


4041 SYSTEM CONTROLLER

*Please Check for
CHANGE INFORMATION
at the Rear of this Manual*

Copyright ©1982 , 1983 Tektronix, Inc. All rights reserved.
Contents of this publication may not be reproduced in any
form without the written permission of Tektronix, Inc.

Products of Tektronix, Inc. and its subsidiaries are covered
by U.S. and foreign patents and/or pending patents.

TEKTRONIX, TEK, SCOPE-MOBILE, and  are
registered trademarks of Tektronix, Inc. TELEQUIPMENT
is a registered trademark of Tektronix U.K. Limited.

Printed in U.S.A. Specification and price change privileges
are reserved.

INSTRUMENT SERIAL NUMBERS

Each instrument has a serial number on a panel insert, tag,
or stamped on the chassis. The first number or letter
designates the country of manufacture. The last five digits
of the serial number are assigned sequentially and are
unique to each instrument. Those manufactured in the
United States have six unique digits. The country of
manufacture is identified as follows:

| | |
|---------|---|
| B000000 | Tektronix, Inc., Beaverton, Oregon, USA |
| 100000 | Tektronix Guernsey, Ltd., Channel Islands |
| 200000 | Tektronix United Kingdom, Ltd., London |
| 300000 | Sony/Tektronix, Japan |
| 700000 | Tektronix Holland, NV, Heerenveen, The Netherlands |

MANUAL REVISION STATUS

PRODUCT: 4041 System Controller

This manual supports the following versions of this product: Firmware Version 1, Level 1 and above.

| REV DATE | DESCRIPTION |
|----------|---|
| JUN 1982 | Original Issue; replaces 061 - 2546 - 00. |
| JUN 1983 | Rewritten; replaces 070-3917-00. |



MANUAL CHANGE INFORMATION

Date: 12-4-85

Change Reference: M59185

Product: 4041 Programmer's Reference Manual

Manual Part No.: 070-3917-01

DESCRIPTION

PG 76

This manual insert describes "UTL2", a set of array-handling romcalls included in standard 4041s with serial numbers above B070100. Included in this insert is a new 8-page section (Section 16) which describes the romcalls, and one new page to be added at the end of Appendix A, which lists the numeric error codes associated with UTL2.

CONTENTS

| Section 1 | INTRODUCTION TO 4041 BASIC | Page |
|------------------|--|-------------|
| | About the 4041 System Controller | 1-1 |
| | 4041 BASIC | 1-2 |
| | 4041 Program Segment Structure | 1-3 |
| | How to Use This Manual | 1-6 |
| Section 2 | ELEMENTS OF 4041 BASIC | |
| | Introduction | 2-1 |
| | Real Numbers | 2-1 |
| | String Constants | 2-4 |
| | Numeric Variables | 2-5 |
| | String Variables | 2-6 |
| | Operators | 2-7 |
| | Expressions | 2-12 |
| | Functions | 2-14 |
| | Statements | 2-20 |
| | Reserved Keywords | 2-21 |
| Section 3 | FRONT PANEL, KEYBOARD, AND OPTION | |
| | Introduction | 3-1 |
| | Front Panel | 3-1 |
| | Program Development (P/D) Keyboard | 3-5 |
| | Instrument Options | 3-12 |
| Section 4 | EDITING, DEBUGGING, AND DOCUMENTATION | |
| | Introduction | 4-1 |
| | Reporting Errors | 4-1 |
| | Reporting Debugging Information | 4-1 |
| | The BREAK Statement | 4-2 |
| | The CONNECT Statement | 4-4 |
| | The DEBUG Statement | 4-5 |
| | The DELETE LINE Statement | 4-6 |
| | The LIST Statement | 4-7 |
| | The NOBREAK Statement | 4-8 |
| | The NOTRACE Statement | 4-9 |
| | The REM Statement | 4-11 |
| | The RENUMBER Statement | 4-12 |
| | The SET DEBUG Statement | 4-15 |
| | The SET SYNTAX Statement | 4-16 |
| | The SLIST Statement | 4-17 |
| | The TRACE Statement | 4-18 |

Section 5 ENVIRONMENTAL CONTROL

Introduction 5-1
The ASK and ASK\$ Functions 5-2
 ASK Functions:
 ANGLE 5-3
 AUTOLOAD 5-3
 BUFFER 5-4
 CHPOS 5-5
 IODONE 5-6
 KEY 5-7
 MEMORY 5-7
 PROCEED 5-8
 SEGMENT 5-8
 SPACE 5-9
 TIME 5-9
 UPCASE 5-9
 ASK\$ Functions:
 CONSOLE 5-10
 DRIVER 5-11
 ERROR 5-11
 ID 5-12
 LU 5-12
 PATH 5-13
 ROMPACK 5-14
 SELECT 5-14
 SELFTEST 5-14
 SYSDEV 5-15
 TIME 5-15
 VAR 5-16
 VOLUME 5-16
The INIT Statement 5-17
The SET Statement 5-18
 ANGLE 5-19
 AUTOLOAD 5-19
 CONSOLE 5-20
 DEBUG 5-20
 DRIVER 5-21
 FUZZ 5-22
 PROCEED 5-24
 SYNTAX 5-24
 SYSDEV 5-25
 TIME 5-26
 UPCASE 5-27

Section 6 MEMORY MANAGEMENT

Introduction 6-1
The COMPRESS Statement 6-2
The DELETE ALL Statement 6-2
The DELETE VAR Statement 6-3
The DIM Statement 6-4
The INTEGER Statement 6-5
The LET ("Assignment") Statement 6-6
The LONG Statement 6-7

| | | |
|------------------|---------------------------------------|------|
| Section 7 | CONTROL STATEMENTS | |
| | Introduction | 7-1 |
| | The CALL Statement | 7-2 |
| | The CONTINUE Statement | 7-3 |
| | The END Statement | 7-4 |
| | The EXIT Statement | 7-5 |
| | The FOR and NEXT Statements | 7-7 |
| | The GO SUB and GO TO Statements | 7-9 |
| | The IF Statement | 7-11 |
| | Invoking User-Defined Functions | 7-12 |
| | The RCALL Statement | 7-13 |
| | The RETURN Statement | 7-15 |
| | The RUN Statement | 7-16 |
| | The STOP Statement | 7-17 |
| | | |
| Section 8 | INPUT/OUTPUT | |
| | Introduction | 8-1 |
| | I/O Drivers | 8-2 |
| | The System Console Device | 8-2 |
| | Specifying Data Paths | 8-4 |
| | Proceed-Mode I/O | 8-5 |
| | The CLOSE Statement | 8-7 |
| | The COPY Statement | 8-8 |
| | The DATA Statement | 8-9 |
| | The GETMEM Statement | 8-10 |
| | The IMAGE Statement | 8-11 |
| | The INPUT Statement | 8-12 |
| | The # Clause | 8-15 |
| | The ALTER Clause | 8-16 |
| | The BUFFER Clause | 8-16 |
| | The DELN Clause | 8-17 |
| | The DELS Clause | 8-17 |
| | The INDEX Clause | 8-18 |
| | The PROMPT Clause | 8-18 |
| | The USING Clause | 8-19 |
| | The OPEN Statement | 8-28 |
| | The PRINT Statement | 8-30 |
| | The # Clause | 8-33 |
| | The BUFFER Clause | 8-33 |
| | The INDEX Clause | 8-34 |
| | The USING Clause | 8-34 |
| | The PUTMEM Statement | 8-43 |
| | The RBYTE Statement | 8-44 |
| | The READ Statement | 8-47 |
| | The RESTORE Statement | 8-48 |
| | The SELECT Statement | 8-49 |
| | The WBYTE Statement | 8-50 |

| | | |
|-------------------|---|-------|
| Section 9 | INSTRUMENT CONTROL WITH GPIB | |
| | Introduction | 9-1 |
| | Setting the GPIB's Physical Driver Parameters | 9-1 |
| | The SET DRIVER Statement (GPIB) | 9-2 |
| | Setting Up Logical Units | 9-4 |
| | The OPEN Statement (GPIB) | 9-4 |
| | The ASK\$("LU") Function (GPIB) | 9-5 |
| | High-Level Data Transfers | 9-6 |
| | The INPUT Statement (GPIB) | 9-6 |
| | The PRINT Statement (GPIB) | 9-8 |
| | Low-Level Data Transfers and Instrument Control | 9-10 |
| | The RBYTE Statement (GPIB) | 9-10 |
| | The WBYTE Statement (GPIB) | 9-12 |
| | Serial Polls | 9-19 |
| | The POLL Statement | 9-19 |
| | Parallel Polls | 9-22 |
| | The ATN(EOI) Function | 9-22 |
| | | |
| Section 10 | RS-232-C DATA COMMUNICATIONS | |
| | Introduction | 10-1 |
| | Parity | 10-2 |
| | Stop Bits | 10-2 |
| | Typeahead Buffer | 10-2 |
| | Flagging | 10-2 |
| | The EDI Parameter | 10-3 |
| | Control Characters | 10-3 |
| | Control Character Functions | 10-4 |
| | Line Editing | 10-4 |
| | Control Character Notes | 10-6 |
| | The "Meta" Character: CTRL-^ | 10-7 |
| | ITEM Format | 10-7 |
| | | |
| Section 11 | SUBPROGRAMS AND USER-DEFINED FUNCTIONS | |
| | Introduction | 11-1 |
| | The CALL Statement | 11-7 |
| | The FUNCTION Statement | 11-8 |
| | The SUB Statement | 11-10 |

| | | |
|-------------------|---|-------|
| Section 12 | INTERRUPT HANDLING | |
| | Introduction | 12-1 |
| | ABORT | 12-6 |
| | ERRORS | 12-8 |
| | GPIB Conditions | 12-12 |
| | IODONE | 12-14 |
| | SRQ INTERRUPTS (OPT2 DRIVER) | 12-15 |
| | USER-DEFINABLE FUNCTION KEYS | 12-17 |
| | The ADVANCE Statement | 12-19 |
| | The BRANCH Statement | 12-20 |
| | The DISABLE Statement | 12-21 |
| | The ENABLE Statement | 12-22 |
| | The MONITOR Statement | 12-24 |
| | The OFF Statement | 12-25 |
| | The ON Statement | 12-26 |
| | The RESUME Statement | 12-28 |
| | The RETRY Statement | 12-29 |
| | The TRAP Statement | 12-30 |
| | The WAIT Statement | 12-31 |
| | | |
| Section 13 | PROGRAM MANAGEMENT | |
| | Introduction | 13-1 |
| | The APPEND Statement | 13-2 |
| | The LOAD Statement | 13-4 |
| | The SAVE Statement | 13-5 |
| | | |
| Section 14 | DC-100 TAPE | |
| | Introduction | 14-1 |
| | The DELETE FILE Statement | 14-2 |
| | The DIR Statement | 14-3 |
| | The DISMOUNT Statement | 14-4 |
| | The EOF Function | 14-5 |
| | The FORMAT Statement | 14-6 |
| | The INPUT Statement (TAPE) | 14-7 |
| | The OPEN Statement (TAPE) | 14-8 |
| | The PRINT Statement (TAPE) | 14-9 |
| | The RBYTE Statement (TAPE) | 14-12 |
| | The RENAME Statement | 14-13 |
| | The TYPE Function | 14-14 |
| | The WBYTE Statement (TAPE) | 14-15 |
| | | |
| Section 15 | SCSI (OPTION 03) SUB-SYSTEM | |
| | Introduction | 15-1 |
| | ASCII VS Item Files | 15-1 |
| | Logical VS Physical Disk I/O | 15-1 |
| | Wild Cards and Other Special Characters | 15-1 |
| | Examples | 15-2 |
| | Stream Specification Settings for SCSI | |
| | Disk Interface | 15-9 |

| | | |
|-------------------|---|------|
| Appendix A | ERROR MESSAGES | |
| Appendix B | ASCII (GPIB) CODE CHART | |
| Appendix C | GLOSSARY | |
| Appendix D | STREAM SPECIFICATIONS | |
| | Form of a Stream Specification | D-1 |
| | Parameters | D-2 |
| | Logical Units | D-5 |
| Appendix E | INTRODUCTION TO GPIB CONCEPTS | |
| | Mechanical Elements of IEEE-488 Standard | E-1 |
| | Electrical Elements | E-3 |
| | Functional Elements: the 10 Interface Functions..... | E-3 |
| | Addresses: Primary, Talk, Listen, and Secondary | E-4 |
| | Data, Management, and "Handshake" Buses | E-6 |
| | GPIB Communications Protocol: | |
| | Talkers, Listeners, and Controllers | E-8 |
| | Universal Commands | E-10 |
| | Addressed Commands | E-11 |
| | Serial Polling | E-12 |
| | Parallel Polling | E-13 |
| Appendix F | INTRODUCTION TO TEKTRONIX CODES AND FORMATS | |
| | Introduction | F-1 |
| | Compatibility | F-2 |
| | Human Interface..... | F-4 |
| | Representing Numbers | F-4 |
| | Device-Dependent Message Structure | F-5 |
| | Message Conventions..... | F-7 |
| | Status Bytes | F-8 |
| | Queries..... | F-9 |
| | Additional Features | F-10 |

