VARIABLE #LCNS        ( Counter used for assigning LCN- 1-64 valid )
                              ( Prompt user for number of LCN's and set it )
      : GET_LCNS_REQ          ( -- )
          PROMPT" ENTER THE # OF LCN'S REQUIRED (1-64):"  ( Display prompt )
          prompt
          10 STR>#                          ( Convert to decimal )
          IF                                ( Valid decimal value )
              DUP 1 64 BETWEEN?             ( Between 1 & 64? )
              IF                            ( Yes )
                  #LCNS !
                  21 1 START_TIMER          ( Store number )
                  1 NEW_STATE               ( Go to new state )
              ELSE                          ( Not between 1 and 64 )
                  DROP                      ( Clean up stack )
                  " Invalid entry" W.ERROR  ( Generate error message )
              ENDIF
          ELSE                              ( Not a decimal value )
              " Invalid entry" W.ERROR      ( Generate error message )
          ENDIF
          END_PROMPT
      ;
                             ( Prompt user for calling address and set it )
      : GET_CALLING ( -- )                  ( LCNCALLING is a 16 byte string )
          PROMPT" ENTER THE CALLING ADDRESS: "              ( Display prompt )
          prompt LCNCALLING 16 CMOVE        ( Move 1st 16 characters to
                                              LCNCALLING )
          LCNCALLING C@ 15 MIN LCNCALLING C! ( Put smaller of # of digits
                                              entered or #15 as 1st byte of
                                              LCNCALLING )
          21 1 START_TIMER
          3 NEW_STATE                       ( Go to new state )
          END_PROMPT
      ;                     ( Prompt user for LCN and set it )
```

```
      : GET_CALLED ( -- )                     ( LCN called is a 16 byte string )
          PROMPT" ENTER THE CALLED ADDRESS: "( Prompt user )
          prompt LCNCALLED 16 CMOVE           ( Move first 16 characters to
                                                LCNCALLED )
          LCNCALLED C@ 15 MIN LCNCALLED C!    ( Put smaller # of digits entered
                                                and 15 as first byte of
                                                LCNCALLED )
          21 1 START_TIMER
          4 NEW_STATE                         ( Go to new state )
          END_PROMPT
      ;                                       ( Prompt user for LCN and set it )
      : SET_LCN
          PROMPT" ENTER THE LCN NUMBER ( 1-4095 ): "       ( Prompt user )
          10 STR>#                            ( Convert to decimal )
          IF                                  ( Valid decimal value )
              DUP 1 4095 BETWEEN?             ( Between 1 and 4095? )
              IF                              ( Yes )
                  COUNTER1  @ CH =LCN         ( Set of channel and LCN )
                  21 START_TIMER 2 NEW_STATE  ( Go to new state )
              ELSE                            ( Not between 1 and 4095 )
                  " Invalid LCN No." W. ERROR DROP
                              ( Clear up stack and create error message )
              ENDIF
          ELSE                                ( Not a decimal value )
              " Invalid LCN NO." W. ERROR     ( Create error message )
          ENDIF                               ( Not a decimal value )
          END_PROMPT
      ;
#ENDIF                                  ( End conditional compile )

0 STATE_INIT{
      0 COUNTER1 !                      ( Initialize counter )
  }STATE_INIT

0 STATE{
      ?WAKEUP                           ( Start test script automatically )
      ACTION{
          GET_LCNS_REQ                  ( Get # of channels )
      }ACTION
  }STATE

1 STATE{
      21 ?TIMER
      ACTION{
          1 COUNTER1 +!                 ( Increment counter )
          SET_LCN                       ( Set channel and LCN )
      }ACTION
  }STATE
```

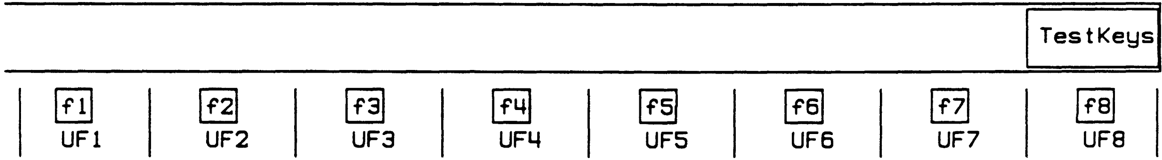
```
2 STATE{
      21?TIMER
      ACTION{
          GET_CALLING                    ( Set calling address for this channel )
      }ACTION
  }STATE

3 STATE{
      21 ?TIMER
      ACTION{
          GET_CALLED                     ( Set called address for this channel )
      }ACTION
  }STATE

4 STATE{
      21 ?TIMER
      ACTION{
          COUNTER1 @ #LCNS @ = 0=        ( Are all LCN's assigned? )
          IF
              21 1 START_TIMER
          ENDIF
      }ACTION
  }STATE
```

User–Defined Function Keys

The labels on the following default function keys can be changed in a test script or by typing in commands. The area provided is 80 columns; however, nine of these columns are taken up by the function key dividing lines.



The following commands are used to change the default function keys.

LABEL_KEY (string\n --)

Displays a string of up to 10 characters on the specified key. Valid key values are 1 through 8.

Example:
Change the display of function key 1 to 'Continue'.

```
" Continue" 1 LABEL_KEY
```

NOTE

Not all keys can be 10 characters wide due to the 9 function key separators on the display. Once the total space available for keys has been used, LABEL_KEY truncates the latest key labeled.

HILITE_LABEL (yes|no\fk# --)
Turns highlighting on/off on the specified function key under the **TestKeys** topic.

Example:
```
YES 1 HILITE_LABEL
NO 2 HILITE_LABEL
```

CLEAR_KEY (n --)
Clears the text for the specified function key. Valid key values are 1 through 8.

Example:
Clear all text from function key 6.
```
6 CLEAR_KEY
```

It is not necessary to label a key to detect it within a test program.

CLEAR_KEYS (--)
Removes the text from all eight function keys.

DEFAULT_KEYS (--)
Restores the labels of all test keys to their default values. The default labels are "UF1", "UF2", ... "UF8".

SET_CURR_TOPIC (string --)
Moves the topic box to the specified topic.

NOTE
To move to a topic, the spelling must match the display on the topic bar.

Example:
Move the topic box to the **TestKeys** topic.

```
0 STATE_INIT{
      " TestKeys" SET_CURR_TOPIC
  }STATE_INIT
```

Direct Key Actions

Actions can be assigned to keys that are executed, without the test manager looking for a function key event.

F1_ACTION

Defines the action of UF1 using the MAKE ... ;AND construct and executes the action independent of what state the program is in. When assigning a new action to a key, it is advisable to relabel the key.

NOTE

No stack comments are shown, as these are dependent on user definition.

NOTE

TestKeys UF2-UF8 can be set using the commands F2_ACTION through F8_ACTION, respectively.

Example:
Define the action for UF1 as BEEP.

```
MAKE F1_ACTION BEEP  ;AND
```

The previous example is equivalent to:

```
UF1 ?KEY
ACTION{
    BEEP
}ACTION
```

Output – Notices and Errors

Notices or error messages can be created using the W.NOTICE or W.NOTICES commands.

Notices are displayed in the Notice Window (Row 17 of the screen) and must not be longer than 79 characters. Press any key to clear the message. When a new notice is displayed, the previous one is overwritten.

W.NOTICE (string --)

Displays a single string in the Notice Window.

Example:

```
" Invalid number" W.NOTICE
```

W.NOTICES ($string_n$\...\$string_1$\n --) or ($string_n$\...\#arg\number\n --)

Where: n = the total number of strings and numbers including the #arg to display

Displays a string and/or number in the Notice Window. If a number is included, it must be preceded by the #arg command.

Example:

```
" Invalid number = " #arg 7 3 W.NOTICES
```

```
Invalid number = 7
```

Example:

Print the values of two counters together with descriptive text. The six arguments are identified by the numbers below.

```
" LCN numbers" #arg COUNTER1 @ " WINDOW=" #arg COUNTER2 @  6 W.NOTICES
        ↑        ↑     ↑           ↑         ↑     ↑
        1        2     3           4         5     6
```

Assuming COUNTER1 and COUNTER2 contain 1 and 2 respectively, the resulting output is:

```
LCN numbers 1 WINDOW=2.
```

NOTE

Arithmetic expressions can be used following an #arg, however, the result of the computation counts as one argument (as illustrated above).

Error messages are displayed in the Error Window in the center of the screen. The message must be no more than 74 characters.

Use the W.ERROR and W.ERRORS commands to display error messages.

W.ERROR (string --)

Displays a single string in the Error Window.

Example:

```
" Invalid number" W.ERROR
```

W.ERRORS ($string_n$\...\$string_1$\n --) or ($string_n$\...\#arg\number\n --)

Displays a string and/or numbers in the Error Window. If a number is included, it must be preceded by the #arg command.

Example:

```
" Invalid number = " #arg 7 3 W.ERRORS
```

```
Error Message
Invalid number = 7

Press any key to clear this message and continue.
```

Starting or Stopping Timers

Interval timers can be started or stopped within the action field. There are 128 user programmable timers available of which only some can be used within user test script programs. Others are used by the emulation program for the protocol implementation. Protocol specific timers are described in the Programmer's Manual for each application.

START_TIMER (timer#\time --)

Starts the specified timer which, if not stopped or restarted, provides a timeout indication. Valid values for the timeout, in tenths of seconds, are 1 through 2147483647 ($7FFFFFFF_{16}$). The resolution is approximately 30 milliseconds.

Example:

Start timer 21. The timeout indication occurs in one minute.

```
21 600 START_TIMER
```

STOP_TIMER (timer# --)

Stops the specified timer.

Example:

Stop timer 21.

```
21 STOP_TIMER
```

In addition to the interval timers, there are 256 lapse timers which can be started and read at any time. Lapse timers are similar to a stop watch and can be used to measure the duration of events.

START_LAPSE_TIMER (timer# --)

Starts the specified timer. Valid values are 0 through 255. If an invalid value is specified, a notice appears stating that the lapse timer number is out of range.

MINUTES_ELAPSED (timer# -- minutes)

Returns the number of minutes elapsed since the timer was started. Valid values for the timer number are 0 through 255.

SECONDS_ELAPSED (timer# -- seconds)

Returns the number of seconds elapsed since the timer was started. Valid values for the timer number are 0 through 255.

MILLISECONDS_ELAPSED (timer# -- milliseconds)

Returns the number of milliseconds elapsed since the timer was started. Valid values for the timer number are 0 through 255.

Example:

This example comes from an ISDN test script. In State 0, when an SABM frame is received, the test manager start lapse timer 2 and timer 21 and go to State 1. In State 1, look first for a DISC frame, then timer 21. If the DISC frame is received prior to timeout of timer 21, the test script determines the interval between the SABM and the DISC and reports it.