INDEX

ILLUSTRATIONS

| Figure | Description | Page |
|--------|---|------|
| 1-1 | The Tektronix 4041 Controller | ix |
| 2-1 | "Row-Major" Mapping in Assignment Statements | 2-11 |
| 3-1 | 4041 Front Panel | 3-2 |
| 3-2 | 4041 Program Development (P/D) Keyboard | 3-5 |
| 10-1 | Control Character Functions | 10-4 |
| E-1 | Allowable Configurations for GPIB Devices | E-2 |
| E-2 | "Typical" Binary Switch System, with Address Set to 19 | E-4 |
| E-3 | Data, Management, and "Handshake" buses | E-6 |
| F-1 | Three Valid Data Representations for the Same Data on the GPIB. Tektronix Codes and Formats Standardizes on ASCII Code with the Most Significant Byte First | F-3 |
| F-2 | Problem: Talker Delimits on <CR> <LF>; Controller Delimits on <CR> | F-7 |
| F-3 | Problem: Sending Binary Data, Using <CR> <LF> as Terminator | F-7 |
| F-4 | Query Commands are Formed by Adding a Question Mark to the Mnemonic for the Setting to be Queried | F-9 |
| F-5 | Devices Should Assert SRQ When an Illegal Command is Received | F-10 |

TABLES

| Table | Description | Page |
|-------|---|------|
| 3-1 | Instrument Options | 3-12 |
| 4-1 | Table of Defaults for RENUMBER | 4-12 |
| 8-1 | Input Using Operators | 8-19 |
| 8-2 | Input Using Modifiers | 8-19 |
| 8-3 | Print Using Operators | 8-35 |
| 8-4 | Print Using Modifiers | 8-35 |
| 10-1 | Control Character Functions | 10-5 |
| 12-1 | 4041 Interrupts | 12-5 |
| D-1 | Physical Parameters for COMM Driver | D-6 |
| D-2 | Logical Parameters for COMM Driver | D-7 |
| D-3 | ASK\$ Parameters for COMM Driver | D-8 |
| D-4 | Logical Parameters for FRTP Driver | D-8 |
| D-5 | Physical Parameters for GPIB Driver | D-8 |
| D-6 | Logical Parameters for GPIB Driver | D-9 |
| D-7 | ASK\$ Parameters for GPIB Driver | D-10 |
| D-8 | Physical Parameters for OPT2 Driver | D-11 |
| D-9 | ASK\$ Parameters for OPT 2 Driver | D-11 |
| D-10 | Logical Parameters for PRIN Driver | D-11 |
| D-11 | Logical Parameters for TAPE Driver | D-11 |
| D-12 | Logical Parameters for TAPE Files | D-12 |
| F-1 | Number Formats (ANSI x 3.42) | F-4 |
| F-2 | Non-Character Arguments | F-6 |
| F-3 | Status Byte Definitions | F-8 |



Figure 1-1. The 4041 System Controller.

Section 1

INTRODUCTION TO 4041 BASIC

ABOUT THE 4041 SYSTEM CONTROLLER

The Tektronix 4041 System Controller combines the power and versatility of a desktop computer with enhanced programmability to control GPIB-compatible and RS-232-C compatible instruments.

The GPIB, or General Purpose Interface Bus, is the name given to the protocol that governs data communication between devices that implement IEEE-488, "IEEE Standard Digital Interface for Programmable Instrumentation" (ANSI/IEEE-488-1978, Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, N.Y. 10017).

GPIB interfacing allows the 4041 to communicate with and control any of hundreds of GPIB-compatible instruments available today, including multimeters, counters, waveform generators, power supplies, data loggers, oscilloscopes, spectrum analyzers, discs, tapes, and other instruments and computer peripherals.

The RS-232-C standard (published by the Electronic Industries Association, 2001 I Street, N.W., Washington, D.C. 20006) applies to data communication between data terminal equipment and data communications equipment employing serial binary data interchange.

With its RS-232-C interface capability, the 4041 can communicate with a variety of data communications equipment, including terminals, printers, plotters, and other computers.

Operating the standard 4041 involves connecting instruments to it, loading a tape into the built-in DC-100 magnetic tape drive, and pressing the AUTOLOAD key.

From this point, the 4041 is capable of operation with any desired degree of supervision, from unattended program execution to interactive operations involving the 4041's front-panel keys, the Program-Development (P/D) keyboard (which plugs into a jack in the 4041's front panel), or peripheral devices, such as a computer terminal attached through an RS-232-C interface port.

While running interactive programs, operators can invoke program functions by pressing one of the ten user-definable function keys on the front panel or P/D keyboard.

The user can also enter numeric values by means of the front-panel keys when the 4041 requests numeric input. Numeric or alphabetic data can also be input by means of the Program Development (P/D) keyboard or computer terminal.

Programs are written on a 4041 equipped with Option 30, Program Development ROMs. Programmers communicate with the 4041 by using the P/D keyboard or a computer terminal.

4041 BASIC

The 4041 uses a version of the BASIC programming language optimized for ease of programming and GPIB device control. 4041 BASIC includes the following features:

8-Character Variable Names

Variables in 4041 BASIC can be given variable names up to eight characters long.

Optional Line Labels

Each line in a 4041 BASIC program can be labeled, and the line can be referred to by that label. This feature improves readability and greatly eases programming effort, by relieving the programmer of the need to remember line numbers. Labels can be chosen to indicate the function that a section of code performs — for instance, the first line of a sorting routine might be labeled SORT, and a line to transfer control to that routine would be written GO TO SORT.

GPIB Support

The 4041 can control any IEEE-488 compatible device in all legal states. 4041 BASIC contains both high- and low-level constructs for communicating with GPIB devices. It also includes several special functions to enhance readability of statements that send commands over the GPIB.

Tek Codes and Formats Support

The 4041 is designed to be especially easy to operate with instruments that support the Tektronix Codes and Formats standard. This standard defines device-dependent message formats and codings to reduce the cost and time required to develop software for controlling GPIB devices.

Subprograms and User-Defined Functions

Subprograms and user-defined functions can be defined with local environments, local as well as global variables, and formal parameter passing. This feature makes it simple to construct modular programs that are easy to read and maintain.

Interrupt-Handling

The user can define handling routines to allow the 4041 to recognize any interrupt and recover from it under program control. The user can also define “local” interrupt-handlers, so that the action taken to recover from an interrupt can vary depending on where the interrupt occurs.

Efficient Memory Management

For efficient use of memory, variables can be stored in integer, “regular” floating point, or long-floating-point (double precision) form.

Engineering Notation

For user convenience, numbers can be entered using abbreviations for such common prefixes as “micro-”, “milli-”, and “kilo-”. Using this notation, for example, the number .0032 could be entered as 3.2M.

Binary/Octal/Hexadecimal Numbers

The user can enter binary, octal, or hexadecimal numbers by enclosing the number in brackets and adding a “B”, “O”, or “H” suffix. For example, the hexadecimal number B97E would be entered as [B97EH].

Debugging Aids

Among the debugging aids included in the 4041 are commands to:

- Trace each change in the value of any user-specified variable;
- Trace program flow;
- Set “breakpoints” to stop the program at any point and allow the programmer to examine contents of variables.

4041 PROGRAM SEGMENT STRUCTURE

A 4041 program consists of one or more segments: a "main program" segment followed optionally by a sequence of subprogram segments.

Each line of the program is numbered, and lines are executed in ascending order, from lowest to highest.

The main program segment consists of that part of the program from the first (lowest-numbered) line to the first END statement, inclusive. The main program segment is usually used to perform "housekeeping" (see explanation below) and to specify the order in which subprograms are called during execution.

Subprogram segments begin with a SUB or FUNCTION statement and end with END statements.

A SUB statement designates the first line of a subprogram or user-defined interrupt handler.

A FUNCTION statement designates the first line of a user-defined function.

THE MAIN PROGRAM SEGMENT

Entire 4041 programs can be written in one main program segment. For complex applications, however, the program is usually divided into a main program segment and several subprogram segments.

The main segment of a well-constructed program usually does two things:

1. It performs "housekeeping" chores the program requires, and
2. It specifies the order in which subprograms are executed.

Included among the "housekeeping" chores usually performed in the main segment at the start of execution are:

1. Naming the system console device;
2. Setting physical parameters for the GPIB and RS-232-C interface ports;
3. Opening logical units for I/O operations.

Naming the System Console Device

The system console device is the driver through which the user communicates with the 4041. It is the device through which the user enters commands for the 4041, and also is the default device for INPUT, PRINT, and other operations where the 4041 must output messages for the user.

Usually, the system console device will be either the front panel and P/D keyboard or a computer terminal connected to the 4041 through an RS-232-C interface port.

When the 4041 powers up, the front panel and P/D keyboard are automatically made the system console device.

In order to change the system console device to a computer terminal connected to an RS-232-C interface port, the command

```
Set console "comm:"
```

must be executed.

If the user has no P/D keyboard and wishes to name the RS-232-C interface port as system console device, he/she should insert a tape containing a file named "AUTOLD" into the DC-100 tape drive and press the "AUTOLOAD" key on the front panel. Pressing this key automatically loads and runs the "AUTOLD" file. This file should contain a command designating the RS-232-C interface port as system console.

Once execution begins, the SET CONSOLE statement is used to change the system console device. (See the description of the SET CONSOLE statement in Section 10, "RS-232-C Data Communications", for more information.)

Setting GPIB and RS-232-C Physical Parameters

Because the GPIB and RS-232-C interface ports are used to communicate with other devices, several parameters governing their operation are typically set once, at the beginning of program execution.

For the GPIB interface ports, for example, these parameters include (among others) the 4041's primary address on the bus, and a status parameter indicating whether or not the 4041 is system controller.

For the RS-232-C interface ports, these parameters include such items as baud rate, number of stop bits, number of bits per character, parity, etc.

The values of these parameters are set by means of the SET DRIVER statement. (See the descriptions of the SET DRIVER statement in Section 9, "Instrument Control With GPIB", and in Section 10, "RS-232-C Data Communications", for more information.)

Opening Logical Units

INPUT and PRINT operations use the system console device as a default. For example, if the front panel and P/D keyboard are the system console device, then upon execution of the command

```
1000 Input number
```

the 4041 expects the value of a numeric variable called Number to be input via the system console device.

Similarly, upon execution of the command

```
1010 Print number
```

the 4041 displays the value of the numeric variable Number on the system console device.

INPUTs and PRINTs to devices other than the system console are primarily done by means of logical units. A logical unit is associated with a device for input or output by means of the OPEN statement.

For example, if the user wishes certain output to go to the thermal printer, then he/she might open logical unit 1 and associate it with the printer, as follows:

```
1500 Open #1:"prin:"
```

Then, the command

```
2000 Print #1:a,b,c
```

would print the values of numeric variables A, B, and C on the thermal printer.

In general, logical units are OPENed in the main program segment and used throughout the remainder of the program. Logical units may, however, be opened and closed at any time.

For more information about opening and closing logical units, see the descriptions of the OPEN statement in Section 8, "Input/Output", and in Section 9, "Instrument Control With GPIB". See also the description of the CLOSE statement in Section 8.

FLOW OF CONTROL BETWEEN PROGRAM SEGMENTS

A 4041 BASIC program executes sequentially, from the lowest line number to the highest, or until an END statement is encountered (whichever comes first).

When the 4041 is executing statements in one program segment, the GO TO and GO SUB commands can only transfer control to other program lines IN THE SAME SEGMENT.

Control is transferred to other segments under the following conditions:

1. Control is transferred to a subprogram by means of the CALL statement.

Example:

```
1575 Call snarf(a,b,c)
```

This command transfers control to a subprogram called Snarf, passing to it the parameters A, B, and C.

2. Control is transferred to a user-defined function by invoking the function's name within another statement.

Example:

```
1600 Zip=sumsquar(x,y)+2
```

This command transfers control to the user-defined function SumSquar, passing to it the parameters X and Y. When control returns from the function, the number 2 is added to the returned value. The result is stored in numeric variable Zip.

3. Control is transferred to a user-defined interrupt handler when the condition it handles is sensed. In order for an interrupt condition to be recognized and control passed to a handler for that condition, two things must be true: (1) the handler for the interrupt condition must be defined; and (2) the 4041 must be enabled to recognize the condition.

Interrupt conditions on which control can be transferred to a handler include ABORT, ERROR, IODONE, KEYS, and several GPIB-dependent interrupt conditions. (See Section 12, "Interrupt Handling", for more information about interrupt conditions.)

Handlers for conditions are defined by the ON statement. Conditions are enabled by the ENABLE statement.

Handlers for ABORT and ERROR conditions are automatically defined and enabled at power-up. The ABORT condition may be disabled and an alternate handler defined for it. Errors may also have alternate handlers defined for them, but may never be disabled. For more information, see Section 12, "Interrupt Handling".

Example:

```
300 On srq then call srqhand
310 enable srq
.
.
1000 sub srqhand ! handler for srq interrupt
```

After the first two statements are executed, control will be transferred to a subprogram called "SRQHand" whenever the 4041 senses the SRQ line being asserted on the standard GPIB interface port, as long as an OFF SRQ or DISABLE SRQ command has not been executed after statement 310.

Interrupt handlers are special cases of subprograms. For information about them, see Section 12, "Interrupt Handling".

4. No program segment can call or invoke itself or any other segment in the "active call sequence".

The "active call sequence" consists of those program segments currently executing. (The main program segment is always in the active call sequence.)

For example, if a program consists of a main segment and subprograms A, B, and C, and the main program calls A, which calls B, which in turn calls C, then C cannot call A, B, or itself — all are in the active call sequence.