```
0 STATE{
      R#SABM ?RX_FRAME
      ACTION{
          2 START_LAPSE_TIMER    ( Start lapse timer )
          101 6000 START_TIMER   ( Start timer 21 for 10 minutes )
          1 NEW_STATE            ( Go to state 1 )
      }ACTION
  }STATE

1 STATE{                         ( Measure and print how long the DISC took )
      R#DISC ?RX_FRAME
      ACTION{
          T." Interval between SABM and DISC ="
          2 MILLISECONDS_ELAPSED T.
          T." milliseconds"
          TCR
      }ACTION

      101 ?TIMER                  ( The DISC never came - test fails )
      ACTION{
          T." Disc not received in 10 minutes."
          TCR
          TM_STOP
      }ACTION
  }STATE
```

Manipulating Counters

There are 32 (or more) counters supplied with each protocol application software: COUNTER1, COUNTER2, . . ., COUNTER32.

These counters can be read, written to, incremented, decremented, and used in decision making. A review of the mechanism for performing these operations is provided below using COUNTER5 as an example.

Further explanation on using and manipulating counters may be found in Sections 3, 4, and 7.

@ (a -- d)

(fetch)

Fetches a 32 bit value 'd' (read) from address 'a'.

Example:

Assume the variable COUNTER5 contains the hex value 08040201. Place 08040201 on the stack.

```
COUNTER5 @
```

! (d\a --)

(store)

Stores a 32 bit value at the specified address.

Example:

Store 127 (7F hex) in COUNTER5.

```
127 COUNTER5 !
```

+! (d\a --)

(plus-store)

Increments/decrements a 32 bit value 'd' to the contents of address 'a'.

Example:

Increment the contents of COUNTER5 by one. Assuming it initially contained 08040201, COUNTER5 would now contain the hex value of 08040202.

```
1 COUNTER5 +!
```

Decrement the contents of COUNTER5 by five. COUNTER5 would now contain the hex value of 080F01 FC.

```
-5 COUNTER5 +!
```

Using Counters in Decision Making

= ($d_1 \backslash d_2$ -- f)

(equality)

Where: d_1, d_2 = 32 bit numeric value to compare
f = result of comparison

Returns true if 'd_1', and 'd_2' are equal.

Example:
Remain in a state until an event has occurred 10 times. The underlying presumption in the following example, is that the COUNTER variable was initialized to the value of zero before entering this state.

```
ACTION{
    1 COUNTER +!         ( Increment the counter )
    COUNTER @ 10 =       ( Counter contents = 10 ? )
    IF                   ( Yes )
        5 NEW_STATE      ( Go to state 5 )
    ENDIF
}ACTION
```

> ($d_1 \backslash d_2$ -- f)

(greater than)

Where: d_1, d_2 = values to compare
f = result of comparison

Returns true if 'd_1' is greater than 'd_2'.

Example:
Remain in the same state until an event occurs more than 9 times. The underlying presumption is that the COUNTER variable was initialized to the value of zero before entering the state.

```
ACTION{
      COUNTER @ 9 >          ( Is COUNTER bigger than COUNTER1 ? )
      IF                     ( Yes )
          2 NEW_STATE        ( Go to state 2 )
      ELSE
          1 COUNTER +!       ( Increment COUNTER )
      ENDIF
}ACTION
```

< ($d_1 \backslash d_2$ -- f)

(less than)

Where: d_1, d_2 = values to compare
f = result of comparison

Returns true if 'd_1' is less than 'd_2'.

Example:
Remain in the same state for at least 10 occurrences of an event. The underlying presumption is that the COUNTER variable was initialized to the value of zero before entering this state.

```
ACTION{
      COUNTER @ 11 <        ( Counter contents < 11 ? )
      IF                    ( Yes )
          1 COUNTER +!      ( Increment COUNTER by one )
      ELSE
          4 NEW_STATE       ( Go to state 4 )
      ENDIF
}ACTION
```

NOTE
The previous three examples result in the same actions being taken.

AND ($d_1 \backslash d_2$ -- d_3)
(logical AND)
Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise logical AND on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:
Assume COUNTER1 contains the value 5 (binary 101) and COUNTER2 contains the value 3 (binary 011). The result of the AND operation leaves the value 1 (binary 001) on the stack.
```
COUNTER1 @ COUNTER2 @ AND
```

OR ($d_1 \backslash d_2$ -- d_3)
(logical OR)
Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise logical OR on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:
Assume COUNTER1 contains the value 5 (binary 101) and COUNTER2 contains the value 3 (binary 011). The result of the OR operation leaves the value 7 (binary 111) on the stack.
```
COUNTER1 @ COUNTER2 @ OR
```

XOR ($value_1 \backslash value_2$ -- $value_3$)
(exclusive OR)
Where: d_1, d_2, d_3 = 32 bit integers

Performs a bitwise exclusive OR on 'd_1' and 'd_2' and leaves the result 'd_3' on the top of the stack.

Example:
Assume COUNTER1 contains the value 5 (binary 101) and COUNTER2 contains the value 3 (binary 011). The result of the XOR operation leaves the value 6 (binary 110) on the stack.
```
COUNTER1 @ COUNTER2 @ XOR
```

Mailing to Another Processor

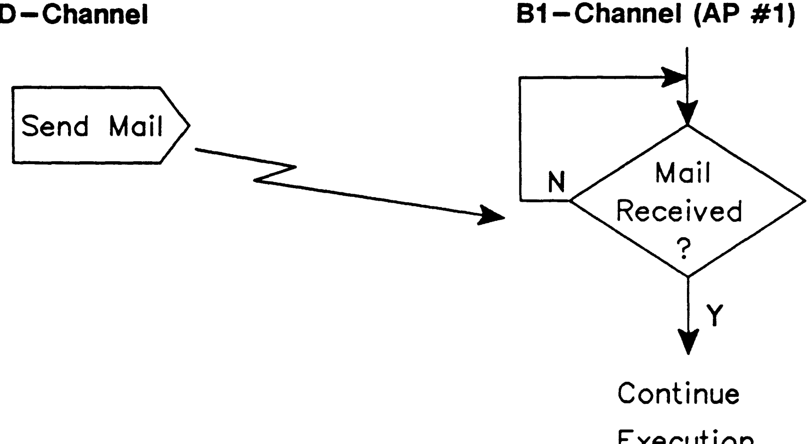
Mail commands allow communication between test scripts that are running on different application processors in the same tester. This functionality is useful for the synchronization of test scripts running on different ports, and to pass information or parameters from one script to another.

Examples of inter-processor mail include the starting and stopping of X.25 B-Channel data under the control of the D-Channel in ISDN. Another example is performing transit delay measurements across a protocol converter.

Up to 12 numerical items can be sent to another processor. The last parameter listed must be a count indicating how many parameters to mail.

NOTE
These commands vary according to machine configuration and are not applicable to single port WAN units.

Example:
To provide synchronization between two running test scripts, the flow chart below would illustrate the necessary technique.



D-Channel Test Program

```
0 STATE[
     UF1 ?KEY
     ACTION[
          1 1 CPU1_MAIL
     ]ACTION
  ]STATE
```

B1-Channel Test Program

```
0 STATE[
     ?MAIL
     ACTION[
          1 NEW_STATE
     ]ACTION
  ]STATE
```

CPU1_MAIL (d_1\...\d_2\n --)
Transmits a list of parameters (maximum 12) plus the count of the number of these parameters to FEF_1. When the application processor receives this mail, the user program must check for this event via ?MAIL and unpack the message by calling EXTRACT_FTH_DATA. See Table 16-1 for the relationship between application processor and the FEF number.

Example:
Transmit the mail commands which are shown in the example under the EXTRACT_FTH_DATA command to AP #1.

```
TCLR
0 STATE{
      ?WAKEUP
      ACTION{
          2 4 6 3 CPU1_MAIL
          21 1 START_TIMER
      }ACTION

      ( The test script on AP #1 will display the following in a trace statement:
        'The mail parameters are 6 4 2' )
  }STATE
```

Table 17-3 lists other mail commands which transmit messages to different application processors. The input parameters are identical to CPU1_MAIL as previously described.

Command	Destination
CPU1_MAIL	FEF_1
CPU2_MAIL	FEF_2
CPU3_MAIL	FEF_3
CPU4_MAIL	FEF_4
CPUD_MAIL	FEF_7
CPUDB_MAIL	FEF_5

Table 17–3 Mail Commands

MAIL_CMD (d_1\...\ d_2\n --)
Transmits a message to the *partner* of the current application processor. Partner processors are defined in Table 17-4. This command is useful for writing a test program that operates independent of the processor it is running on. For example, MAIL_CMD on port 1 of a WAN/WAN machine mails to port 2, while the same command executed on port 2 does the opposite.

Processor Partners	
FEF_1	FEF_2
FEF_3	FEF_4
FEF_5	FEF_7

Table 17–4 MAIL_CMD Partners

See the 'Interprocessor Mail Events' section on page 17-12 for information concerning the detection of mail events and their subsequent interpretation.

Protocol Specific Actions

Each IDACOM application software program provides protocol dependent commands which can be used within action sequences. These commands include turning on and off control leads, activating the S/T Bus (BRA), plus the construction and transmission of higher layer protocol information. For more details about protocol dependent commands, consult the appropriate Programmer's Manual.

17.7 Additional ITL Structures

SEQ{ }SEQ (number --)

Brackets a definition of tasks and outputs which execute as part of the state machine action, and declares them to be a sub-routine. SEQ{ expects a single integer which is the sequence number. Up to 256 sequences are supported. Valid values are 0 through 255. The SEQ{ }SEQ partners are extremely useful when more than one action sequence calls the same tasks and outputs. The SEQ{ }SEQ definition is defined outside the STATE{ }STATE definition and then called using the RUN_SEQ command.

This is an alternate mechanism to generate colon definitions. This mechanism causes the equivalent of a colon definition (now accessed via a numeric identifier) to be compiled into the test script dictionary rather than the user dictionary.

RUN_SEQ (number --)

Executes a specified set of tasks defined in a SEQ{ }SEQ definition. It is called inside an ACTION{ }ACTION definition and must be preceded with a defined sequence number.

Example:
The following program operates in conjunction with either the ISDN Monitor or Emulation software to collect statistics on types of received frames within a certain interval.

In state 0. set the topic bar to the **TestKeys** topic, label three of these keys, initialize counters, expose the Data Window, and call State 1. The test manager now remains in State 1 until the test script is stopped. The user can choose, via the three test keys, to view the Data Window or the User Window without moving the topic bar to the **Background** topic. The counters can be re-initialized with the third function key.

```
TCLR
WAKEUP_ON

0 SEQ{
     0 COUNTER1 !              ( Initialize counters )
     0 COUNTER2 !
     0 COUNTER3 !
     0 COUNTER4 !
     0 COUNTER5 !
     0 COUNTER6 !
     0 COUNTER7 !
     0 COUNTER8 !
     0 COUNTER9 !
     0 COUNTER10 !
     0 COUNTER11 !
     0 COUNTER12 !
  }SEQ

1 SEQ{
     POP_USER                  ( Open the User Window )
     CLEAR_TEXT WHI_FG PAINT   ( Clear screen text and color )
     0  20 THERE  W." Statistics Display after 1 minute"
     2  30 THERE  W." I      =" COUNTER1 @ W.
     3  30 THERE  W." RR     =" COUNTER2 @ W.
     4  30 THERE  W." RNR    =" COUNTER3 @ W.
     5  30 THERE  W." REJ    =" COUNTER4 @ W.
     6  30 THERE  W." SABM   =" COUNTER5 @ W.
     7  30 THERE  W." SABME  =" COUNTER6 @ W.
     8  30 THERE  W." DISC   =" COUNTER7 @ W.
     9  30 THERE  W." UA     =" COUNTER8 @ W.
     10 30 THERE  W." DM     =" COUNTER9 @ W.
     11 30 THERE  W." FRMR   =" COUNTER10 @ W.
     12 30 THERE  W." UI     =" COUNTER11 @ W.
     13 30 THERE  W." XID    =" COUNTER12 @ W.
     CLOSE_WINDOW
  }SEQ

0 STATE{
     ?WAKEUP
     ACTION{
         0 RUN_SEQ
         " TestKeys" SET_CURR_TOPIC    ( Zero counters )
         " SHOW_DATA" 1 LABEL_KEY      ( Label keys 1, 2, and 3 )
         " SHOW_STATS" 2 LABEL_KEY
         " RESTART" 3 LABEL_KEY
         SHOW_DATA                     ( Show the Data Window )
         1 NEW_STATE
      }ACTION
  }STATE
```

```
1 STATE{
     R#I ?RX_FRAME                  ( Check event field for I Frame )
     ACTION{
          1 COUNTER1 +!              ( Increment counter )
          OPEN_USER                  ( Open User Window )
          2 38 THERE                 ( Position text at row 2, column 38 )
          COUNTER1 @ W.              ( Display number of I frames received )
          CLOSE_WINDOW
     }ACTION

     R#RR ?RX_FRAME                 ( Check event field for RR frame )
     ACTION{
          1 COUNTER2 +!
          OPEN_USER
          3 38 THERE
          COUNTER2 @ W.              ( Display number of RR frames received )
          CLOSE_WINDOW
     }ACTION

     R#RNR ?RX_FRAME                ( Check event field for RNR frame )
     ACTION{
          1 COUNTER3 +!
          OPEN_USER
          4 38 THERE
          COUNTER3 @ W.              ( Display number of RNR frames received )
          CLOSE_WINDOW
     }ACTION

     R#REJ ?RX_FRAME                ( Check event field for REJ frame )
     ACTION{
          1 COUNTER4 +!
          OPEN_USER
          5 38 THERE
          COUNTER4 @ W.              ( Display number of REJ frames received )
          CLOSE_WINDOW
     }ACTION

     R#SABM ?RX_FRAME               ( Check event field for SABM frame )
     ACTION{
          1 COUNTER5 +!
          OPEN_USER
          6 38 THERE
          COUNTER5 @ W.              ( Display number of SABM frames received )
          CLOSE_WINDOW
     }ACTION
```

```
R#SABME ?RX_FRAME           ( Check event field for SABME frame )
ACTION{
    1 COUNTER6 +!
    OPEN_USER
    7 38 THERE
    COUNTER6 @ W.           ( Display number of SABME frames received )
    CLOSE_WINDOW
}ACTION

R#DISC ?RX_FRAME            ( Check event field for DISC frame )
ACTION{
    1 COUNTER7 +!
    OPEN_USER
    8 38 THERE
    COUNTER7 @ W.           ( Display number of DISC frames received )
    CLOSE_WINDOW
}ACTION

R#UA ?RX_FRAME              ( Check event field for UA frame )
ACTION{
    1 COUNTER8 +!
    OPEN_USER
    9 38 THERE
    COUNTER8 @ W.           ( Display number of UA frames received )
    CLOSE_WINDOW
}ACTION

R#DM ?RX_FRAME              ( Check event field for DM frame )
ACTION{
    1 COUNTER9 +!
    OPEN_USER
    10 38 THERE
    COUNTER9 @ W.           ( Display number of DM frames received )
    CLOSE_WINDOW
}ACTION

R#FRMR ?RX_FRAME            ( Check event field for FRMR frame )
ACTION{
    1 COUNTER10 +!
    OPEN_USER
    11 38 THERE
    COUNTER10 @ W.          ( Display number of FRMR frames received )
    CLOSE_WINDOW
}ACTION

R#UI ?RX_FRAME              ( Check event field for UI frame )
ACTION{
    1 COUNTER11 +!
    OPEN_USER
    12 38 THERE
    COUNTER11 @ W.          ( Display number of UI frames received )
    CLOSE_WINDOW
}ACTION
```

```
    R#XID ?RX_FRAME          ( Check event field for XID frame )
    ACTION{
        1 COUNTER12 +!
        OPEN_USER
        13 38 THERE
        COUNTER12 @ W.        ( Display number of XID frames received )
        CLOSE_WINDOW
    }ACTION

    UF1 ?KEY                  ( Event is function key #1 )
    ACTION{
        SHOW_DATA             ( Show the Data Window )
    }ACTION

    UF2 ?KEY
    ACTION{
        1 RUN_SEQ             ( Show statistics )
    }ACTION

    UF3 ?KEY
    ACTION{
        0 RUN_SEQ             ( Zero counters )
    }ACTION
}STATE
```

LOAD_RETURN_STATE (number --)

Permits the test script writer to program the equivalent of subroutine calls (used with RETURN_STATE). LOAD_RETURN_STATE sets the state to which control is to be returned. LOAD_RETURN_STATE must be within the action field; nesting is not permitted.

RETURN_STATE (--)

Returns control to the state specified by LOAD_RETURN_STATE from a state subroutine call.

Example:
This example is taken from an ISDN test script. State 250 can be called from more than one action sequence. In the portion of code shown, state 0 calls '1 LOAD_RETURN_STATE' then '250 NEW_STATE'. After the execution of S_DISC in state 250, the program returns to state 1 when RETURN_STATE is executed.

```
TCLR
WAKEUP_ON

0 STATE{
    ?WAKEUP
    ACTION{
        T." Test started" TCR
        0 COUNTER !
        1 LOAD_RETURN_STATE
        21 1 START_TIMER
        250 NEW_STATE
    }ACTION
}STATE
```

```
1 STATE{
      R#UA ?RX_FRAME
      ACTION{
          BEEP
          " Press UF1 to release,"
          " UF2 to send packet,"
          " UF3 to send invalid packet"
          3 W.NOTICES
          2 NEW_STATE
      }ACTION
  }STATE

.     . .
.     . .
.     . .

250 STATE{
      21 1 ?TIMER
      ACTION{
          S_DISC          ( Send a disconnect )
          RETURN_STATE
      }ACTION
    }STATE
```

NEW_TM (filename --)

Loads and compiles the specified file and then starts the test manager at state 0. It can be included as part of the action field to load and execute another scenario.

Example:
In state 15, the file TEST2.F is called to start the execution of the next test which originates in state 0 of file TEST2.F.

```
15 STATE{
      21 ?TIMER
      ACTION{
          T." Test 1 is complete" TCR    ( Display Test 1 completion )
          " TEST2.F" NEW_TM              ( Load in Test 2 )
      }ACTION
   }STATE
```

17.8 Test Scripts

Test scripts can be loaded, run, stopped, and saved from the Command Window or from within other test scripts. Corresponding commands are described in the following sections.

Loading a Test Script

To run a test script, the file or a test script binary containing either source code must be loaded from a floppy or hard disk and run on an application processor.

The equivalent function key sequence (if applicable) is shown after the command.

EXECF (filename --)
Loads a source format test script. The filename must be entered in quotes.

Example:
```
" DR0:TEST_SEQ.F" EXECF
```

If there are no errors in the test script, the following notice message is displayed:
```
The (DR0:TEST_SEQ.F) test script is loaded.
```

TestScript topic
Load Script function key

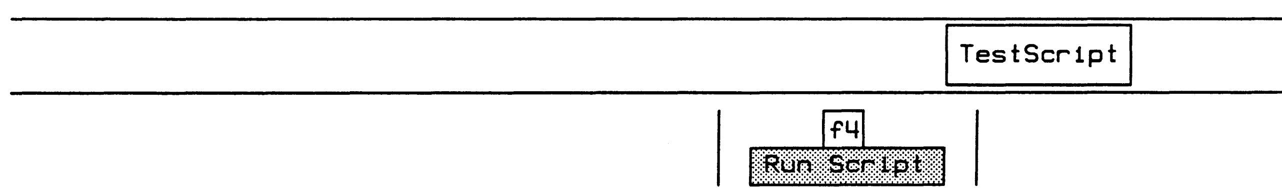
EXECTS (filename --)
Loads a binary format test script. The filename is entered in quotes.

Example:
```
" DR0:TEST_SEQ.B" EXECTS
```

If there are no errors in the test script, the following notice message is displayed:
```
The (DR0:TEST_SEQ.F) test script is loaded.
```

Starting a Test Script

To start the test script:



If the following error message is displayed, either the test script was not loaded or the test script is incomplete:

```
State 0 is undefined: ...
```

The *Run Script* function key remains highlighted while the test script is running.

The test manager can also be started by typing the following command in the Command Window:

TM_RUN (--)
Starts the execution of the test script. If a test script was not loaded or the test script was incomplete, the following error message is displayed:

```
State 0 is undefined: ...
```

Because ITL's state machine architecture requires an event to occur before any actions execute, eg. function key, received frame, etc., the wakeup mechanism provides a single event upon test program startup.

When either the *Run Script* function key is pressed or the TM_RUN command is issued, the wakeup timer is started for 100 milliseconds. This allows sufficient time for any initialization code to execute, after which time, the test program detects the expired timer and performs the required actions.

WAKEUP_ON (--)
Activates the wakeup timer (timer 34) for 100 milliseconds when either the *Run Script* function key is pressed or the TM_RUN command is issued. If ?WAKEUP is used in State 0, the scenario execution starts. The default is WAKEUP_OFF.

Example:
Upon expiry of the wakeup timer, label four keys, and go to State 1.