HOW TO USE THIS MANUAL

GUIDE TO NOTATION FOR SYNTAX AND DESCRIPTIVE FORMS

The following conventions apply for the syntax and descriptive forms of commands shown in this manual:

1. Items enclosed in square brackets are optional.

Example:

Syntax Form: RUN [numexp]

The square brackets indicate that the numeric expression following the RUN keyword is optional. Any of the following forms of the RUN statement, therefore, are syntactically correct:

```

RUN
RUN 1000
RUN start+ 50
    
```

2. Optional entries within optional entries cannot be used by themselves.

Example:

Syntax Form:

```

[line-no.] LIST [strexpr] [numexp [TO numexp] ]
                [subname[TO subname]]
    
```

The following forms of the LIST statement are syntactically correct:

```

LIST
1050 LIST 10 TO 1000
2000 LIST A TO B
    
```

The following form, however, is incorrect:

```

LIST TO 2000
    
```

3. Stacked items enclosed in BRACES make up a selection list from which one item **MUST** be selected.

Example:

```

Syntax Form: [line-no.] ENABLE {DCL[(numexp)]}
                                {EOI[(numexp)]}
                                {IFC[(numexp)]}
                                {MLA[(numexp)]}
                                {MTA[(numexp)]}
                                {SRQ[(numexp)]}
                                {TCT[(numexp)]}
    
```

Any of the following forms of the ENABLE statement may be entered:

```

ENABLE SRQ
1000 ENABLE TCT(1)
1500 ENABLE IFC(5)
    
```

The following cannot be entered:

```

2000 ENABLE
    
```

4. Stacked items enclosed in SQUARE BRACKETS make up a selection list from which zero or one item **MAY** be selected.

Example:

```

Syntax Form: [line-no.] INIT [VAR var[,var]. . . ]
                                [ALL]
                                [SELFTTEST]
    
```

Any of the following forms of the INIT statement are syntactically correct:

```

INIT
INIT VAR a,b,c,d
INIT ALL
INIT SELFTTEST
1000 INIT ALL
    
```

5. Three dots (...) indicate that the preceding item may be repeated as often as required.
6. Keywords and command delimiters should be in the order and position shown. (Keywords and delimiters are printed in upper case and boldface in syntax and descriptive forms.)

Keywords longer than four characters may be shortened to four-character length when entered.
7. A space must precede and follow each keyword.
8. Line numbers must be positive integers in the range from 1 to 65535.
9. The construct "line-no.", when it appears before a keyword in a syntax or descriptive form, denotes a line number optionally followed by a line label and colon.

Abbreviations

The following abbreviations are commonly used in the syntax and descriptive forms of the commands described in this manual. See the Glossary (Appendix C) for definitions of these terms.

| | |
|----------|---|
| exp | expression (either numeric or string) |
| line-no. | line number (optionally followed by a line label and colon) |
| numarray | numeric array variable |
| numexp | numeric expression |
| numvar | numeric scalar variable |
| strarray | string array variable |
| strexpr | string expression |
| strvar | string scalar variable |
| subname | subprogram or user-defined function name |

Section 2

ELEMENTS OF 4041 BASIC

INTRODUCTION

This section defines the fundamental elements and concepts used in 4041 BASIC. Terms discussed in this section are used extensively throughout the rest of this manual.

When a program is being entered, the 4041 treats upper and lower case letters identically. The first letter following the statement number in each line is always displayed in upper case, with remaining characters in lower case, except for letters in quoted strings or remarks, which are displayed in the form in which they were entered.

Some devices (e.g., the front panel, some computer terminals) are incapable of displaying lower case letters in lower case. Most such devices display lower case letters in upper case. Output or listings sent to such devices will always appear in upper case, but may appear in lower case on other output devices (depending on the case of the characters actually sent by the transmitting device).

REAL NUMBERS

INTERNAL NUMBER STORAGE

The 4041 stores numbers internally in one of three formats: (1) as integer numbers; (2) as short floating point numbers; or (3) as long floating point numbers.

Integer numbers take less storage space and are processed more quickly than short or long floating point numbers; long floating point numbers allow the programmer greater precision than integer or short floating point numbers, but take up more internal space and take longer to process.

Integers

Integers take up two bytes of memory. Integers may range from -32768 to $+32767$; attempting to assign values outside this range to an integer variable results in an overflow error.

NOTE

Although -32768 is treated as an integer value, it is stored as a floating-point value. It therefore appears in listings as " -32768.0 ".

Short Floating Point Numbers

Short floating point numbers take up four bytes of memory.

The allowable range for short floating point numbers is: maximum $+/-3.40282E+38$; minimum $+/-2.93874E-39$.

Attempting to assign values outside this range to a floating point variable results in an error.

Entering a number with a decimal point automatically stores the number internally as a short-floating point number, even if the number is in the integer range. Thus, entering the number "38" stores the value 38 in integer representation internally, while entering the number "38." stores 38 in short-floating-point representation internally.

Long Floating Point Numbers

Long floating point numbers take up eight bytes of memory.

The allowable range for long floating point numbers is: maximum $+/-1.7976931348623E+308$; minimum $+/-5.562684646269E-309$.

Attempting to assign a value outside this range to a long floating point variable results in an error.

Entering seven or more digits to represent a number stores the number internally in long-floating point representation. Thus, entering "1.234" stores the value 1.234 in short-floating-point representation, while entering "1.234000" stores the value 1.234 in long-floating-point representation.

Evaluating Expressions

When evaluating an expression, if the two operands of a dyadic operator are of the same type, the result of the operation must be in the range allowed by that type or an error is generated.

Example:

```
*print 24*3600
*** ERROR # 89
```

This example causes an error because the result of the multiplication operation is not within the range allowed by the type of the two operands (integer; -32768 to $+32767$).

Mixing Number Types Within an Operation

Whenever "mixed-type" operations are performed, the number of the "less accurate" type (where accuracy increases in the following order: integer, short floating point, long floating point) is converted to the more accurate type before the operation is performed.

Example:

```
*print 24.*3600
86400.0
```

Since one of the operands is entered in short-floating-point representation, the answer is calculated and displayed in short-floating-point representation, and the allowable range of the answer is the short-floating-point range.

EXTERNAL NUMBER REPRESENTATION

While numbers are always stored in the 4041 as integer, short floating point, or long floating point numbers, numbers can be input to or output from the 4041 in several formats. These formats can be grouped as follows: standard positional notation; scientific notation; engineering notation; binary/octal/hexadecimal notation.

Standard Positional Notation

Numbers represented in standard positional notation are numbers written in the way we are used to reading them.

Examples are the numbers 5, 98.6, -0.043, and 32000.0.

Imbedded spaces and commas are not allowed within numbers in standard positional representation (nor, for that matter, in any others).

Scientific Notation

Numbers represented in scientific notation have a base part called the mantissa and a power-of-ten part called the exponent. The exponent specifies the power of ten to which the base part is to be raised.

For example, the number $3.28E+6$ is a number written in scientific notation; 3.28 is the mantissa and 6 is the exponent. The number $3.28E+6$ means "3.28 times ten to the sixth power", or 3280000.

The mantissa and the exponent are separated by the letter "E" or "e". The mantissa may be a signed or unsigned integer or floating point number. (If no sign is specified, + is used.)

The exponent, however, must be a signed or unsigned integer. (If no sign is specified, + is used.) Attempting to input a number in scientific notation with an exponent that contains a decimal point results in an error.

Engineering Notation

Any number within the 4041's range can be written within a program in engineering notation. This notation gives the programmer a convenient, shorthand way to write numbers with such common prefixes as "kilo-" ($1E3$), "milli-" ($1E-3$), "micro-" ($1E-6$), "nano-" ($1E-9$), and "pico-" ($1E-12$).

Numbers are expressed in engineering notation as integer or floating point numbers followed by one of the letters K or k (for kilo-), M or m (for milli-), U or u (for micro-), N or n (for nano-), or P or p (for pico-).

Examples:

| Number | Engineering Constant Representation |
|-----------------------------|-------------------------------------|
| 3280 | 3.28K |
| .003 ($3.0E-3$) | 3M |
| .0000065 ($6.5E-6$) | 6.5U |
| .000000000009 ($9.0E-12$) | 9P |

NOTE

The letter "M", when used in as a suffix in engineering notation, signifies "milli-" ($10E-3$) whether the letter appears in upper or lower case. There is no engineering notation symbol for "mega-" ($10E6$).

NUMERIC VARIABLES

Numeric variables may be of three types: integer, short floating point, and long floating point. Unless declared otherwise, all variables are assumed to be short floating point variables when they are introduced.

Short floating point variables take up four bytes of memory; integer variables take up only two bytes; long floating point variables take up eight bytes of memory. Integer variables are processed more quickly than short floating point variables; long floating point variables take longer to process than either integer or short floating point variables.

Integer variables are declared in an **INTEGER** statement; long floating point variables are declared in a **LONG** statement. (See Section 6, "Memory Management", for more information on these statements.)

VARIABLE NAMES

Variable names can have at most eight characters in 4041 BASIC. The name must begin with a letter, and may be followed by a combination of letters, numerals, or imbedded underscores. (Leading or trailing underscores, special characters, and imbedded spaces are not allowed.)

The only other restriction on variable names is that the name may not be a reserved keyword (see list of reserved keywords at the end of this section).

Some examples of legal variable names are:

```
A
Length
Portland
Ch_Size
```

Some examples of illegal variable names are:

| | |
|---------------|---|
| PayrollNumber | (Too long) |
| Len_ | (Trailing underscore) |
| Hen%ry | (% is not legal character for variable names) |
| 3num | (First character must be a letter) |
| Goto | (Reserved keyword) |

NUMERIC SCALAR AND NUMERIC ARRAY VARIABLES

A scalar variable has only one value associated with its variable name. Scalar variables are declared implicitly through their appearance in a program. For example, the program statement

```
*A=5
```

declares the variable *A* to be a short-floating-point scalar variable, reserves space for it in memory, and stores the value 5 in that space.

An array variable, on the other hand, represents not one value but a set of values. The array can have either one or two dimensions. An array's dimensions and number of elements are declared by a **DIM** statement or an **INTEGER** or **LONG** statement (for an integer or long-floating-point array). Thus, the statement

```
*100 Dim a(20)
```

declares *A* to be the array variable for a 20-element, one-dimensional array and reserves space for the array, while the statement

```
*100 Long num(3,10)
```

reserves memory space for a 30-element, two-dimensional array *Num* of long-floating-point numbers. The 30 elements of *Num* are ordered into three "rows" and ten "columns". (See descriptions of the **DIM**, **INTEGER**, and **LONG** statements in Section 6, "Memory Management", for more information.)

Individual elements of an array are addressed by a subscript assigned to each element. Subscripts have an implied lower bound of 1, and an upper bound given by the **DIM**, **INTEGER**, or **LONG** statement that declared the variable to be an array. Thus,

```
A(10)
```

signifies the 10th element in the one-dimensional array "A", while

```
Num(2,15)
```

signifies the element in the 2nd row and 15th column of the two-dimensional array "Num".

STRING VARIABLES

The maximum allowable array subscript is 32,767.

Subscripts may be any numeric expression. For example,

Area(I+ J)

signifies the "I+ J"th element of a one-dimensional array, "Area".

The value of a subscript is always interpreted as an integer. If a numeric expression used as a subscript evaluates to a floating point number, its value is rounded to the nearest integer.

STRING VARIABLES

String variables are variables that contain strings.

STRING VARIABLE NAMES

String variable names are formed using the same rules as numeric variable names, except for two additional rules: (1) a string variable name has a minimum length of two characters, and (2) the last character of a string variable name must be a dollar sign ("\$"). (Therefore, the number of unique characters that can be specified for a string variable name is 7.)

Examples of legal string variable names are:

A\$
Name\$
LastNam\$
Input\$

(Note that "INPUT" is a reserved keyword, and thus is not a legal name for a numeric variable. "INPUT\$", however, is not a reserved keyword, and therefore is a legal name for a string variable.)

Examples of illegal string variable names are:

LastName\$ (Too long; both numeric and string
variable names have eight-character
limits)

Str?\$ (Question mark not allowed; only
letters, numerals, imbedded under-
scores, and the trailing dollar sign)

\$Word (Dollar sign not in last position)

Str\$ (Reserved keyword)

STRING ARRAYS

Arrays of strings may be declared using the DIM statement. (See the description of the DIM statement in Section 6, *Memory Management*, for more information.)

Each element of a string array is itself a string. The maximum length of the string array elements may be declared explicitly in the DIM statement. If a maximum length is not specified, the default maximum length is 72 characters. The maximum allowable length of a string or string array element is 32,767.

Individual elements of string arrays are addressed in the same way as elements of numeric arrays, i.e., by means of subscripts. The rules governing string array subscripts are the same as those governing numeric array subscripts.

OPERATORS

ARITHMETIC OPERATORS

4041 BASIC uses the following arithmetic operators:

| Operator | Meaning | Result | Example |
|----------|--|-----------|---------|
| ↑ | Exponentiation | 6↑4 | 1296 |
| * | Multiplication | 6*4 | 24 |
| / | Floating point division (returns floating point result) | 6/4 | 1.5 |
| DIV | Integer division (returns integer result) | 6 DIV 4 | 1 |
| + | Addition (dyadic); positive (monadic) | 6+4 +4 | 10 4 |
| - | Subtraction (dyadic); negative (monadic) | 6-4 -4 | 2 -4 |
| MIN | The smaller of | 6 MIN 4 | 4 |
| MAX | The larger of | 6 MAX 4 | 6 |
| MOD | Modulo (returns remainder of integer division) | 6 MOD 4 | 2 |

NOTE

THE "↑", "DIV", AND "MOD" OPERATORS

The "↑" operator operates on either integer or floating point numbers, and returns a floating point result.

The "DIV" and "MOD" operators can take floating point numbers as operands, but always round their values to integers before performing the operation.

Results of all three operations can be stored as integers, short floating point or long floating point numbers, depending on the type of variable the user specifies to store the result in.