```
TCLR
WAKEUP_ON

0 STATE{
      ?WAKEUP
      ACTION{
            " Show Stat" 1 LABEL_KEY
            " Show Data" 2 LABEL_KEY
            " Clear" 3 LABEL_KEY
            " Stop Test" 4 LABEL_KEY
            1 NEW_STATE
      }ACTION
  }STATE
```

WAKEUP_OFF (--)
Deactivates the wakeup timer (default). With WAKEUP_OFF, the wakeup timer is not started when the test script is run.

Stopping a Test Script

TM_STOP (--)
Stops the execution of a test script. The test suite remains in memory and can be re-executed using TM_RUN.

Saving a Test Script Binary

There are two methods of saving a test script binary. The first method uses the SAVETS command and saves a binary which can be loaded using the *Load Script* function key. The second method uses the ASTART_ON command which automatically runs a test scenario at program startup.

SAVETS (filename --)
Saves the compiled test script as a binary with the specified filename. To save a test script binary:

- Load a source test script as instructed in the 'Loading a Test Script' section on page 17-39.
- Press the **ESC** key to open the Command Window.
- Type: " filename" SAVETS.

ASTART_ON (--)
Automatically executes a test scenario at program startup.

- Load the emulation.
- Change to the desired parameters.
- Load the test scenario.
- Press the **ESC** key to open the Command Window.
- Type: ASTART_ON.
- Resave the binary by typing: " Program Name" SAVEB.

The binary is saved as 'filename.extension' where valid filenames are listed in Table 17-5, and valid extensions in Table 17-6 (depending on machine configuration).

Application	Filename
ISDN Monitor	ISDN_MON
ISDN Emulation	ISDN_EMUL
Universal Monitor	USM_MON
Universal Simulator	USM_SIM
X.25 Monitor	X25_MON
X.25 Emulation	X25_EMUL
X.25 MLP Emulation	X25_EMUL
X.25 Load Generator	X25_LOAD
SNA Monitor	SNA_MON
SDLC Emulation	SDLC_MON
Bisync Monitor	BSC_MON
Bisync Emulation	BSC_EMUL
X.75 Monitor	X75_MON
X.75 Emulation	X75_EMUL
SNA Verification	SNA_VER
Teletex Monitor	TTX_MON
X.25 Network Performance Analysis	X25_STAT
SNA Network Performance Analysis	SNA_STAT
SS#7 Monitor	SS7_MON
SS#7 Simulation	SS7_SIM

Table 17-5 Binary Filename Prefixes

	D-Chan	WAN	PRA	PRA/WAN	BRA	BRA/WAN	BRA/BRA	PRA/BRA/ WAN	WAN/WAN
AP #1		B1	B3	B1	B1	B1	B1	B1	B1
AP #2			B4	B3	B2	B2	B2	B2	B2
AP #3	B			B4	B	B	B	B	
AP #4							B3	B3	
AP #5							B4	B4	
AP #6							B5		

Table 17-6 Binary Filename Extensions

Whenever this binary is loaded, the configuration is set as previously defined and the test manager executes automatically.

The same steps are followed for the monitor but the binary would be saved by typing:
" X25_EMUL.B1" SAVEB.

NOTE
The ASTART feature is not available in ISDN or SS#7 applications.

ASTART_OFF (--)
Disables the automatic execution of a test script at program startup (default).

18
CONFIGURATION FILE

When the menu system software is loaded, the default configuration source file (HOME.D) is executed which automatically configures the remote and printer ports.

Similarly, when a monitor or emulation application is loaded from the Home processor, a corresponding default configuration file is executed which automatically configures the application. These configuration files are named 'filename.extension' where valid filenames are listed in Table 17–5, and valid extensions in Table 18–1 (depending on machine configuration).

These default configuration files contain executable ITL commands and can be customized using the editor under the **Files** topic.

Example:
The following default configuration file can be edited on AP#1 on a BRA/WAN machine to customize the Bisync Emulation program:
`BSC_EMUL.D1`

	D–Chan	WAN	PRA	PRA/WAN	BRA	BRA/WAN	BRA/BRA	PRA/BRA/WAN	WAN/WAN
AP #1		D1	D3	D1	D1	D1	D1	D1	D1
AP #2			D4	D3	D2	D2	D2	D2	D2
AP #3	D			D4	D	D	D	D	
AP #4							D3	D3	
AP #5							D4	D4	
AP #6							D5		

Table 18–1 Configuration File Name Extensions

The status of the configuration file, is displayed as either a notice or error message.

For example, one of the following notices could be displayed for emulation:

- `Executed configuration file : X25_EMUL.D1`
  Indicates that the configuration file is loaded and all commands are executed.
- `Not able to find configuration file : X25_EMUL.D1`
  Indicates that the configuration file is not found on any of the floppy disk drives or hard disk partitions.

NOTE
If the configuration file does not exist, the flow of the program will not be affected.

For example, one of the following error messages could be displayed for the emulation:

- `Code error in configuration file : X25_EMUL.D1`
  Indicates that the configuration file is loaded, but contains a coding error.
- `File : X25_EMUL.D1  Unrecoverable Physical I/O Error!  Disk not formatted?`
  Indicates that a disk error occurred during loading.

See the appropriate Programmer's Manual for protocol specific configuration commands. See Sections 9.1 and 10 in this manual for printer and remote port configuration commands.

A
TEST SCRIPT COMPATIBILITY

This appendix describes commands available in previous versions of the test manager. These commands are still supported but are not recommended for use with the current version of the test manager. The procedure for using these commands might have changed.

A.1 ?KEYBOARD

?KEYBOARD (--)

NOTE
It is recommended that all occurrences of ?KEYBOARD be replaced with the PROMPT" string" *END_PROMPT partners.*

Detects keyboard entry of character strings or numerical values within the event field. ?KEYBOARD requires no parameters. The interpreter does not process keyboard entries in states which contain ?KEYBOARD. A true flag is returned when a keyboard entry is terminated with the RETURN key.

WARNING
To use ?KEYBOARD, follow this procedure:

- Move the topic box to the **TestScript** topic.
- Press the *Script Window* function key.
- Press the *Script Keys* function key.

Any characters typed on the keyboard are echoed in the Test Script Window. Pass the string to the test script by pressing ↵ (RETURN). While the *Script Keys* function key is highlighted, the topic bar is removed from the screen.

A.2 Numerical Value Entry

Numerical entry can be accessed by the EXPECT_NUM command.

EXPECT_NUM (-- d\f)

Converts a character entry into a numerical value and returns a value and a true flag if successful.

Example

Set timer 21 to the value of time entered.

```
2 STATE{
     ?KEYBOARD                         ( Check for keyboard entry )
     ACTION{
         EXPECT_NUM                    ( Get value and flag on stack )
         IF
             21 SWAP START_TIMER       ( Start timer 21 )
             3 NEW_STATE               ( Go to state 3 )
         ELSE
             DROP                      ( If invalid, remove value from stack )
             " Invalid entry" W.NOTICE ( Advise user of invalid entry)
         ENDIF
     }ACTION
  }STATE
```

A.3 Output

The following commands generate comments or prompts to either the Script Window or the Command Window.

NOTE

It is recommended that all occurrences of these commands be replaced with the PROMPT" string" *END_PROMPT partners.*

WARNING

To use these commands, use one of the following procedures:

To direct text to the Script Window:

- Move the topic box to the **TestScript** topic.
- Press the *Script Window* function key.

To direct text to the Command Window:

- Press the **ESC** key.

DISPLAY (string\attr --)
Prints the string in the active window using the color specified by 'attr'. See Table 9-1 for valid attributes.

NOTE
There must be a single space inserted after the first quotation mark prior to the text.

Example:
Display text using a red background in the active window.
```
" Do you wish to continue? [Y/N]:" RED_BG DISPLAY
```

." text" (--)
(dot quote)
Prints the enclosed text in the currently open window. The maximum number of characters is 255.

NOTE
There must be a single space inserted after the first quotation mark prior to the text.

Example:
Print the text in the currently open window.
```
." Do you wish to continue? [Y/N] :"
```

. (d --)
(dot)
Prints a signed 32 bit from a two's complement value converted to the current numeric base. The default numerical base is decimal. A trailing space follows the display of the number.

Example:
Display the contents of COUNTER5 in the currently active window.
```
COUNTER5 @ .
```

.H (d --)
(dot H)
Displays a 4 byte (32 bit) hex value.

Example:
Display 0001E240 in the currently open window.
```
123456 .H
```

.HB (d --)
(dot HB)
Displays a one byte (8 bit) hex value.

Example:
Display 7F in the currently open window.
```
127 .HB
```

.HH (d --)

(dot HH)

Displays a half byte (4 bit) hex value.

Example:

Display F in the currently open window.