RELATIONAL OPERATORS

4041 BASIC uses the following relational operators:

| Operator | Meaning |
|-------------|--------------------------|
| = | Equals |
| < | Less than |
| > | Greater than |
| <> | Not equal to |
| >= (or = >) | Greater than or equal to |
| <= (or = <) | Less than or equal to |

Relational operators require two values as parameters. Each operator returns an integer result of 1 if the relation is true, or 0 if the relation is false.

The relational operators are affected by the setting of the FUZZ parameter. See the description of the SET FUZZ statement in Section 5 for more information.

LOGICAL OPERATORS

4041 BASIC uses the following logical operators:

| Operator | Meaning |
|----------|---|
| AND | Logical "and" (true if both inputs are true) |
| OR | Logical "or" (true if either input is true) |
| XOR | Logical "exclusive or" (true if one input or the other is true, but not both) |
| NOT | Logical "not" (converts true to false and vice versa) |

Logical operators AND, OR, and XOR require two values as parameters. Each operator returns an integer result of 1 (true) or 0 (false), based on the result of the operation performed on the two arguments.

In evaluating arguments, if the absolute value of an argument is less than 0.5, the argument is treated as a logical 0 (false). Otherwise, the argument is treated as a logical 1 (true).

ELEMENTS
OPERATORS

The NOT logical operator requires only one parameter, and returns the inverse of the logical evaluation of that parameter (1 if the parameter is false, 0 if the parameter is true).

The argument of the NOT logical operator must be enclosed in parentheses whenever NOT is used in an expression.

STRING OPERATOR

There is one string operator in 4041 BASIC:

| Operator | Meaning |
|-----------------|----------------|
| & | Concatenation |

The concatenation operator (&) joins two character strings.

Any number of concatenation operators may appear in a statement.

Example:

```
*A$="BAT"&"MAN"&" AND ROBIN"  
*print A$  
BATMAN AND ROBIN
```

The three strings "BAT", "MAN", and " AND ROBIN" are joined. The resulting string, "BATMAN AND ROBIN", is stored in the string variable A\$.

BINARY OPERATORS

Binary operators in 4041 BASIC operate on 32-bit signed integers. Operands are rounded to the nearest integer before binary operations are performed.

The binary operators perform bit-by-bit comparisons of the integer representations of their operands, except for the BNOT operator, which takes only one operand.

4041 BASIC uses the following binary operators:

| Operator | Meaning |
|-----------------|---|
| BAND | Binary "and" — returns a value of "1" for a bit if both corresponding bits are true, "0" otherwise. |
| BOR | Binary "or" — returns a value of "1" for a bit if either corresponding bit is true, "0" otherwise. |
| BXOR | Binary "exclusive or" — returns a value of "1" for a bit if either corresponding bit is true, but not both. |
| BNOT | Binary "not" — reverses the value of each bit in the operand. |

The BNOT operator is, strictly speaking, a function. BNOT always takes one numeric expression, enclosed in parentheses, as an argument. BNOT then returns the 2's complement of the argument, i.e., $BNOT(X) = -(X + 1)$.

Examples:

```
*print 13 band 6
4
```

To find the result of this expression, we convert 13 and 6 to their binary equivalents, and then test each bit in both numbers for an "AND" result (1 if both are true, 0 if one or the other is false).

$$\begin{array}{r} 13 - 1101 \\ 6 - 0110 \\ \hline 0100 - 4 \end{array}$$

Therefore, 13 BAND 6 is equal to 4.

```
*print 13 bor 6
15
```

To find the result of this expression, we convert 13 and 6 to their binary equivalents, and then test each bit in both numbers for an "OR" result (1 if either bit is true, 0 only if both are false).

$$\begin{array}{r} 13 - 1101 \\ 6 - 0110 \\ \hline 1111 - 15 \end{array}$$

Therefore, the result of 13 BOR 6 is 15.

```
*print 13 bxor 6
11
```

After converting 13 and 6 to their binary equivalents, test each bit for an "XOR" result (1 if either bit is true, but not both).

$$\begin{array}{r} 13 - 1101 \\ 6 - 0110 \\ \hline 1011 - 11 \text{ (decimal)} \end{array}$$

Therefore, the result of 13 BXOR 6 is 11 (decimal).

```
*print bnot(13)
-14
```

The 32-bit binary equivalent of the decimal number 13 is

0000 0000 0000 0000 0000 0000 0000 1101

The 1's complement of this number is formed by converting each 0 to a 1, and each 1 to 0:

1111 1111 1111 1111 1111 1111 1111 0010

This number is interpreted as a 32-bit 2's complement number and returned as the value of BNOT(13). In this case, the number returned is -14.

OPERATORS

IMPLIED ARRAY OPERATIONS

Certain 4041 BASIC functions and operators support operations using implied array references. An implied array reference is a reference to an array with no subscripts specified.

The following operators/functions support operations with implied array references:

| | | | |
|----|---|------|-------|
| + | (dyadic/ monadic) | ABS | LGT |
| - | (dyadic/ monadic) | ACOS | LOG |
| * | | AND | MAX |
| / | | ASIN | MIN |
| ↑ | | ATAN | MOD |
| = | (assignment/ relational operator) | BAND | NOT |
| <> | | BNOT | OR |
| > | | BOR | RND |
| < | | BXOR | ROUND |
| <= | | COS | SGN |
| >= | | DIV | SIN |
| | | EOF | SQR |
| | | EXP | TAN |
| | | INT | XOR |

Array Operations Involving Monadic Operators/Functions

Arrays used as target variables for assignment statements may be of any size, and need not be identically dimensioned to any array on the right-hand side of the equal sign.

If the expression on the right-hand side of the equal sign includes an operation involving an array and a monadic operator or function, the following rules apply when the arrays on the left- and right-hand sides are not dimensioned equally:

If the target array is dimensioned smaller than the array on the right-hand side of the equal sign, the target array is filled with as many elements from the right-hand side as will fit.

If the target array is dimensioned larger than the array on the right-hand side of the equal sign, excess elements in the target array are left unchanged after the array operation.

If the arrays on the left- and right-hand sides are not dimensioned identically, the assignment statement performs a "row-major" mapping from the array on the right-hand side into the target array.

Example:

```

100 Integer a(4,3),b(6,2)
110 Print "here's array A:"
120 For i=1 to 4
130   For j=1 to 3
140     A(i,j)=3*(i-1)+j
150     Print a(i,j);" ";
160   Next j
170   Print
180 Next i
190 Print
200 B=a
210 Print "and here's array B:"
220 For k=1 to 6
230   For l=1 to 2
240     Print b(k,l);" ";
250   Next l
260   Print
270 Next k
280 End

*run
here's array A:
1 2 3
4 5 6
7 8 9
10 11 12

and here's array B:
1 2
3 4
5 6
7 8
9 10
11 12
    
```

Line 100 dimensions two arrays, A and B.

Lines 110 through 180 assign values to the 4 X 3 array A, and prints the array in four rows and three columns.

Line 200 "maps" the values from array A into array B.

Lines 210 through 270 print array B in six rows and two columns.

Figure 2-1 illustrates the mapping procedure.

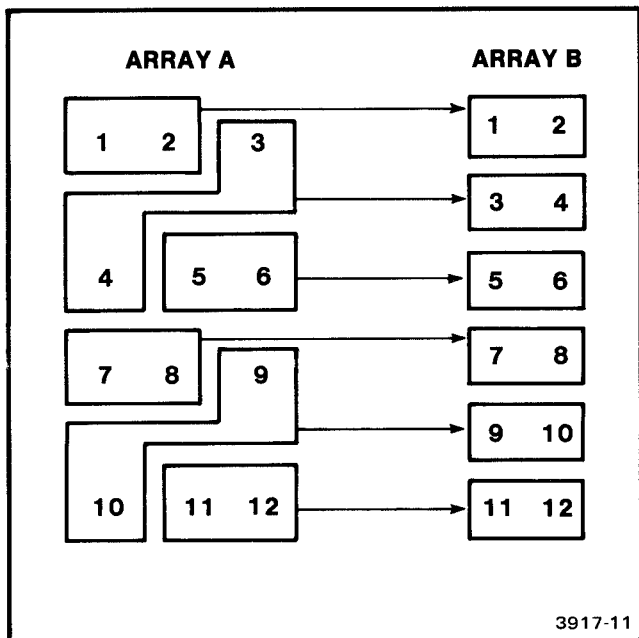


Figure 2-1. "Row-Major" Mapping in Assignment Statements.

Array Operations Using One Array and One Scalar

Array operations using one array and one scalar give an array result. For each element of the original array,

$$\text{result-array-element} = \text{original-array-element} \\ \text{OPERATION scalar}$$

Example:

```
*100 integer a(3),b(3)
*110 a(1)=1
*120 a(2)=2
*130 a(3)=3
*140 b=a+5
*150 print b
*run
6 7 8
```

Each element of array B is equal to 5 plus the corresponding element of array A.

Array Operations Using Two Arrays

Arrays used in array operations involving two arrays must be dimensioned identically (same number of rows, same number of columns).

Such operations yield an array result. For each element of the result array,

$$\text{result-element} = \text{array-A-element} \text{ OPERATION } \text{array-B-element}$$

The rules for mapping the results of such operations into an array on the left-hand side of the equal sign are the same as for array assignments involving monadic operators/functions, discussed previously.

Example:

```
*100 integer a(3),b(3),c(3)
*110 a(1)=1
*120 a(2)=2
*130 a(3)=3
*140 b(1)=-5
*150 b(2)=6
*160 b(3)=-7
*170 c=a+abs(b)
*180 print c
*190 end
*run
6 8 10
```

EXPRESSIONS

Expressions in 4041 BASIC may be of two types:
numeric or string.

NUMERIC EXPRESSIONS

A numeric expression is an expression that yields a numeric value when a statement containing the expression is executed. Numeric expressions are combinations of numeric variables, numeric constants, numeric functions, numeric arrays, and operations with the arithmetic, relational, and logical operators.

Numeric expressions may also include one or more string expressions operated on by functions or operators that return a numeric value.

Examples:

$X \uparrow Y \text{ MOD } 3 * Z$

$2 + \text{ASC}(\text{"B"})$

STRING EXPRESSIONS

A string expression yields a string when a statement containing the expression is executed. String expressions are combinations of string variables, string constants, operations with the string (concatenation) operator, and functions that yield string results.

Examples:

$A\$ \& B\$$

$\text{SEG}\$(C\$,3,10)$

$\text{CHR}\$(65) \& Y\$$

EXECUTION PRECEDENCE

Operations in 4041 BASIC are evaluated in the following order:

User-defined functions

(

Mathematical and other functions (including the NOT logical operator and BNOT binary operator)

Monadic +, -

↑ (exponentiation)

*, /, DIV

Dyadic +, -

MIN, MAX, MOD

=, <>, <, <=, >, >=

AND, OR, XOR, BAND, BOR, BXOR

& (concatenation)

)

The term "monadic" refers to an operator that requires only one operand. The term "dyadic" refers to an operator that requires two operands. In certain instances '+', for example, is a monadic operator rather than a dyadic one (as in the statement "A= + 5").

Operators with the same priority level are evaluated from left to right within an expression.

Example:

The expression

$$8/4 + 3*2 \text{ MIN } 5 \text{ MAX NOT}(1)$$

is evaluated as follows:

1. The NOT logical operator is evaluated first. Since NOT(1)= 0, we have:

$$8/4 + 3*2 \text{ MIN } 5 \text{ MAX } 0$$

2. The division and multiplication are performed next, yielding:

$$2 + 6 \text{ MIN } 5 \text{ MAX } 0$$

3. The addition is performed next, giving:

$$8 \text{ MIN } 5 \text{ MAX } 0$$

4. The MIN and MAX functions are performed in order, from left to right within the expression. Evaluating the MIN function leaves:

$$5 \text{ MAX } 0$$

5. Finally, the expression 5 MAX 0 is evaluated, yielding:

$$5$$

The expression $8/4 + 3*2 \text{ MIN } 5 \text{ MAX NOT}(1)$ evaluates to 5.

FUNCTIONS

NUMERIC FUNCTIONS

Numeric functions are special-purpose mathematical operations that take a numeric expression and return a numeric result.

For example, the SIN function returns the sine of the angle given by the numeric expression that follows the SIN keyword.

Arguments of numeric functions must be enclosed in parentheses.

Trigonometric Functions

The following trigonometric functions are provided in 4041 BASIC:

| | |
|---------------|--------------------|
| SIN (sine) | ASIN (arc sine) |
| COS (cosine) | ACOS (arc cosine) |
| TAN (tangent) | ATAN (arc tangent) |

Each of these functions takes one argument.

The arguments following SIN, COS, and TAN are interpreted as radians, degrees, or grads, depending on the value of the ANGLE parameter: 0 = radians, 1 = degrees, 2 = grads. (0 is the power-up default.) Similarly, the values returned by ASIN, ACOS, and ATAN will be radians, degrees, or grads, depending on the value of the ANGLE parameter. (See Section 5, "Environmental Control," for more information on setting the ANGLE environmental parameter.)

Other Numeric Functions

In addition to the trigonometric functions, the other numeric functions in 4041 BASIC are:

| Function | Effect |
|----------|--|
| ABS(X) | Returns absolute value of X |
| EXP(X) | Returns e (2.718281828459) to the X power |
| INT(X) | Returns largest whole number less than or equal to X |
| LGT(X) | Returns logarithm (base-ten) of X |
| LOG(X) | Returns logarithm (base-e) of X |
| PI | Returns pi (3.1415926535898) |
| RND(X) | Returns a random number (see explanation following this table) |
| ROUND(X) | Returns nearest whole number to value of X |
| SGN(X) | Returns + 1 if X > 0, 0 if X = 0, -1 if X < 0 |
| SQR(X) | Returns square root of X |
| SUM(X) | Returns arithmetic sum of numeric array X |

RND Function. The 4041's internal pseudo-random number generator accepts a numeric expression as an argument and generates a number between 0 and 1 as output.

The RND function effectively forms a "chain" of random numbers, wherein the value output by the previous invocation of the RND function becomes the input or "seed" for the next RND function.

When the RND function's argument is between 0 (inclusive) and -1 (exclusive), the RND function outputs a predetermined value, which depends on the value of the input argument, e.g., $\text{RND}(0) = 0.4041$.

When the RND function's argument is greater than 0, the RND function uses its output from the last time it was invoked as a new "seed" value to produce a new output. If the RND function has not been invoked previously, a "seed" value of 0 is used.

The RND seed value is set to 0 upon power-up, and upon execution of a RUN, DEBUG, INIT, or DELETE ALL statement.

A reproducible "chain" of random numbers can be formed by invoking the RND function once with an argument between 0 (inclusive) and -1 (exclusive), and an argument greater than 0 thereafter.

When the RND function's argument is less than or equal to -1 , the RND function generates an output based upon the time in milliseconds since power-up.

Using a long-floating point expression for a RND function argument returns a long-floating point result.

STRING FUNCTIONS

String functions are special purpose functions that manipulate character strings.

Some string functions produce string constants as their result; the SEG\$ function, for example, extracts a substring from the main body of a string.

Other string functions produce numeric results; an example is the LEN function, which returns a numeric value denoting the length of a string.

All string functions returning a string result have a dollar sign (\$) at the end of their function names; those returning numeric results do not.

String functions that return numeric results can be part of a numeric expression.

String functions yielding string results cannot be used in numeric expressions, unless the resulting string is acted upon by a function or operator that returns a numeric result.

The POS, POSN, REP\$, SEG\$, and CHR\$ functions take numeric expressions as arguments. The value of each numeric expression is rounded to the nearest integer before the function is executed. If a numeric expression used as an argument in one of these functions rounds to an integer value less than -32768 or greater than $+32767$, an overflow error is generated.

ELEMENTS
FUNCTIONS

A description of each string function and the action it performs follows:

ASC Function

Syntax Form:
ASC(strexp)

Descriptive Form:
ASC(character/string)

The ASC function returns the ASCII decimal equivalent of the first character of the string expression.

Example:

```
*print asc("A")
65
```

This command returns the ASCII decimal equivalent of the character "A" (65).

CHR\$ Function: CHR\$(numexp)

Syntax Form:
CHR\$(numexp)

Descriptive Form:
CHR\$(number)

The CHR\$ function performs the inverse of the ASC function. It evaluates the numeric expression, rounds to the nearest integer, takes this result modulo 256, and converts the resulting decimal value to its ASCII character equivalent.

Example:

```
*print chr$(66)
B
```

This command returns the ASCII character equivalent of the number 66 ("B").

LEN Function

Syntax Form:
LEN(strexp)

Descriptive Form:
LEN(string-to-be-counted)

The LEN function returns a count of the number of characters in the specified string expression.

Example:

```
*print len("Jack")
4
```

This command returns the length in characters of the string "Jack".

POS Function

Syntax Form:
POS(strexp,strexp,numexp)

Descriptive Form:
POS(string-to-be-searched,substring,
start-char-pos)

The POS function returns a numeric value giving the starting character position of a substring within a string. The first string expression given after the POS keyword is searched for an occurrence of the second string expression, starting at the character position specified by the numeric argument. If a match is found, the function returns the starting character position of the second string expression within the first string expression. If no match is found, the function returns a value of 0.

If the starting position specified is less than or equal to zero, the function starts the search at character position 1.

If the starting position specified is greater than the length of the string to be searched, the function returns a value of 0.

If the substring is not found, the function returns a value of 0 unless both the string to be searched and the substring are null; then, if the starting position is less than or equal to 1, the function returns a value of 1, else it returns a value of 0.

If the substring being searched for is null, but the string being searched is not, the POS function returns the starting location for the search.

NOTE

When the 4041 powers up, lower-case letters are considered equivalent to upper-case letters for string comparisons. See the description of the UPCASE parameter under the SET statement, discussed in Section 5, "Environmental Control", for more information about the SET UPCASE parameter.

Example:

```
*print pos("3.14159","159",1)
5
```

The string expression "3.14159" is searched for the first occurrence of the string "159", starting with character position 1 in the original string. When string "159" is found, POS returns its starting character position in the first string; in this case, a value of 5 is returned.

The POSN Function

Syntax Form:

```
POSN(strexp,strexp,numexp,numexp)
```

Descriptive Form:

```
POSN(string-to-search,substring-to-search-for,
start-pos,rep-count)
```

The POSN function is very similar to the POS function, except that POSN allows the user to specify a repetition count for the substring to be searched for. POSN then returns the position of the first character of the nth occurrence of a substring.

If the substring does not occur within the search string n times, POSN returns a value of 0, unless both the substring and the string to be searched are null. Then, if the starting position specified is less than or equal to 1, POSN returns a value of 1, else it returns a value of 0.

Example:

```
*print posn("ABABABAB","AB",1,3)
5
```

This function returns a value of 5, signifying the starting position of the third occurrence of the substring "AB" within the string "ABABABAB", starting from position 1.

REP\$ Assignment Statement

Syntax Form:

```
[line-no.] REP$(strvar,numexp,numexp)= strexp
```

Descriptive Form:

```
[line-no.] REP$(original-string,
character-position-to-
start-deleting,
#-of-chars-to-delete)
= string-to-be-inserted
```

The REP\$ assignment statement is a string "function" that inserts a substring into a string at a specified point after deleting a specified number of characters from the original string.

The resulting string can be either shorter or longer than the original string. However, characters in the resulting string in excess of the original string's dimensioned length are truncated.

Because REP\$ is a form of assignment statement and not strictly a string function, it cannot be used in a string expression, e.g., "A\$ = REP(B\$,3,2) = C\$" is not allowed.

If the starting position specified for deletion/insertion is less than or equal to 0, the new string is inserted before the start of the original string.

ELEMENTS
FUNCTIONS

If the starting position specified is greater than the length of the original string, the new string is simply appended to the original string.

If more characters are specified for deletion than there are characters following the starting position in the original string, then only as many characters as are to the right of the starting position in the original string are deleted.

If the number of characters to be deleted is less than or equal to zero, then no characters are deleted.

If the original string is an array, then a subscript must be given as part of the REP\$ argument or an error results.

Example:

```
*a$="Tom, Jack, and Harry"  
*rep$(a$,6,4)="Dick"  
*print a$  
Tom, Dick, and Harry
```

Four characters, starting with the sixth character of the original string A\$, are deleted, and a new substring is inserted at the point at which deletion began. The result, "Tom, Dick, and Harry", is displayed.

SEG\$ Function

Syntax Form:
SEG\$(strex, numexp, numexp)

Descriptive Form:
SEG\$(string, start-pos, substring length)

A substring of a specified length is extracted from a string, starting at a specified character position.

If the starting position specified is less than or equal to 0, it is considered equal to 1.

If the starting position is greater than the length of the string, SEG\$ returns the null string.

If the specified substring length is less than or equal to 0, SEG\$ returns the null string.

If the specified substring length would require reading past the end of the original string, SEG\$ returns a substring containing only the characters between the specified starting position and the end of the original string.

Example:

```
*print seg$("Firstname E. Lastname",14,8)  
Lastname
```

The string returned is the substring of length 8 starting at character position 14 of the string "Firstname E. Lastname".

STR\$ Function

Syntax Form:
STR\$(numexp)

Descriptive Form:
STR\$(number-to-be-converted-to-string)

A numeric expression is evaluated and converted to a string. STR\$ forms this string to match the argument's default print format.

Example:

```
*pi$=str$(3.14159)
```

The number 3.14159 is converted to string representation and stored in string variable Pi\$.

TRIM\$ Function

Syntax Form:
TRIM\$(strex)

Descriptive Form:
TRIM\$(string-to-be-trimmed)

The TRIM\$ function removes leading and trailing spaces from a string.

Example:

```

100   A$="    this-is-string-A$    "
110   Print a$
120   Print len(a$)
130   A$=trim$(a$)
140   Print a$
150   Print len(a$)
*run
      this-is-string-A$
27
this-is-string-A$
17

```

Line 130 removes leading and trailing spaces from the string " This-is-string-A\$ " and stores the result in string variable A\$.

VAL Function**Syntax Form:**

VAL(strexp)

Descriptive Form:

VAL(string-to-be-converted-to-number)

The VAL function searches a string expression for an occurrence of a substring that would form a valid number if converted to numeric representation. The substring is then converted to a number. If no legal number is found, an error occurs.

The VAL function recognizes standard positional and scientific notations, but does not recognize engineering or binary/octal/hexadecimal notations.

After the first character is encountered that could be part of a valid number (either a numeral, or a plus or minus sign followed by a numeral), the computer continues to scan the string to the right until it either reaches the end of the string or finds a character that could not be part of a valid number. Note that the letter "E" is included among characters that could be part of a valid number.

The ASK("CHPOS") function returns the position within the string of the last character scanned by a VAL or VALC function. (See the description of the ASK function in Section 5, "Environmental Control", for more information.)

Examples:

```

*print val("3.14159abc")
3.14159

*print val("teststring3.14159e010moreteststring")
3.14159E+10

*print val("teststring3.14moreteststring")
3.14

```

The VALC Function**Syntax Form:**

VALC(strexp,numexp)

Descriptive Form:

VALC(string-to-be-searched,start-pos)

The VALC function works similarly to the VAL function, except that the user specifies the starting position for the search. The first ASCII string that evaluates to a valid number, beginning with the specified starting position, is extracted from the original string.

The ASK("CHPOS") function returns the position within the string of the last character scanned by a VAL or VALC function. (See the description of the ASK function in Section 5, "Environmental Control", for more information.)

Example:

```

100   Temp$="abc1.23456789012345,6.78"
110   A=val(temp$)
120   B=valc(temp$,ask("chpos"))
130   Print a
140   Print b
*run
1.23457
6.78

```

Line 110 stores the value 1.23457 in A. Line 120 then reads the remainder of string Temp\$ from the point at which the last VAL or VALC function stopped. Line 120 then stores the first valid number found, 6.78, into B.

STATEMENTS

STATEMENT NUMBERS

4041 BASIC is a "statement-oriented" language. Each statement of a 4041 BASIC program is preceded by a statement number (also called a "line number"). Line numbers are integers between 1 and 65535, inclusive.

Statements are executed in increasing line-number order, unless: (1) a statement changes the order of execution by transferring control to a different statement than the next one in sequence, or (2) a condition is sensed that transfers control to a special handler for that condition.

LABELS

An optional alphanumeric label may follow any line number. The label is followed by a colon to set it off from the rest of the statement. A line may be referenced by either its label or its number.

Example:

```
400      Goto target
:
500 Target:  ! execution continues from here
```

Line 400 transfers control to the line with the label "Target".

Labels are formed using the same rules as numeric variable names: the label can have a maximum of eight characters, the first character must be a letter, and the remaining characters can be any combination of letters, numbers, or imbedded underscores.

A label is actually a floating point variable, with a numeric integer value equal to the number of the program line the label appears on. That value is in the closed interval 1 to 65535.

There are, however, restrictions that apply to labels that do not apply to other variables. The only way the value of a label variable can change is for the line number of the program line that the label appears on to change, as with a RENUMBER operation. After renumbering, the label variable will have an integer value equal to the line number that the line has been renumbered to.

DELETE VAR will not work with label variables. To make a label undefined, either the program line the label appears on must be deleted, or the program line edited and the label removed. Either way, the label variable then becomes a normal undefined short-floating-point variable.

A label variable can be used anywhere a numeric expression is valid, either in a numeric expression or by itself. A label variable referenced anywhere a numeric expression is invalid (e.g., as an assignment statement destination, or in an INIT VAR statement) will generate an error when the statement is executed.

No two lines in the same program segment can have the same label. In addition, if two lines have the same label in different program segments, one of the labels must be specified as a "local" variable to that program segment. (See Section 11, "Subprograms and User-Defined Functions", for more information about local variables.)

RESERVED KEYWORDS

Certain combinations of letters are reserved as keywords in 4041 BASIC. Keywords cannot be used as numeric or string variable names, or as alphanumeric line labels (with two exceptions: INDEX and SYSDEV).

Keywords of more than four letters may be abbreviated to four-letter length when used either in a program or in immediate mode. Abbreviations of other lengths are not accepted as keywords; such abbreviations may be used as variable names in programs.

Stream specification parameters and ASK/ASK\$ parameters are NOT reserved keywords.