```
127 .HH
```

A.4 Detecting Cursor Keys

WARNING

Cursor keys are passed to the test manager only when the Script Keys function key under the TestScript topic has been pressed.

Cursor keys can be detected with the ?KEY command. Appropriate values passed to ?KEY to detect cursor movements are listed below.

CUP	⇑	Up cursor key
CDOWN	⇓	Down cursor key
CLEFT	⇐	Left cursor key
CRIGHT	⇒	Right cursor key

The **HOME** key and **ESC** key cannot be detected. **HOME** always returns the user to the Home processor. **ESC** always opens the Command Window.

Example:

Detect the pressing of the UP arrow key (⇑).

```
CUP ?KEY
ACTION{
    ...
}ACTION
```

B
ERROR RECOVERY

IDACOM testers are built around the Motorola MC68000 family of processors and contain from three to seven CPU's, depending on machine configuration.

B.1 Description

Bugs in user test scripts or improperly used ITL commands might cause program malfunction or a system crash. These errors generally fall into two categories: memory and arithmetic.

The MC68000 views memory as a continuous sequence of bytes 0 through 0xFFFFFF, or 16 Mb. Operations on this memory space must use a 32, 16, or 8 bit operand size.

All memory addresses, with the exception of 8 bit operations, must be specified as an *even* number. Passing an odd address to '@' or 'W@' creates an *address error exception*. This causes all activity on the application processor to halt and a traceback of the error to print on the screen (see Figure B-1).

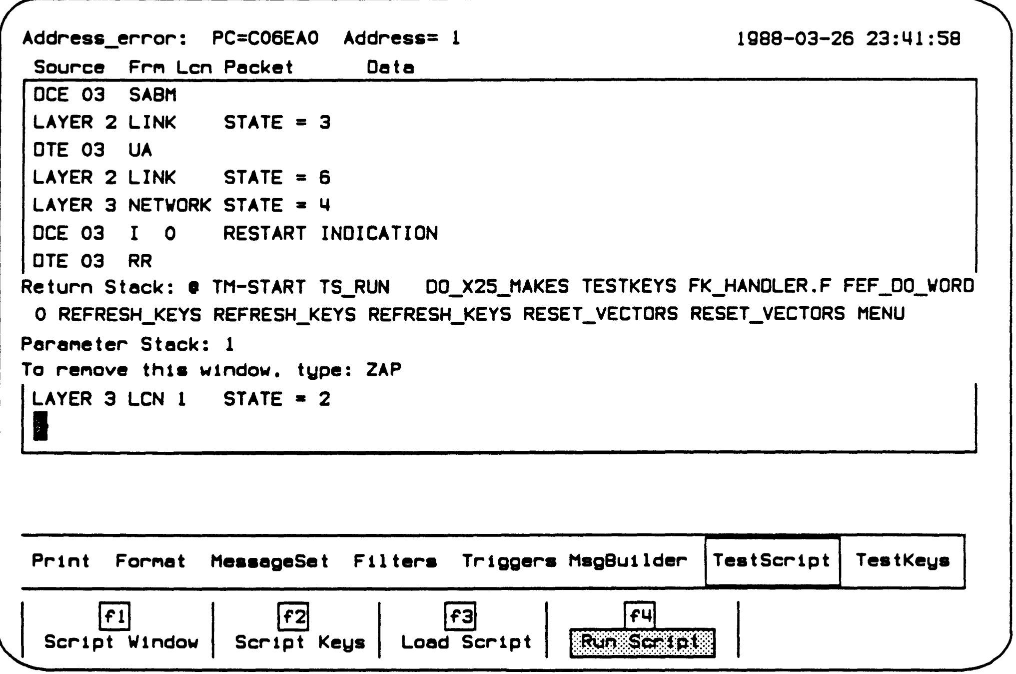


Figure B-1 Address Error Screen Display

A memory operation on an address where no actual memory exists generates a *bus error exception.* This usually indicates a pointer variable within a test program that is not being set or referenced correctly.

Attempting to divide by zero generates a *zero divide exception.*

All exceptions produce a similar screen display and halt all activity on the application processor. The emulation and test script stop running while the error message is displayed.

B.2 Recovery

Whenever an exception halts all processor activity, the application program must be reset and restarted.

- ❑ Press SHIFT/CONTROL f8 simultaneously.

The screen should clear and display a blank window.

- ❑ Type MENU and press ↵ (RETURN) to re-initialize the application program.

If this operation fails to clear the screen, reboot the unit by pressing the reset switch.

C
CODING CONVENTIONS

This section outlines some coding and style conventions recommended by IDACOM. Although you can develop your own style, it is suggested to stay close to these standards to enhance readability.

C.1 Stack Comments

A stack comment is surrounded by parentheses, and shows two stack pictures. The first picture shows any items or 'input parameters' that are consumed by the command; the second picture shows any items or 'output parameters' returned by the command.

Example:
The '=' command has the following stack comment.

(n_1\n_2 -- flag)

In this example, n_1 and n_2 are numbers and the flag is either 0 for a false result, or 1 for a true result. This same example could also be written as follows.

(n_1\n_2 -- 0|1)

The '\' character separates parameters when there is more than one. The parameters are listed from left to right with the leftmost item representing the bottom of the stack and the rightmost item representing the top of the stack.

The '|' character indicates that there is more than one possible output. The above example indicates that either a 0 (false result) or a 1 (true result) is returned on the stack after the '=' operation.

C.2 Stack Comment Abbreviations

Following is a list of commonly used abbreviations. In most cases, the stack comments shown in this manual have been written in full rather than abbreviated.

Symbol	Description
a	Memory address
b	8 bit byte
c	7 bit ASCII character
n	16 bit signed integer
d	32 bit signed integer
u	32 bit unsigned integer
f	Boolean flag (0=false, non-zero=true)
ff	Boolean false flag (zero)
tf	Boolean true flag (non-zero)
s	String (actual address of a character string which is stored in a count prefixed manner)

Table C-1 ITL Symbols

C.3 Program Comments

Program comments appear in source code surrounded by parentheses. These describe the intent or purpose of the definition or line of code.

There must be at least one space on each side of the parentheses.

Example:

```
:  HELLO   ( -- )                    ( Display text Hello in Notice Window )
       " HELLO"                      ( Create string )
       W.NOTICE                      ( Output to Notice Window )
;
```

The program comment should be kept to a minimum and yet contain enough information that another programmer can tell the intent at a glance.

C.4 Test Manager Constructs

Coding conventions for user test scripts should generally follow the style presented throughout this manual.

Indenting nested program structures should be done using the tab key in the editor. The use of many meaningful comments is highly recommended to enhance the continued maintainability of the program.

Example:
(State definition purpose comment)

```
0 STATE{
     EVENT Recognition Commands         ( Comment )
     ACTION{
         Action Commands                ( Comment )
         IF
             ...                        ( Comment )
             ...                        ( Comment )
         ENDIF
     }ACTION
  }STATE
```

C.5 Spacing and Indentation Guidelines

The following list outlines the general guidelines for spacing and indentations:

- One space between colon and name in colon definitions.
- One space between opening parenthesis and text in comments.
- One space between numbers and words within a definition.
- One space between initial " in strings (i.e. with " string", W." string", T." string", P." string", X" hex characters", etc...)
- One or more spaces at the end of each line unless defining a string which requires additional characters.
- Tab for nested constructs.
- Carriage return after colon definition and stack comment.
- Carriage return after last line of code in colon definition and semi-colon.

See the examples in Appendices C.6 and C.4.

C.6 Colon Definitions

The colon definition should be preceded by a short comment and should start at the first column of a line. All codes underneath the definition name should be preceded by one tab. Each element within the colon definition should be well defined.

Example:
(Description of command)