The following words (and, for those longer than four letters, their first-four-letter subsets) are reserved keywords in 4041 BASIC:

| | | | | | |
|----------|----------|----------|---------|----------|---------|
| ABORT | CALL | ELSE | IF | PI | STEP |
| ABS | CHRS | ENABLE | IFC | POLL | STOP |
| ACOS | CLOSE | END | IMAGE | POS | STR\$ |
| ADVANCE | COMPRESS | EOF | INDEX* | POSN | SUB |
| ALL | CONNECT | EOI | INIT | PPC | SUM |
| ALTER | CONSOLE | ERROR | INPUT | PPU | SYNTAX |
| AND | CONTINUE | EXIT | INT | PRINT | SYSDEV* |
| ANGLE | COPY | EXP | INTEGER | PROCEED | |
| APPEND | COS | | IODONE | PROGRAM | TAN |
| ASC | | FILE | KEY | PROMPT | TCT |
| ASIN | DATA | FLOW | KEYS | PUTMEM | THEN |
| ASK | DCL | FOR | | | TIME |
| ASK\$ | DEBUG | FORMAT | LEN | RBYTE | TO |
| ATAN | DELETE | FUNCTION | LET | RCALL | TRACE |
| ATN | DELN | FUZZ | LGT | READ | TRAP |
| AUTOLOAD | DELS | | LINE | REM | TRIM\$ |
| | DIM | GET | LIST | REN | TYPE |
| BAND | DIR | GETMEM | LLO | RENAME | UNL |
| BNOT | DISABLE | GO | LOAD | RENUMBER | UNT |
| BOR | DISMOUNT | GOSUB | LOCAL | REP\$ | UPCASE |
| BRANCH | DIV | GOTO | LOG | RESTORE | USING |
| BREAK | DRIVER | GTL | LONG | RESUME | |
| BUFFER | | | MAIN | RETRY | VAL |
| BXOR | | | MAX | RETURN | VALC |
| | | | MIN | RND | VAR |
| | | | MLA | ROUND | VIEW |
| | | | MOD | RUN | |
| | | | MONITOR | SAVE | WAIT |
| | | | MTA | SDC | WBYTE |
| | | | | SEG\$ | |
| | | | NEXT | SELECT | XOR |
| | | | NOBREAK | SELFTEST | |
| | | | NOT | SET | |
| | | | NOTRACE | SGN | |
| | | | | SIN | |
| | | | OF | SLIST | |
| | | | OFF | SPD | |
| | | | ON | SPE | |
| | | | OPEN | SQR | |
| | | | OR | SRQ | |

*MAY be used as line label, variable name, or subprogram name.

Section 3

FRONT PANEL, P/D KEYBOARD, AND INSTRUMENT OPTIONS

INTRODUCTION

This section discusses the layout and functions of controls and other features on the front panel. It lists and describes optional features and interfaces avail-

able for the 4041. It also describes the keys on the optional Program Development (P/D) Keyboard (Option 31).

FRONT PANEL

The 4041 front panel is divided into seven sections (Figure 3-1):

1. The power switch;
2. The 20-character thermal printer;
3. The DC-100 magnetic tape drive;
4. The jack for plugging in the Program Development Keyboard;
5. The 20-character, 16-segment alphanumeric display;
6. Four status lights to indicate the state of the system;
7. The user-definable function/numeric keypad.

Refer to the 4041 Computer/Controller Operator's Manual for additional information on the following items.

FRONT PANEL, KEYBOARD, & OPTIONS

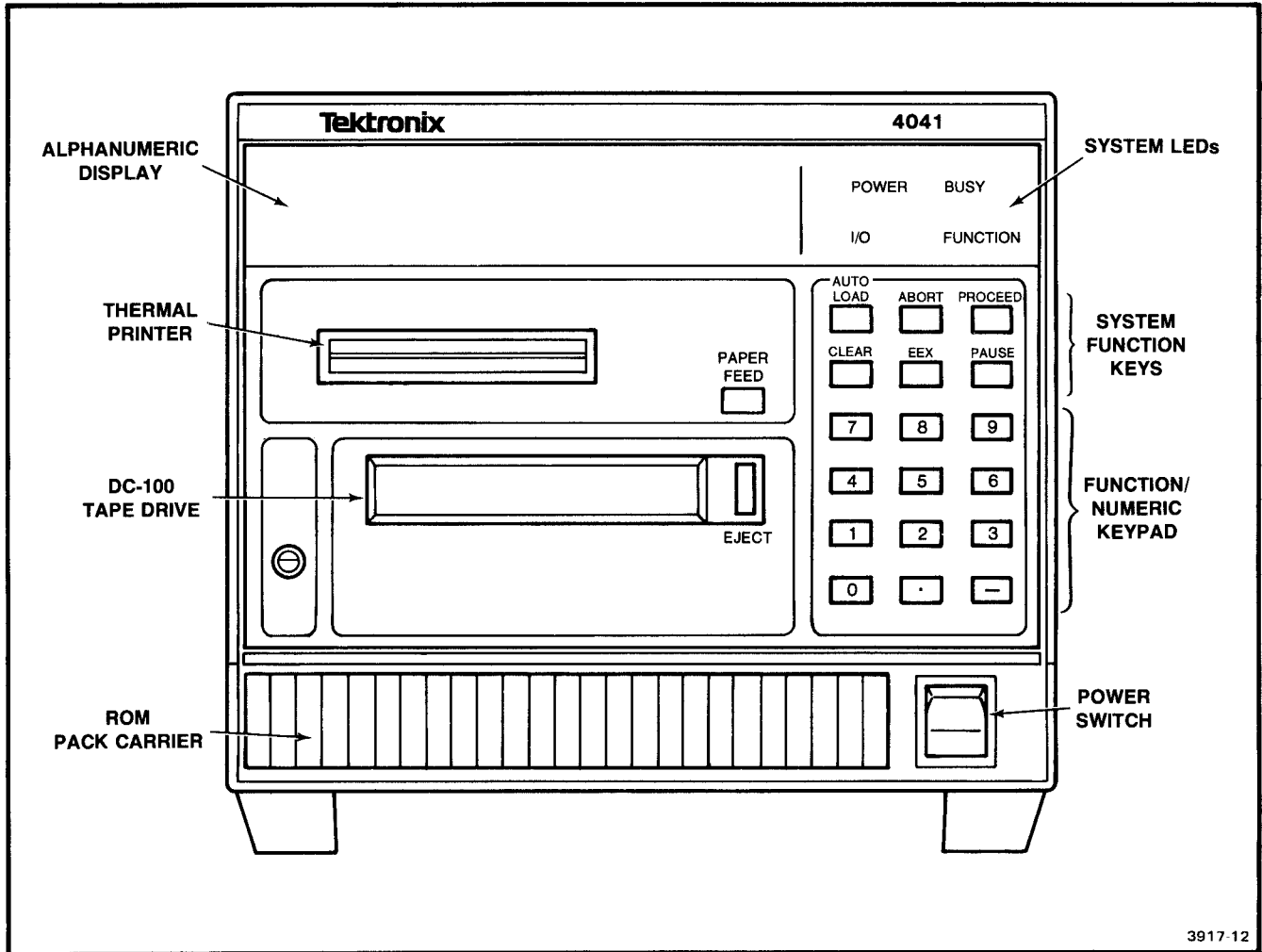


Figure 3-1. 4041 Front Panel.

POWER SWITCH

The 4041 power switch is located in the lower right-hand corner of the front panel (Figure 3-1).

THERMAL PRINTER

The 20-character printer uses thermal-sensitive 60mm paper to print program listings and results. Instructions for loading the printer paper are given on the inside cover of the paper access door, located atop the 4041, and in the 4041 Computer/Controller Operator's Manual.

PAPER FEED Button

Pressing the PAPER FEED button advances the printer paper.

DC-100 TAPE DRIVE

The drive for the DC-100 magnetic tape is located on the front panel below the paper feed slot for the thermal printer. The drive is designed to accept 3.18" x 2.4" x .47" DC-100 magnetic tape cartridges.

TAPE EJECT Button

Pressing the TAPE EJECT button releases the DC-100 magnetic tape cartridge from the drive.

KYBD INPUT

The KYBD INPUT jack is used to attach a Program Development (P/D) Keyboard (Option 31) to the 4041.

ALPHANUMERIC DISPLAY

The 20-character, 16-segment alphanumeric display is located on the top left of the front panel.

SYSTEM STATUS LIGHTS

Four LED's that indicate the state of the system are located at the top right of the front panel. The LED's and their meanings are as follows:

- BUSY: Indicates that a program is running. A blinking BUSY light indicates that the system is PAUSEd.
- POWER: Indicates the machine is on.
- I/O: Indicates an I/O operation is being performed.
- FUNCTION: Indicates that numeric keys are set to select user-definable functions, rather than numeric input.

SYSTEM FUNCTION KEYS

Six system functions keys are located below the function lights on the front panel. They include the AUTOLOAD key, the ABORT key, the PROCEED key, the CLEAR key, the EEX key, and the PAUSE key.

The AUTOLOAD Key

Pressing the AUTOLOAD key loads and runs a file named AUTOLD from the current DC-100 tape when the front panel is the system console. If the front panel is not assigned as system console, this key has no effect.

The ABORT Key

Pressing the ABORT key cancels execution of a running program if the ABORT condition is not disabled. Control transfers to a user-defined ABORT handler, if one is in effect. The ABORT key also terminates LIST and SAVE operations in immediate mode. If the ABORT condition is not disabled, the front panel ABORT key cancels execution regardless of the system console assignment.

The PROCEED Key

When the front panel is assigned as system console, pressing the PROCEED key: (a) starts execution beginning with the first line of the current program, if the system is not PAUSEd, or (b) continues program execution at the interrupt point following a PAUSE. The PROCEED key also completes an input operation from the front panel or P/D Keyboard.

The CLEAR Key

Pressing the CLEAR key clears the alphanumeric display during input from the front panel keypad or from the P/D Keyboard. CLEAR has no effect on data already stored in the 4041's memory.

The EEX Key

Pressing the EEX key during an input operation indicates that the numbers that follow are part of the exponent of a number written in scientific notation. For example, to input the number 1000 using the numeric keypad, the user can press the key sequence <1>, <0>, <0>, <0>, <PROCEED>, or the user can press the key sequence <1>, <EEX>, <3>, <PROCEED>. This second sequence of keys tells the 4041 that the user is inputting "one-times-ten-to-the-third", or 1000.

The PAUSE Key

Pressing the PAUSE key stops program execution without cancelling it, regardless of system console assignment. The PAUSE key also halts execution of a list or save operation in immediate mode.

Execution can be resumed from the point at which the program was PAUSEd by issuing a CONTINUE command or, if the front panel is the system console, by pressing the PROCEED key or the P/D Keyboard CONTINUE key. PAUSEd LIST or SAVE operations cannot be resumed.

If proceed-mode I/O is in progress at the time the PAUSE key is pressed, the proceed-mode I/O is allowed to complete before the PAUSE key is recognized. Refer to Section 8, Input/Output.

USER-DEFINABLE FUNCTION/NUMERIC KEYS

The twelve user-definable function/numeric keys are located below the six system function keys.

When the front panel is the system console device and the user-definable functions are enabled, pressing one of the keys 0 through 9 transfers control to a user-defined function for the key that was pressed. (Pressing key 0 transfers control to a handler for function key "10".) If the user-definable function keys are not enabled, the 4041 stores one function key entry in a buffer until the keys are enabled. If the 4041 already has an entry stored, it emits a loud "beep" when another of these keys is pressed.

When an INPUT operation is being performed from the front panel (I/O system light is on), keys 0 through 9 act as a numeric keypad. The "." key is a decimal point, and the "-" key is a minus sign.

PROGRAM DEVELOPMENT (P/D) KEYBOARD

The Program Development (P/D) Keyboard (Figure 3-2) is available as Option 31 on the 4041. The P/D Keyboard can be used for alphabetic and numeric input to the standard 4041, and for program development on a 4041 equipped with Option 30 (P/D ROMs and carrier).

The P/D Keyboard resembles a standard typewriter keyboard with additional keys to perform specific functions. A description and explanation of the special keys on the P/D Keyboard follows.

NOTE

The special function keys described in the following section, with the exception of ABORT and PAUSE, only work when the front panel is assigned as the system console.

SPECIAL FUNCTION KEYS

The RUN Key

Typing a line number and pressing the RUN key starts the current program from that line number.

Pressing the RUN key without first typing a line number starts the current program from the first line.

If the Program Development ROMs (Option 30) are not installed, the run key begins execution at the program start. Starting line numbers are not permitted.

Running programs by pressing the RUN key disables all breakpoints and trace flags within the program. (See Section 4 on Editing, Debugging, & Documentation for more information about breakpoints and trace flags.) The RUN key performs the same "housekeeping" functions as the RUN statement (described in Section 7, Control Statements).

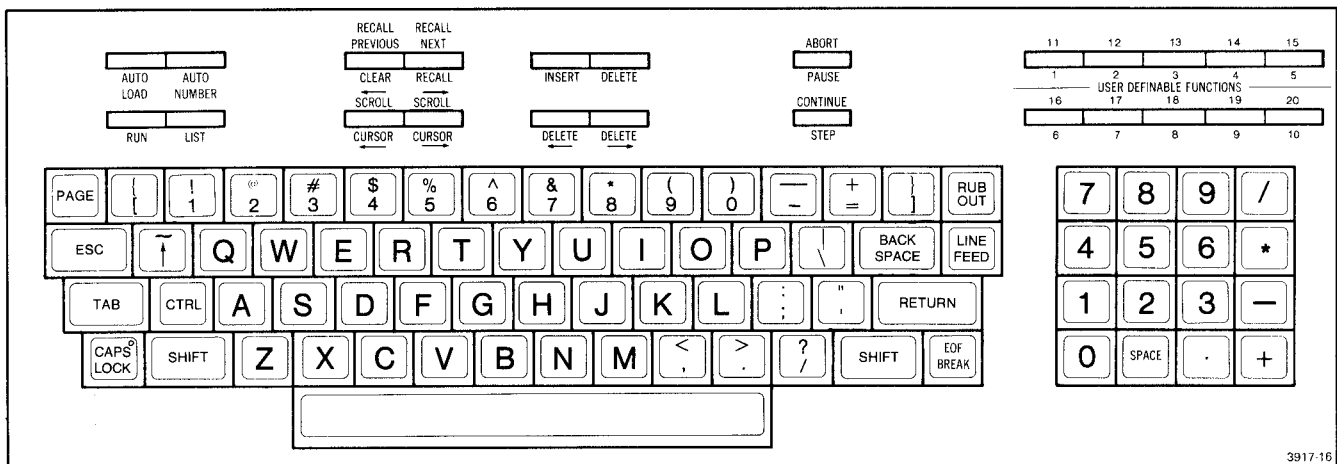


Figure 3-2. 4041 Program Development (P/D) Keyboard.