```
:  COMMANDNAME                        ( Stack description )
        .....                         ( Comment for first line of code )
        IF
            ....                      ( Comment )
            DOCASE
                CASE  X  [ ... ]      ( Comment )
                CASE  Y  [ ... ]      ( Comment )
                CASE  DUP  [ ... ]    ( Comment )
            ENDCASE
        ELSE
            BEGIN
                .....                 ( Comment )
                .....                 ( Comment )
            UNTIL
        ENDIF
;
```

D
ASCII/EBCDIC/HEX CONVERSION TABLE

HEX	DEC	OCT	ASCII	EBCDIC
00	0	00	NUL	NUL
01	1	01	SOH	SOH
02	2	02	STX	STX
03	3	03	ETX	ETX
04	4	04	EOT	PF
05	5	05	ENQ	HT
06	6	06	ACK	LC
07	7	07	BEL	DEL
08	8	10	BS	GE
09	9	11	HT	SPS
0A	10	12	LF	RPT
0B	11	13	VT	VT
0C	12	14	FF	FF
0D	13	15	CR	CR
0E	14	16	SO	SO
0F	15	17	SI	SI
10	16	20	DLE	DLE
11	17	21	DC1	DC1
12	18	22	DC2	DC2
13	19	23	DC3	DC3
14	20	24	DC4	RES
15	21	25	NAK	NL
16	22	26	SYN	BS
17	23	27	ETB	POC
18	24	30	CAN	CAN
19	25	31	EM	EM
1A	26	32	SUB	UBS
1B	27	33	ESC	CUI
1C	28	34	FS	IFS
1D	29	35	GS	IGS
1E	30	36	RS	IRS
1F	31	37	US	IUS
20	32	40	SP	DS
21	33	41	!	SOS
22	34	42	"	FS
23	35	43	#	WUS
24	36	44	$	BYP
25	37	45	%	LF
26	38	46	&	ETB
27	39	47	'	ESC
28	40	50	(	SA
29	41	51	)	SFE
2A	42	52	*	SM/SW
2B	43	53	+	CSP
2C	44	54	,	MFA
2D	45	55	–	ENQ
2E	46	56	.	ACK
2F	47	57	/	BEL

HEX	DEC	OCT	ASCII	EBCDIC
30	48	60	0	
31	49	61	1	
32	50	62	2	SYN
33	51	63	3	IR
34	52	64	4	PP
35	53	65	5	TRN
36	54	66	6	NBS
37	55	67	7	EOT
38	56	70	8	SBS
39	57	71	9	IT
3A	58	72	:	RFF
3B	59	73	;	CU3
3C	60	74	<	DC4
3D	61	75	=	NAK
3E	62	76	>	
3F	63	77	?	SUB
40	64	100	@	SP
41	65	101	A	
42	66	102	B	
43	67	103	C	
44	68	104	D	
45	69	105	E	
46	70	106	F	
47	71	107	G	
48	72	110	H	
49	73	111	I	
4A	74	112	J	cent
4B	75	113	K	.
4C	76	114	L	<
4D	77	115	M	(
4E	78	116	N	+
4F	79	117	O	|
50	80	120	P	&
51	81	121	Q	
52	82	122	R	
53	83	123	S	
54	84	124	T	
55	85	125	U	
56	86	126	V	
57	87	127	W	
58	88	130	X	
59	89	131	Y	
5A	90	132	Z	!
5B	91	133	[	$
5C	92	134	\	
5D	93	135	]	)
5E	94	136	^	;
5F	95	137	_	¬

HEX	DEC	OCT	ASCII	EBCDIC	HEX	DEC	OCT	ASCII	EBCDIC
60	96	140	`	-	90	144	220		
61	97	141	a	/	91	145	221		j
62	98	142	b		92	146	222		k
63	99	143	c		93	147	223		l
64	100	144	d		94	148	224		m
65	101	145	e		95	149	225		n
66	102	146	f		96	150	226		o
67	103	147	g		97	151	227		p
68	104	150	h		98	152	230		q
69	105	151	i		99	153	231		r
6A	106	152	j	|	9A	154	232		
6B	107	153	k	,	9B	155	233		}
6C	108	154	l	%	9C	156	234		□
6D	109	155	m	_	9D	157	235		)
6E	110	156	n	>	9E	158	236		±
6F	111	157	o	?	9F	159	237		■
70	112	160	p		A0	160	240		-
71	113	161	q	^	A1	161	241		o
72	114	162	r		A2	162	242		s
73	115	163	s		A3	163	243		t
74	116	164	t		A4	164	244		u
75	117	165	u		A5	165	245		v
76	118	166	v		A6	166	246		w
77	119	167	w		A7	167	247		x
78	120	170	x		A8	168	250		y
79	121	171	y	\	A9	169	251		z
7A	122	172	z	:	AA	170	252		
7B	123	173	{	#	AB	171	253		└
7C	124	174	|	@	AC	172	254		┌
7D	125	175	}	'	AD	173	255		[
7E	126	176	~	=	AE	174	256		≥
7F	127	177	DEL	"	AF	175	257		•
80	128	200			B0	176	260		0
81	129	201		a	B1	177	261		1
82	130	202		b	B2	178	262		2
83	131	203		c	B3	179	263		3
84	132	204		d	B4	180	264		4
85	133	205		e	B5	181	265		5
86	134	206		f	B6	182	266		6
87	135	207		g	B7	183	267		7
88	136	210		h	B8	184	270		8
89	137	211		i	B9	185	271		9
8A	138	212			BA	186	272		
8B	139	213		{	BB	187	273		┘
8C	140	214		≤	BC	188	274		┐
8D	141	215		(	BD	189	275		]
8E	142	216		+	BE	190	276		≠
8F	143	217		ǂ	BF	191	277		-

HEX	DEC	OCT	ASCII	EBCDIC
C0	192	300		{
C1	193	301		A
C2	194	302		B
C3	195	303		C
C4	196	304		D
C5	197	305		E
C6	198	306		F
C7	199	307		G
C8	200	310		H
C9	201	311		I
CA	202	312		
CB	203	313		
CC	204	314		
CD	205	315		
CE	206	316		
CF	207	317		
D0	208	320		}
D1	209	321		J
D2	210	322		K
D3	211	323		L
D4	212	324		M
D5	213	325		N
D6	214	326		O
D7	215	327		P
D8	216	330		Q
D9	217	331		R
DA	218	332		
DB	219	333		
DC	220	334		
DD	221	335		
DE	222	336		
DF	223	337		
E0	224	340		\
E1	225	341		
E2	226	342		S
E3	227	343		T
E4	228	344		U
E5	229	345		V
E6	230	346		W
E7	231	347		X
E8	232	350		Y
E9	233	351		Z
EA	234	352		
EB	235	353		
EC	236	354		
ED	237	355		
EE	238	356		
EF	239	357		

HEX	DEC	OCT	ASCII	EBCDIC
F0	240	360		0
F1	241	361		1
F2	242	362		2
F3	243	363		3
F4	244	364		4
F5	245	365		5
F6	246	366		6
F7	247	367		7
F8	248	370		8
F9	249	371		9
FA	250	372		
FB	251	373		
FC	252	374		
FD	253	375		
FE	254	376		
FF	255	377		

COMMAND CROSS REFERENCE LIST

This appendix cross references old commands and variables, not appearing in this manual, with new replacement commands. Reference should be made to the previous versions of this manual for description of the old commands. The new commands achieve the same function, however, the input/output parameters may have changed.

Old Command	New Command
<#...#>	#>STR
#	#>STR
#S	#>STR
HOLD	#>STR
SIGN	#>STR

INDEX

@, 3–1, 17–28
>>#, 5–3
>>, 5–3
>, 7–1, 17–29
=, 7–1, 17–29
<<#, 5–3
<<, 5–2
<, 7–2, 17–29
;, 16–1
:, 16–1
/, 4–2
.", A–3
., A–3
–, 4–1
+$, 8–3
+!, 3–2, 4–1, 17–28
+, 4–1
*, 4–1
$=, 8–3
$!, 8–3
!, 3–1, 17–28

0=, 7–4

2DROP, 15–1
2DUP, 15–1

3DROP, 15–1
3DUP, 15–1

ABS, 4–2
Absolute value, 4–2
ACTION{ }ACTION, 17–6
Addition, 4–1
  high speed, 4–3
  timestamps, 12–4
Address Error, *see* Error(s)
Alarms, 13–3, 17–18
ALLOT, 3–4
ALL_LEADS, 17–8
Anchored Match, *see* Search
AND, 5–2, 17–30
  combining expressions, 7–3
;AND, 16–2
#arg, 17–25, 17–26
Arithmetic Operations, 4–1 *to* 4–3
  addition, 4–1, 4–3
  division, 4–2, 4–3
  multiplication, 4–1 *to* 4–3
  subtraction, 4–1
  timestamps, 12–4
ASTART_OFF, 17–43
ASTART_ON, 17–41
Attribute(s)
  character sets, 9–9
  color, 9–3
  color to monochrome, 9–4
  in user window, 9–7, 9–9 *to* 9–14
  monochrome, 9–3
  monochrome to color, 9–4
AUTO_CLEAR_OFF, 9–9
AUTO_CLEAR_ON, 9–9

B, *see* BACKWARD
Background
  data window, 9–1
  user window, 9–6
BACKWARD, 11–6
BASE, 8–5
BB, *see* SCRN_BACK
BEEP, 13–3, 17–18
BEEP_DUR, 13–3
BEEP_OFF, 13–3, 17–18
BEEP_ON, 13–3, 17–18
BEEP_TONE, 13–3
BEGIN/AGAIN, 6–6
BEGIN/UNTIL, 6–6
BEGIN/WHILE/REPEAT, 6–7

BETWEEN?, 7–2
BIN, 8–5
Bit Manipulation, 5–1 *to* 5–3
BLK_BG, 9–3
BLU_BG, 9–3
BLU_FG, 9–3
Boolean Operators
  AND, 5–2, 17–30
  exclusive OR, 5–2, 17–30
  OR, 5–1, 17–30
BOTTOM, 11–5
Boundaries, checking, 7–2, 7–5
BRITE, 9–3
BRITE INVERSE, 9–3
Buffer(s)
  allocating, 3–4
  copying, 3–5
  searching, 8–3
  timestamps, 12–2
Bus Error, *see* Error(s)

C!, 3–3
C@, 3–3
Capture RAM
  playback from remote port, 11–5, 11–6
  saving data to, 17–16
  trace statements, 9–4
  transferring from, 11–6
CAPT_OFF, 17–16
CAPT_ON, 17–16
Carriage Return
  during printing, 10–3
  in trace statements, 9–3
  in user window, 9–7
  remote port, 11–4
Carrier Detect, *see* Leads, interface
CDOWN, A–4
Character
  displaying, 9–2, 9–7
  fetch, 3–3
  printing, 10–3
  store, 3–3
Character Sets, 9–9
CHARS/LINE, 10–2
CHECKSUM_CORRECTION, 11–7
CLEAN_WINDOW, 9–9
Clear to Send, *see* Leads, interface
CLEAR_KEY, 17–23
CLEAR_KEYS, 17–23
CLEAR_ROW, 9–9
CLEAR_TEXT, 9–9
CLEFT, A–4
CLOSE, 13–6
CLOSE_FILE, 13–6
CLOSE_WINDOW, 9–6
CMOVE, 3–5, 8–2
<CMOVE, 3–6
Colon Definitions, 16–1