The LIST Key

Pressing the LIST key causes a current program listing to be printed on the thermal printer.

Typing a line number and pressing the LIST key causes that line to be printed on the thermal printer.

Typing two line numbers separated by the word TO and pressing the LIST key prints all program lines with line numbers greater than or equal to the first number and less than or equal to the second number on the thermal printer.

Example:

If the program currently in memory is

```
10  Rem this is a sample program
20  Apples=10
30  Oranges=20
40  Fruit=apples+oranges
```

then pressing the LIST key lists the entire program; typing "20" and pressing the LIST key prints line 20 on the thermal printer; typing "10 TO 30" and pressing the LIST key prints lines 10, 20, and 30 on the thermal printer; and typing "15 TO 35" and pressing the LIST key prints lines 20 and 30 on the thermal printer.

If the Program Development ROMs (Option 30) are not installed, the LIST key is ignored.

The AUTOLOAD Key

Pressing the AUTOLOAD key loads a file named AUTOLD from the DC-100 tape.

The AUTOLOAD key only takes effect when the 4041 is idle (not executing a BASIC statement). If the AUTOLOAD key is pressed when a program is executing, it has no effect.

The AUTONUM Key

The AUTONUM key activates and de-activates the 4041's automatic numbering feature.

Pressing the AUTONUM key causes the 4041 to print the number 100 on the alphanumeric display. The user then enters a program line and presses RETURN, whereupon the line number is automatically incremented by 10.

Typing a number and pressing the AUTONUM key causes the 4041 to print the number entered, accept a line, and increment the line number by 10 each time the RETURN key is pressed.

Typing two numbers separated by a comma and pressing the AUTONUM key causes the 4041 to display the first number, accept a line, and increment the line number by the amount of the second number each time the RETURN key is pressed.

Pressing the AUTONUM key when the automatic numbering feature is activated de-activates automatic numbering.

The AUTONUM key is ignored if the Program Development ROMs (Option 30) is not installed.

The CLEAR Key

Pressing the CLEAR key clears the alphanumeric display during input from the front panel or Program Development Keyboard.

The RECALL PREV Key

Pressing the RECALL PREV key brings into the display the line preceding the last line recalled with the RECALL, RECALL PREV, or RECALL NEXT keys.

The RECALL PREV key is ignored if the Program Development ROMs are not installed.

The RECALL Key

Typing a number and pressing the RECALL key brings the line having that line number into the display.

If the program contains no line with the number specified, the program line with the next-lower number appears in the display.

The RECALL key is ignored if the Program Development ROMs are not installed.

The RECALL NEXT Key

Pressing the RECALL NEXT key brings into the display the line following the last line recalled with the RECALL, RECALL PREV, or RECALL NEXT keys.

The RECALL NEXT key is ignored if the Program Development ROMs (Option 30) are not installed.

The CURSOR < Key

Pressing the CURSOR < key moves the display cursor left. When the cursor is in a display position also occupied by a character, the cursor and the character are displayed alternately.

If the display cursor is in the first position of the display, pressing the CURSOR < key scrolls the contents of the display to the right.

If the first position of the display contains both the first character of the line being displayed and the cursor, pressing the CURSOR < key has no effect.

The SCROLL < Key

Pressing the SCROLL < key causes the contents of the display, including the position of the cursor, to move one space to the left. The first character currently in the display moves off the display to the left, while the character following the rightmost character currently on the display moves into the display from the right.

The cursor retains its position relative to the message being displayed (i.e., if it shared a display position with the first character of a word, it continues to share that position after the SCROLL < key is pressed).

If the cursor is in the first (left-most) position of the display, however, pressing the SCROLL < key causes the contents of the display to move one space to the left, but the cursor remains in the first position. Only in this case does pressing the SCROLL < key cause the cursor to change its position within the message.

If the current line contains no characters after the one in the right-most display position, pressing the SCROLL < key has no effect.

The CURSOR > Key

Pressing the CURSOR > key causes the cursor to move one position to the right in the display.

If the cursor occupies the rightmost position of the display, pressing the CURSOR > key causes the contents of the display to "scroll" one space to the left.

If the cursor is to the right of the rightmost character in the current line, pressing the CURSOR > key has no effect.

The SCROLL > Key

Pressing the SCROLL > key causes the contents of the display to move one character position to the right.

The cursor retains its position relative to the message being displayed (i.e., if the cursor shares a position with the first character of a word, the cursor shares that position after the SCROLL > key is pressed).

If the cursor is in the rightmost position of the display, however, pressing the SCROLL > key causes the contents of the display to move one space to the right while the cursor remains in the rightmost position. (The SCROLL > key can never move the cursor off the display.) Only in this case does pressing the SCROLL > key change the position of the cursor within the message.

If the first character of the the current line is in the leftmost display position, pressing the SCROLL > key has no effect.

The DELETE < Key

Pressing the DELETE < key deletes the character to the left of the cursor.

If the cursor is in the leftmost position of the display, pressing the DELETE < key has no effect. The DELETE < key will not delete characters that the user cannot see on the alphanumeric display.

The INSERT Key

Pressing the INSERT key places the 4041 in Insert mode from Replace mode.

Replace Mode. The 4041 is said to be in Replace mode whenever it is not in Insert mode. The 4041 powers up in Replace mode. In Replace mode, a " " cursor is used.

In this mode, the cursor replaces characters already in the display with new characters entered from the keyboard. If the display contained the characters

DEF

with the cursor in the same display position as the letter "D", typing the characters "ABC" with the 4041 in replace mode yields the following display:

ABC

The cursor then occupies the display position to the right of the letter "C". The contents of the original display have been replaced, hence the name Replace mode.

Insert Mode. Pressing the INSERT key puts the 4041 into Insert mode. Pressing the INSERT key again when the 4041 is in Insert mode returns the 4041 to Replace mode.

When the 4041 is placed into Insert mode, characters in the same position as and to the right of the cursor are shifted right one space. The cursor always occupies a blank space when the 4041 is in Insert mode. Insert mode uses a " " cursor.

Entering a character when in Insert mode causes the character just entered to appear at the position previously occupied by the cursor. The cursor and all characters to the right of it are shifted one space to the right as each character is entered.

If the cursor occupies the rightmost position of the display, then the cursor remains stationary when characters are entered, while all characters to the left of the cursor are shifted left.

The DELETE > Key

Pressing the DELETE > key when the 4041 is in Insert mode deletes the character to the right of the cursor. Display contents to the right of the deleted character are shifted left one space.

Pressing the DELETE > key when the 4041 is in Replace mode deletes the character in the display position occupied by the cursor. Display contents to the right of the deleted character are shifted left one space; the cursor then shares a display position with the character following the deleted character.

In Insert mode, pressing the DELETE > key has no effect if the cursor is in the rightmost display position. The DELETE > key never deletes a character that the user cannot see (i.e., one not currently in the display).

If the cursor is to the right of the rightmost character in the current line, pressing the DELETE > key has no effect.

The DELETE Key

Typing a number and pressing the DELETE key deletes the line with that number from the program and clears the display. If no line with that number is in the program, pressing the DELETE key has no effect.

Typing two numbers separated by the word TO and pressing the DELETE key deletes program lines with line numbers equal to or greater than the first and less than or equal to the second.

Example:

If the program currently in memory is

```
10  Rem this is a sample program
20  Apples=10
30  Oranges=20
40  Fruit=apples+oranges
```

typing "40" and pressing the DELETE key deletes line 40; typing "10 TO 20" and pressing the DELETE key deletes lines 10 and 20; typing "15 TO 35" and pressing the DELETE key deletes lines 20 and 30; and typing "25" and pressing the DELETE key has no effect.

If no number is specified in the display, pressing the DELETE key has no effect. The DELETE key is ignored if the Program Development ROMs are not installed.

The PAUSE Key

Pressing the PAUSE key halts execution of a running program. The 4041 recalls the point at which the program was halted. Execution can be continued from this point by pressing the CONTINUE key or the front panel PROCEED key.

The PAUSE key can also terminate a LIST or SAVE operation. The LIST or SAVE operation cannot be resumed, though.

This key has the same effect as the BREAK key.

The ABORT Key

Pressing the ABORT key cancels execution of a running program if the ABORT condition is not disabled. Control transfers to a user-defined ABORT handler, if one is in effect, or to the system ABORT handler. The ABORT key also terminates LIST and SAVE operations in immediate mode. If the ABORT condition is not disabled, the front panel ABORT key cancels execution regardless of the system console assignment.

See also the description of the ABORT condition in Section 12, "Interrupt Handling."

The STEP Key

Pressing the STEP key executes the first line of the program or the next line of a PAUSED program. It puts the 4041 into Debug mode. Refer to Section 4, Editing, Debugging, and Documentation.

The STEP key is ignored if the Program Development ROMs are not installed.

The CONTINUE Key

Pressing the CONTINUE key restarts execution of a paused program (stopped by a PAUSE or BREAK or a STOP statement) from the point at which execution was interrupted. Execution will resume under Run or Debug mode, depending on the mode it was in when the pause occurred.

The CONTINUE key does not start execution of a program that is not paused. If the CONTINUE key is pressed when the current program is not being executed and not stopped or paused, an error is generated. Pressing the CONTINUE key while a program is executing has no effect.

THE USER-DEFINABLE FUNCTION KEYS

The user-definable function keys allow the user to define up to 20 P/D Keyboard interrupts.

When the FRTD is the system console device and the user-definable function keys are enabled, pressing a user-definable function key for which a handler routine is defined transfers control to that handler after the current statement completes execution. (See Section 12, "Interrupt Handling", for more information about function key interrupts.)

Functions 1 through 10 are obtained by pressing the corresponding user-definable function key alone. Functions 11 through 20 are obtained by pressing the SHIFT key and the corresponding user-definable function key.

(Functions 1 through 10 are also obtainable by pressing function keys 1 through 9 and 0 (10) on the front panel.)

THE CURSOR/NUMERIC KEYPAD

The numeric keypad is used to enter numeric data. In addition, the keypad can enter the characters +, -, *, /, ., and space. The numeric keypad can not be used to select user-definable functions.

THE ALPHANUMERIC KEYBOARD

The alphanumeric keypad is used to enter the 128 ASCII (American Standard Code for Information Interchange) characters. This keyboard also contains some special function keys.

The BREAK Key

Pressing the BREAK key halts execution of a running program. The 4041 recalls the point at which the program was halted. Execution can be continued from this point with the CONTINUE statement. If the front panel is the system console, execution can also be resumed by pressing the CONTINUE key or the front panel PROCEED key.

The BREAK key also terminates LIST or SAVE operations. These operations can not be resumed.

If the program currently in memory is not being executed, pressing the BREAK key has no effect.

If proceed-mode I/O is in progress when the BREAK key is pressed, the proceed-mode I/O is allowed to complete before the BREAK key is recognized.

This key has the same effect as the PAUSE key.

The SHIFT Key

This key is similar to the SHIFT key on a typewriter. It determines whether alphabetic characters are interpreted as upper or lower case; for keys with more than one associated function, it determines which function is to be performed.

The CTRL Key

This key is used with other keyboard keys to enter the 32 ASCII control characters (<0> - <31>. For example, the ASCII character EXT, or Control-C, is entered by holding down the CTRL key and pressing "C".

The RUBOUT Key

The RUBOUT key backspaces and erases the previous character.

The RETURN Key

RETURN, or Carriage Return, ends a line of input and sends the line to the 4041.

The CAPS LOCK Key

This key is lit when pressed on, and unlit when pressed off. When on, all letters entered are sent as upper case characters, regardless of the SHIFT key. This key has no effect on characters other than the 26 letters of the alphabet.

Control Characters

Control characters are sent by pressing the CTRL key and another key.

Control characters that are printed as program output are executed, i.e., a line feed character produces a line feed in output, etc.

Control characters that are listed as part of a program are displayed and not executed. On the front panel or on a user terminal, control characters are displayed as an up-arrow followed by the appropriate character (J for line feed, I for tab, etc.). Control characters listed on the thermal printer are underscored.

The keys to press along with the CTRL key to obtain various control characters are as follows:

BELL — G
BACKSPACE — H
TAB — I
LINE FEED — J
PAGE — L
CARRIAGE RETURN — M
ESCAPE — [

INSTRUMENT OPTIONS

Table 3-1 lists the available instrument options to the standard 4041. Note that certain of these options are described as field installable, as opposed to factory-installed options.

Options 01, 02, and 03 are mutually exclusive. Any 4041 may be equipped with up to 512K of memory, regardless of other options installed on the unit.

Table 3-1
INSTRUMENT OPTIONS

| Option | Description |
|-------------|--|
| 01 | One additional GPIB port with Direct Memory Access, and one additional RS-232 port. |
| 02 | 8-bit parallel TTL Interface. |
| 03 | One SCSI interface port and one additional RS-232-C port. |
| 20 | 64K bytes total memory. |
| 21 | 96K bytes total memory. |
| 22 | 128K bytes total memory. |
| 23 | 160K bytes total memory. |
| 24 | 256K bytes total memory. |
| 25 | 512K bytes total memory. |
| 30 | Program development ROMs and carrier. |
| 31 | Program development keyboard. |
| A1 | Universal European 220V/16A power cord. |
| A2 | United Kingdom 240V/13A power cord. |
| A3 | Australian 240V/10A power cord. |
| A4 | North American 240V/15A power cord. |
| A5 | Swiss 250V/15A/50 Hz power cord. |
| 4041F01 | Field-installable Option 01. |
| 4041F02 | Field-installable Option 02. |
| 4041F03 | Field-installable Option 03. |
| 4041F30 | Field-installable Option 30. |
| 4041F31 | Field-installable Option 31. |
| 040-1021-00 | Field-installable initial memory expansion; expands memory from 32K bytes to 64K bytes. |
| 040-1022-00 | Additional 32K bytes field-installable memory for memory expansion above 64K bytes and up to 160K bytes. |

Options 01 and 4041F01 — Additional GPIB and RS-232-C Interface Ports

Option 01 (and its field-installable equivalent, 4041F01) provide an additional GPIB and RS-232-C interface port on the rear panel.