Color
  attributes, 9–3
  in trace statements, 9–3
  in user window, 9–9 *to* 9–14
  to monochrome, 9–4
Column Number, 9–8
Command Window
  output, A–2
Command(s)
  creating, 16–1
  format, 1–3
  pointers to, 16–2
  remote processor execution, 16–3
Comparison, 7–1 *to* 7–5
  boolean negation, 7–4
  combining expressions, 7–3
  equality, 7–1, 17–29
  false flag, 7–1
  fast zero equality, 7–4
  greater than, 7–1, 17–29

INDEX [continued]

Comparison *[continued]*
  in test manager, 17-7
  less than, 7-2, 17-29
  maximum, 7-5
  minimum, 7-5
  range checking, 7-2
  timestamps, 12-5
  true flag, 7-1
Compiler Control, 14-1, 14-2
  conditional compilation, 14-1
  conditional definition, 14-2
  #IF/#ELSE/#ENDIF, 14-1
  #IFDEF/#ENDIF, 14-2
  #IFNOTDEF/#ENDIF, 14-2
Computational Stack, *see* Stack(s)
CONFIG, 10-2, 11-2
Configuration
  printer port, 10-1
  remote port, 11-1, 11-2
Control Structures, 6-1 *to* 6-7
  BEGIN/AGAIN, 6-6
  BEGIN/UNTIL, 6-6
  BEGIN/WHILE/REPEAT, 6-7
  compiler, 14-1, 14-2
  DO/+LOOP, 6-5
  DO/LOOP, 6-3
  DOCASE/ENDCASE, 6-2
  IF/ELSE/ENDIF, 6-1
  IF/ENDIF, 6-1
  index, 6-3
  mixed, 6-2
  nested, 6-2, 6-4
CONTROL-Z<>EOF, 11-7
CONTROL-Z=EOF, 11-7
Conversion
  strings to numbers, 8-6
  timestamps, 12-3
CONV_STR, 8-4
Copy
  block in memory, 3-5, 3-6
  strings, 8-2, 8-3
  timestamps, 12-5
  value on stack, 15-1
COUNT, 8-2
COUNTER1, 17-28
Counters, 17-28 *to* 17-30
CPU1, 13-8
CPU1_MAIL, 17-31
CPU2, 13-8
CPU3, 13-8
CPU4, 13-8
CPU5, 13-8
CPU7, 13-8
CRC_CORRECTION, 11-7
CREATE_FILE, 13-5
Creating Commands, 16-1 *to* 16-3
CRIGHT, A-4
CTRACE, 9-4
CUP, A-4
Cursor
  control, 9-7
  keys, A-4
CURSOR_OFF, 9-7
CURSOR_ON, 9-7
CYA_BG, 9-3
CYA_FG, 9-3

Data Set Ready, *see* Leads, interface
Data Signal Rate Select, *see* Leads, interface
Data Terminal Ready, *see* Leads, interface
Data Types
  numbers, 2-1
  strings, 2-1, 2-2
Data Window
  displaying, 9-1
  trace reporting, 9-1 *to* 9-5
DCE/DTE, *see* Port Identifiers
DECIMAL, 8-5
Decrement Number, *see* Memory
DEFAULT_KEYS, 17-23
Delete Value(s) on Stack, 15-1
DEST_DRIVE, 11-7
DIM, 9-3
DIM INVERSE, 9-3
Directory Listing, 11-5
DISABLE_LEAD, 17-8
Disk Files, *see* File(s)
Disk Recording, *see* Recording to Disk
DISK_OFF, 17-17
DISPLAY, A-3
Display
  32 bit hex number, 9-2, 9-7, A-3
  4 bit hex number, A-4
  8 bit hex number, A-3
  errors, 17-25, 17-26
  notices, 17-24, 17-25
  numbers, 9-2, 9-6, A-3
  remote screen, 11-4
  single character, 9-2, 9-7
  starting/stopping, 17-16
  test manager RAM, 17-2
  trace statements, 9-4
Division
  32 bit, 4-2
  high speed, 4-3
  remainder only, 4-2
  signed, 4-2
  unsigned, 4-2
DO/+LOOP, 6-5
DO/LOOP, 6-3
  index, 6-3
  nested, 6-4
  terminate, 6-5
DOCASE/ENDCASE, 6-2
DOER, 16-2
DR0, 13-4
DR1, 13-4
Drive Selection, 13-4
DROP, 15-1
DROP_TEST, 9-15
DTRACE, 9-5
DUP, 15-1
Duplicate, 15-1

ENABLE_LEAD, 17-8
EOL=CR, 11-6
EOL=CRLF, 11-6
Equal To, 7-1, 7-4, 17-29
Error(s)
  address error, B-1
  bus error, B-1
  displaying, 17-25, 17-26
  file, 13-5
  messages, 17-25
  recovery, B-2
  traceback, B-1
  window, 17-26
  zero divide, B-2
Errors
  during file transfer, 11-7
Event Recognition, 17-7 *to* 17-15
  cursor keys, A-4
  frame, 17-10
  function keys, 17-11, 17-12
  layer 1, 17-8, 17-9
  mail, 17-12
  timer, 17-10, 17-11
  wildcards, 17-14, 17-15
EVENT-TYPE, 17-14
EXECF, 17-39
EXECTS, 17-39
EXPECT_NUM, A-2
Expressions, 7-3
EXTRACT_FTH_DATA, 17-13

F, *see* FORWARD

INDEX [continued]

F1_ACTION, 17-24
False Flag, *see* Comparison
FDSTATUS, 13-5
FEF_DO_WORD, 16-3
Fetch
  16 bit value, 3-2
  32 bit value, 3-1, 17-28
  8 bit value, 3-3
FF, *see* SCRN_FWD
File Transfer
  configuration, 11-7
File(s)
  accessing, 13-4 *to* 13-7
  closing, 13-6
  creating, 13-5
  disk recording, *see* Recording to Disk
  errors, 13-5
  opening, 13-6
  reading, 13-6
  removing, 13-7
  renaming, 13-7
  status, 13-5
  transferring, 17-1
  writing, 13-6
FILEX, 17-1
FILL, 3-5
FILLW, 3-5
FORWARD, 11-5
Frame(s)
  event recognition, 17-10
  timestamps, 12-1
FROM_CAPT, 11-5
FTH, 13-9
FULL_DUPLEX, 11-6
Function Key(s)
  actions, 17-24
  event recognition, 17-11, 17-12
  labelling, 17-22
  user-defined, 17-18 *to* 17-24

GET_TS, 12-5
GET_TSTAMP_MICRO, 12-3
GET_TSTAMP_MILLI, 12-3
Greater Than, 7-1, 17-29
GRN_BG, 9-3
GRN_FG, 9-3

.H, A-3
HALF_DUPLEX, 11-6
HALT, 11-5
Hard Disk
  partitions, 13-4
  selection, 13-4
  shutdown, 13-4
.HB, A-3
HEX, 8-5
Hex Number(s)
  displaying, A-3, A-4
  in trace statements, 9-2
  in user window, 9-7
  printing, 10-3
.HH, A-4
HILITE_LABEL, 17-23

I/F_TYPE, 10-1
#IF/#ELSE/#ENDIF, 14-1
IF/ELSE/ENDIF, 6-1
IF/ENDIF, 6-1
#IFDEF/#ENDIF, 14-2
#IFNOTDEF/#ENDIF, 14-2
Increment Number, *see* Memory
Index
  loop counter, 6-3
  nested loop counter, 6-4
Input, 17-18 *to* 17-24
  keyboard entry, A-2
  prompts, 17-18
Interprocessor Mail, 17-12

ITL, 1-1

J, *see* Index, nested

?KEY, 17-12
?KEYBOARD, A-1

LABEL_KEY, 17-22
Lapse Timer(s)
  milliseconds elapsed, 17-27
  minutes elapsed, 17-26
  seconds elapsed, 17-27
  starting, 17-26
Layer 1
  event recognition, 17-8, 17-9
Leads
  interface, 17-9
LEAVE, 6-5
Less Than, 7-2, 17-29
LINES/PAGE, 10-2
Listing Directory, 11-5
Loading Test Scripts, 17-39
LOAD_RETURN_STATE, 17-37
LOCK_LOGO, 11-4
Logical Operations
  AND, 5-2, 17-30
  boolean negation, 7-4
  combining expressions, 7-3
  comparison words, *see* Comparison
  exclusive OR, 5-2, 17-30
  OR, 5-1, 17-30
  shift left, 5-2, 5-3
  shift right, 5-3
  test manager, 17-7

M*, 4-2
M/, 4-2
MAG_BG, 9-3
MAG_FG, 9-3
Mail
  between processors, 17-31, 17-32
  event recognition, 17-12, 17-13
?MAIL, 17-13
MAIL_CMD, 17-32
MAIN, 13-8
MAKE, 16-2
?MATCH, 8-4
MAX, 7-5
Maximum, 7-5
Memory
  accessing, *see* Fetch
  allocating, 3-4
  decrement number in, 3-2, 4-1
  filling, 3-5
  increment number in, 3-2, 4-1
  moving/copying block, 3-5
  operations, 3-1 *to* 3-3
  storage, 2-1
  strings, 8-1
  test manager, 17-2
MENU, B-2
MILLISECONDS_ELAPSED, 17-27
MIN, 7-5
Minimum, 7-5
MINUTES_ELAPSED, 17-26
MOD, 4-2
Monochrome
  attributes, 9-3
  to color, 9-4
Move
  block in memory, 3-5
  strings, 8-2
Multiplication
  32 bit, 4-2
  high speed, 4-3
  signed, 4-1
  unsigned, 4-2

INDEX [continued]

Negation, 7-4
Nested
  DO/LOOP, 6-4
  IF/ENDIF, 6-2
NEW_STATE, 17-7
NEW_TM, 17-38
Notice Window, 17-24
Notice(s)
  displaying, 17-25
Notices, displaying, 17-24
NO_TRUNCATE, 13-6
Number(s)
  bases, 8-5
  converting to strings, 8-5
  data types, 2-1
  displaying, 9-2, 9-6, A-3
  in user window, 9-6
  printing, 10-3
  trace statements, 9-2

OCTAL, 8-5
OPEN_FILE, 13-6
OPEN_TEST, 9-15
OPEN_USER, 9-6
OR, 5-1, 17-30
  combining expressions, 7-3
OTHER_EVENT, 17-14
Output, 9-1 *to* 9-15, A-2
  errors, 17-26
  notices and errors, 17-24 *to* 17-26
  to the data window, 9-1 *to* 9-5
  to the remote port, 11-3, 11-4
  to the Test Script Window, 9-14, 9-15
  to the user window, 9-5 *to* 9-14
OVER, 15-2

P., 10-3
P.", 10-3
P.H, 10-3
P.TYPE, 10-3
PAINT, 9-9
PAINT_FIELD, 9-10
PAINT_ROW, 9-10
Parameters, *see* Stack(s)
Partners, MAIL_CMD, 17-32
PCR, 10-3
PEMIT, 10-3
PICK, 15-2
PLAYBACK, 11-5, 17-17
Playback, Remote Port
  from capture RAM, 11-5
  from disk recording, 11-5
  scrolling, 11-5
  transferring data, 11-6
POP_USER, 9-6
Port Identifiers, 13-1 *to* 13-3
PORT-ID, 13-1
PRINTER, 10-1
Printer Port
  configuration, 10-1 *to* 10-4
PRINTER_EOL, 10-2
PRINTER_MODE, 10-2
Printing
  32 bit hex number, 10-3
  a screen, 10-3
  a single character, 10-3
  carriage return, 10-3
  character string, 10-3
  numbers, 10-3
PRINT_SCREEN, 10-3
Processor(s)
  FEF DO words, 16-3
  identifiers, 13-2, 16-3
  mailing between, 17-12, 17-31, 17-32
  restarting, B-2
  switching, 13-8, 13-9
PROMPT/END_PROMPT, 17-18
Prompts, 17-18

PRT-L, 10-4

Queue, 12-1
QUIT_TRA, 11-6
Quote Space, 8-1

R, 15-2
>R, 15-2
R-FILEX, 17-1
R>, 15-2
Range Checking, 7-2, 7-5
RCR, 11-4
RCV-TIMEOUT, 11-7
RE, 13-5
READ_BLOCKS, 13-6
Real-Time Clock, 12-5
REBOOT, 13-4
RECEIVE_FILEX, 11-8
RECORD, 17-16
Recording to Disk
  closing, 17-17
  drive selection, 13-4
  filename, 17-16
  opening, 17-16
  playback from remote port, 11-5, 11-6
  trace statements, 9-5
RED_BG, 9-3
RED_FG, 9-3
Remainder, 4-2
REMOTE, 11-1
Remote Command Execution, 16-3
Remote Port, 11-1 *to* 11-8
  carriage return, 11-4
  configuration, 11-1, 11-2
  data playback, 11-5, 11-6
  directory listing, 11-5
  file transfer, 11-7
  screen display, 11-4
  scrolling, 11-5
  sending strings, 11-3, 11-4
  transferring data, 11-6
REMOTE_OUT, 11-3
REMOTE_OUT_W, 11-4
REMOVE, 13-7
RENAME, 13-7
REP_OFF, 17-16
REP_ON, 17-16
Request to Send, *see* Leads, interface
Return Stack, *see* Stack(s)
RETURN_STATE, 17-37
REV_VIDEO, 9-3
REWR, 13-5
Ring Indicate, *see* Leads, interface
RLINE, 11-4
RMT_DIR, 11-5
RMT_DIRL, 11-5
RMT_OFF, 11-5
RMT_ON, 11-5
ROT, 15-1
Row Number, 9-8
RSCREEN, 11-4
RTRACE, 9-4
RUN_SEQ, 17-33
RX_SPEED, 10-1, 11-1

SAVEB, 17-41
SAVETS, 17-41
Saving
  a binary, 17-41
  data to capture RAM, 17-16
  test scripts, 17-41 *to* 17-43
SCRN_BACK, 11-6
SCRN_FWD, 11-6
Scrolling
  line backward, 11-6
  line forward, 11-5
  page backward, 11-6
  page forward, 11-6

INDEX [continued]

Scrolling *[continued]*
  to bottom screen, 11–5
  to top screen, 11–5
SDL Diagrams, 17–3
Search
  anchored match, 8–4
  for a string, 8–3
  unanchored match, 8–3
?"SEARCH, 8–3
SECONDS_ELAPSED, 17–27
SEEK, 13–6
SEND-TIMEOUT, 11–7
SEND_FILEX, 11–8
Sequences, 17–33
SEQ{ }SEQ, 17–33
SET_CURR_TOPIC, 17–23
Shift
  left, 5–2
  right, 5–3
  variable left, 5–3
  variable right, 5–3
SHOW_DATA, 9–1
SHOW_TEST, 9–15
SHOW_USER, 9–6
SHUTDOWN, 13–4
Signal Quality, *see* Leads, interface
STACK, 15–1
Stack(s), 1–1
  computational, 15–1
  copy to return, 15–2
  delete value(s), 15–1
  display contents, 15–1
  duplicate a value, 15–1
  LIFO, 1–1
  notation, 1–2
  parameters, 15–1
  pop from return, 15–2
  push to return, 15–2
  reorder values, 15–1
  return, 15–1
  switch places, 15–1
START_LAPSE_TIMER, 17–26
START_TIMER, 17–26
State(s)
  definition, 17–6
  initialization, 17–6
  machine, 17–3
  messages, 17–17
  symbols, 17–3
  transition, 17–7
STATE_INIT{ }STATE_INIT, 17–6
STATE_OFF, 17–17
STATE_ON, 17–17
STATE{ }STATE, 17–6
Status
  data, 12–2
  file, 13–5
STOP_TIMER, 17–26
Store
  16 bit value, 3–2
  32 bit value, 3–1, 17–28
  8 bit value, 3–3
  plus, 3–2, 4–1, 17–28
  timestamps, 12–5
#>STR, 8–6
STR>#, 8–6
String(s), 8–1 *to* 8–6
  converting to numbers, 8–6
  copying, 8–3
  data type, 2–1, 2–2
  definition, 2–1
  dot quote space, A–3
  in user window, 9–6
  length, 8–2
  moving/copying, 8–2
  printing, 10–3
  remote port, 11–3, 11–4
  searching for, 8–3
  trace statements, 9–2
  user window, 9–7
Subtraction, 4–1
  timestamps, 12–4
SWAP, 15–1
SWITCH, 13–8
System Crash, B–1

T-LIMIT, 11–8
T., 9–2
T.", 9–2
T.H, 9–2
T.TYPE, 9–2
TCLR, 17–6
TCOLOR, 9–3
TCR, 9–3
TEMIT, 9–2
Terminate
  BEGIN/AGAIN, 6–6
  BEGIN/UNTIL, 6–6
  DO/LOOP, 6–5
Test Manager, 17–1 *to* 17–43
  actions, 17–6, 17–15 *to* 17–33
  automatic start, 17–40, 17–41
  comparison, 17–7
  counters, 17–28 *to* 17–30
  event recognition, 17–7 *to* 17–15
  initializing, 17–6
  input, 17–18 *to* 17–24, A–1
  logical operations, 17–7
  memory, 17–2
  output, 17–24 *to* 17–26
  protocol specific actions, 17–33
  sequences, 17–33
  state definition, 17–6
  state display, 17–17
  state initialization, 17–6
  state transition, 17–7
  stopping, 17–7, 17–41
  subroutines, 17–37
  timers, 17–26, 17–27
  wakeup timer, 17–11, 17–40
Test Script Window, 9–14, A–1
Test Script(s)
  automatic start, 17–41
  binary, 17–39, 17–41
  chaining, 17–38
  clearing, 17–6
  constructs, 17–6, 17–33 *to* 17–38
  loading, 17–39
  multiple, 17–2, 17–38
  prompts, 17–18
  running, 17–39, 17–40
  saving, 17–41 *to* 17–43
  stopping, 17–7, 17–41
  structure, 17–5 *to* 17–7
THERE, 9–8
TIMEOUT, 17–10
TIMER, 17–10
?TIMER, 17–11
Timer(s)
  decoding, 17–10
  event recognition, 17–10, 17–11
  lapse, *see* Lapse Timer(s)
  starting/stopping, 17–26
  wakeup, *see* Wakeup Timer
TIMER-NUMBER, 17–10
Timestamp(s), 12–1 *to* 12–5
  addition, 12–4
  beginning/end of frame, 12–2
  comparison, 12–5
  conversion, 12–3
  copying, 12–5
  format, 12–3
  frames, 12–1
  in the data buffer, 12–2
  microseconds, 12–3
  milliseconds, 12–3

INDEX [continued]

Timestamp(s) *[continued]*
  storing, 12–5
  subtraction, 12–4
-TITLE, 17–16
TM.RAM, 17–2
TM_RUN, 17–40
TM_STOP, 17–7, 17–41
TONE, 13–3
TOP, 11–5
Topics, 17–23
TO_DCE_RX, 13–2
TO_DTE_RX, 13–2
Trace Statement(s), 9–1 *to* 9–5
  capturing, 9–4
  carriage return, 9–3
  character string, 9–2
  displaying, 9–4
  displaying numbers, 9–2
  recording to disk, 9–5
  single character, 9–2
  turning on/off, 9–4, 9–5, 17–17
  using color, 9–3
TRANSFER, 11–6
Transfer
  files, 17–1
  from capture RAM, 11–6
TRANSLATE_OFF, 11–7
TRANSLATE_ON, 11–7
True Flag, *see* Comparison
TRUNCATE, 13–6
TSTAMP_ADD, 12–4
TSTAMP_COMP, 12–5
TSTAMP_SUB, 12–4
TURN_OFF, 11–2
TURN_ON, 11–2
TX_CONTROL, 10–2, 11–2
TX_SPEED, 10–1, 11–1

U*, 4–2
U/, 4–2
Unanchored Match, *see* Search
UNLOCK_LOGO, 11–4
User Window, 9–5 *to* 9–14
  accessing, 9–6
  carriage return, 9–7
  character sets, 9–9 *to* 9–14
  character string, 9–7
  clearing text, 9–9
  color, 9–9 *to* 9–14
  creating text in, 9–6, 9–7
  cursor control, 9–7, 9–8
  number in, 9–6
  row/column number, 9–8
  scrolling mode, 9–8
  single character, 9–7
  wraparound mode, 9–8

VARIABLE, 3–4
Variables, Creating, 3–3
Vectored Operation, 16–2

W!, 3–2
W., 9–6
W.", 9–6
W.ERROR, 17–25
W.ERRORS, 17–26
W.H, 9–7
W.NOTICE, 17–24
W.NOTICES, 17–25
W.TYPE, 9–7
W.TYPE_A, 9–7
W@, 3–2
?WAKEUP, 17–11
Wakeup Timer
  event recognition, 17–11
  starting, 17–40
  turning off, 17–40
WAKEUP_OFF, 17–40
WAKEUP_ON, 17–40
WCR, 9–7
WD0–7, 13–4
WDOWN, 9–8
WEMIT, 9–7
WHERE, 9–8
WHI_BG, 9–3
WHI_FG, 9–3
Wildcards
  event recognition, 17–14, 17–15
  match, 8–4
Window(s)
  command, *see* Command Window
  data, *see* Data Window
  error, *see* Error Window
  notice, *see* Notice Window
  test script, *see* Test Script Window, User Window
  user, *see* User Window
Word
  fetch, 3–2
  store, 3–2
WR, 13–5
WRAP, 9–8
WRITE_BLOCKS, 13–6
WSCROLL, 9–8
WUP, 9–8

X Quote Space, 8–2
XMODEM, 17–1
XOR, 5–2, 17–30

YEL_BG, 9–3
YEL_FG, 9–3

Zero Divide, *see* Error(s)