The additional GPIB interface port allows the 4041 to control up to 30 GPIB devices simultaneously (15 on the standard interface port and 15 on the optional one). In addition, the optional GPIB interface port can use Direct Memory Access (DMA) to speed data transfers.

With the optional RS-232-C interface port, the 4041 can communicate with two RS-232-C devices simultaneously. Thus, for example, a computer terminal connected to the standard RS-232-C interface port could be used as the system console device, while a line printer connected to the optional port could be used to provide program listings or hardcopy output.

The optional GPIB port is designated by the driver name "GPIB1:". Its operation is identical to that of the standard GPIB interface port (driver name "GPIB0:") except:

1. The TRA parameter may be set to DMA on GPIB1, but not on GPIB0; and
2. Logical units 0 through 30, if not specified otherwise, address the devices with corresponding primary addresses on the standard GPIB interface port. Data paths to devices on the optional GPIB interface port must be explicitly specified, either via an OPEN statement or via a "#" clause in an I/O statement.

See Sections 8 ("Input/Output") and 9 ("Instrument Control With GPIB"), and Appendix D ("Stream Specifications") for more information about the use of the GPIB0 and GPIB1 driver names with OPEN and other I/O statements.

The optional RS-232-C interface port is designated by the driver name "COMM1:", and data paths to it are specified by means of that driver name, used within an OPEN statement or a "#" clause in an I/O statement. In all other respects, the operation of the optional RS-232-C interface port is identical to that of the standard port.

Options 02 and 4041F02 — 8-Bit Parallel TTL Interface

Option 2 and its field-installable equivalent, 4041F02, provide an 8-bit parallel TTL interface port on the rear panel. This port allows the user to write custom interfaces to control non-GPIB and non-RS-232-C devices.

The 4041 communicates with the Option 2 interface port by means of RCALL routines "Opt2in" and "Opt2out". Both these routines are described along with the RCALL statement in Section 7, *Control Statements*.

Options 30 and 4041F30 — Program Development (P/D) ROMs and Carrier

Option 30 and its field-installable equivalent, 4041F30, consist of the Program Development ROMs and their carrier that fits into the bottom of the front panel.

A 4041 equipped with Option 30 can translate BASIC statements and data into the 4041's internal data storage (ITEM) format, and can execute immediate-mode statements.

4041 units without Option 30 cannot LIST, LOAD, or SAVE programs in ASCII, RECALL program lines, or execute immediate-mode statements. For this reason, 4041 units without Option 30 are sometimes referred to as "execute-only" or "XO" versions of the 4041. When more than one 4041 is available, several XO units can be used to run programs in the manufacturing or test environment while other Option-30-equipped units are used to develop programs.

Options 31 and 4041F31 — Program Development (P/D) Keyboard

Option 31 and its field-installable equivalent, 4041F31, consist of an ASCII keyboard that can be plugged into the 4041 front panel. This keyboard allows the user to enter alphanumeric characters to the front panel. The P/D keyboard's special functions keys are also useful in debugging programs.

INSTRUMENT OPTIONS

Option 03 and 4041F03-SCSI and Additional RS-232-C Interface Ports

NOTE

4041 firmware version 2.0 or higher is required for this modification.

Option 03 (and the field installable 4041F03) provide a Small Computer System Interface (SCSI) port and an additional RS-232-C interface port on the rear panel.

The SCSI adds disk based mass storage to the 4041. It is designed to interface with the TEK 4925 and 4926 modified mass storage units, but can be used with other SCSI compatible devices. The SCSI can communicate with up to seven devices (the 4041 is the eighth device).

Refer to the 4041 Option 01 for the optional RS-232-C interface port information.

The SCSI interface port is designated by the driver name "DISK".

The optional RS-232-C interface port is designated by the driver name "COMM1:", and data paths to it are specified by means of that driver name, used within an OPEN statement or a "#" clause in an I/O statement. In all other respects, the optional RS-232-C interface port operation is identical to that of the standard port.

Section 4

PROGRAM EDITING, DEBUGGING, & DOCUMENTATION

INTRODUCTION

This section discusses commands in 4041 BASIC that are used to edit programs under development, to debug programs, and to add comments to programs.

An additional debugging feature of 4041 BASIC besides those described here is the ability to examine contents of variables any time the 4041 is idle (i.e., not actively executing a program).

To examine the contents of a variable, simply type the variable name from the P/D keyboard or user terminal and press RETURN. The value of the variable will appear on the front-panel display or user terminal.

To examine the contents of variables, simply type one or more variable names (separated by commas) on the system console device and press RETURN. The values of the variables specified will then appear on the system console.

The user can also change the values of variables by means of immediate-mode statements (on units equipped with Option 30, Program Development ROMs). System functions can also be executed in immediate mode (e.g., SIN(3)< cr> prints the value of the sine of 3 in the current trigonometric units on the system console).

REPORTING ERRORS

The 4041 reports all errors to the system console by default.

It may not always be desirable, however, to use the system console device for this purpose. For example, a programmer may wish to enter commands via a computer terminal, but have all syntax errors displayed on the front panel. This way, the user could use the editing keys on the P/D keyboard to correct syntax errors.

To make the 4041 report errors to a device other than the console, the SET SYNTAX statement must be executed. For example, to send error reports to the front panel, this statement would take the following form:

```
set syntax "frtp:"
```

For more information, see the description of the "SET SYNTAX" statement, later in this section.

REPORTING DEBUGGING INFORMATION

The 4041 sends all debugging information (i.e., messages generated during execution in "DEBUG" mode by means of the BREAK and TRACE commands) to the system console device by default.

In some cases, however, the user may wish to log debugging information on a different device, such as the thermal printer or the DC-100 tape.

To make the 4041 send debugging information to a device other than the console, the 4041 must execute a SET DEBUG statement. To send debugging information to the thermal printer, this statement would take the following form:

```
set debug "prin:"
```

For more information, see the description of the "SET DEBUG" statement, later in this section.

BREAK**The BREAK Statement**

Syntax Form: [line-no.] BREAK [numexp [,numexp] . . .]

Descriptive Form: [line-no.] BREAK [line-number [,line-number] . . .]

PURPOSE

The BREAK command sets or lists breakpoints used for debugging.

EXPLANATION

A breakpoint is a point in the program at which execution halts temporarily when the program is run in DEBUG mode. While the program is halted, the programmer can examine the contents of variables or modify the program.

If a breakpoint is encountered during execution in DEBUG mode, the line number to which the breakpoint is assigned is displayed and the program halts BEFORE that line is executed. Execution can be continued by pressing the CONTINUE key on the P/D keyboard, pressing the PROCEED key on the front panel, or by typing the command "CONTINUE" on a user terminal or the P/D keyboard (for 4041 units equipped with Option 30, Program Development ROMs).

A single breakpoint is set by typing the keyword BREAK and the line number at which the program is to be halted.

Multiple breakpoints are set by typing the keyword BREAK followed by a series of line numbers, each separated by a comma.

Typing the keyword BREAK followed by a carriage return prints the list of current breakpoints on the system console device.

Breakpoints can only be set on executable statements. Attempting to set a breakpoint on a nonexecutable statement (DATA; IMAGE; FUNCTION; REM; SUB) results in an error.

Attempting to set a breakpoint at a line that does not exist results in an error.

A breakpoint set at the first executable line of the program is not recognized the first time the line is executed. However, the breakpoint is recognized if the line is re-executed later (e.g., after a GOTO statement sending control back to the first line of the program).

Once set with the BREAK command, breakpoints can be cleared with the NOBREAK statement. (See the description of the NOBREAK statement, later in this section.)

EXAMPLES

```
break 100
```

This command sets a single breakpoint at line 100.

```
break 200,250,300,350
```

This command sets a breakpoint at line 200, line 250, line 300, and line 350 in the current program.

BREAK

```

100     A=1
110     B=2
120     C=3
130     Print a+b+c
*break 100,110,120
*break
BREAK LIST =100   110   120
*debug
Line 100

Break at line 110
*cont
Line 110

Break at line 120
*cont
Line 120
6.0
Line 130

```

(For this example, assume that the system debug device is set to its default value of system console, and the the TRACE VAR flag is set to its default value of ON.)

The first BREAK command sets breakpoints at lines 100, 110, and 120. The second BREAK command prints the list of current breakpoints on the system console device.

The DEBUG command executes the program in DEBUG mode. A message indicating that line 100 has been executed appears on the system console, followed by a message that a breakpoint has been encountered at line 110. (Note that the breakpoint at line 100 was not recognized, since line 100 is the first line of the program.)

The two ensuing CONTINUE commands resume execution in DEBUG mode after the breakpoints. The TRACE VAR flag prints the number of each line after it is executed. Note the program output appearing between two TRACE VAR messages (see the description of the TRACE command, later in this section, for more information).

CONNECT

The CONNECT Statement

Syntax and

Descriptive Forms: CONNECT [program segment]

PURPOSE

CONNECT is an immediate-mode command used to set up TRACE flags within subprograms.

NOTE

The CONNECT command is only available on 4041 units equipped with Option 30 (P/D ROMs and carrier).

EXPLANATION

TRACE information is always set up "local" to a given program segment. Thus, if a programmer is debugging in the main-program environment, the TRACE VAR ALL command would produce TRACE information about all main-program variables, but not about variables local to any subprograms or user-defined functions.

The CONNECT statement overcomes this restriction. Entering the keyword "CONNECT" followed by a program segment name effectively changes the debugging environment to that program segment. The user may then set TRACE flags affecting debugging during execution of that program segment only. (The FLOW, VAR, and VIEW flags may all be set "locally" when the 4041 is connected to a program segment environment.)

Entering the "CONNECT" command followed by a carriage return, or re-starting the program (via the "RUN" or "DEBUG" commands, the "PROCEED" key on the front panel, or the "RUN" key on the P/D keyboard) returns the 4041 to the main-program environment.

For more information, see the description of the TRACE statement, later in this section.

EXAMPLE

```

100   Var1=1
110   Call a
120   End
130 Sub a local var2
140   Var2=2
150   Return
160   End
*notrace prog
*trace var all
*debug
Line 100 var1=1.0
*connect a
*trace var all
*debug
Line 100 var1=1.0
Line 140 var2=2.0
    
```

The first TRACE VAR ALL command applies only to the main program environment. Thus, when the program is first executed in DEBUG mode, VAR1's change information is traced, but not VAR2's.

By connecting to Subprogram A and executing a TRACE VAR ALL statement, the user enables the TRACE VAR flag for all variables in Subprogram A as well as all variables for which the TRACE VAR flag was previously set. Thus, the second program execution in DEBUG mode displays change information about both VAR1 and VAR2.

The DEBUG Statement

Syntax Form: DEBUG [numexp]

Descriptive Form: DEBUG [line-number]

PURPOSE

DEBUG is an immediate-mode command that executes a program with breakpoints and TRACE flags enabled.

NOTE

The DEBUG command is only available on 4041 units equipped with Option 30 (P/D ROMs and carrier).

EXPLANATION

DEBUG functions similarly to the RUN command, except that DEBUG enables breakpoints and TRACE information, while RUN does not.

Typing DEBUG and pressing return starts execution in debug mode from the first line of the program.

Typing DEBUG followed by a numeric expression and pressing return executes the program in debug mode, starting with the line given by the numeric expression. A line used to start a program with DEBUG must be in the main program segment.

While a program started by means of a DEBUG statement is running, the 4041 is said to be in "debug" mode.

When a program is executed using the DEBUG command, execution stops at each breakpoint set by the programmer. Information called for by enabled TRACE flags is also displayed on the system debug device.

DEBUG, in addition to putting the 4041 in "debug" mode, also performs the same "housecleaning/initialization" functions as the RUN statement (described in Section 7, "Control Statements").

DELETE LINE**The DELETE LINE Statement**

| | |
|--------------------------|--|
| Syntax Form: | <pre> [line-no.] DELETE LINE {numexp} [TO numexp] {subname} [TO subname] {MAIN} </pre> |
| Descriptive Form: | <pre> [line-no.] DELETE LINE first-line-to-delete [TO last-line-to-delete] </pre> |

PURPOSE

The DELETE LINE statement deletes lines from the current program.

EXPLANATION

If only one line number is specified, DELETE LINE deletes that line. If the line does not exist in the current program, DELETE LINE has no effect.

If a range of line numbers is specified (by means of the TO keyword), all lines in the current program greater than or equal to the first line number and less than or equal to the second are deleted.

If a subprogram or user-defined function name is specified as the first argument, the entire subprogram will be deleted from the program.

If a subprogram or user-defined function name is specified as the second argument of the DELETE LINE command, the last line of the subprogram will be the last line deleted.

Delete line will NOT delete lines that cannot be legally deleted for syntax reasons (e.g., attempting to delete a SUB or FUNCTION statement without deleting the remainder of the subprogram or function). Attempting to delete lines illegally causes an error and does NOT delete any lines.

EXAMPLES

```
300      Delete line 150 to 200
```

This command deletes lines 150 through 200, inclusive, in the current program.

```
400      Delete line aaaa to bbbb
```

If aaaa and bbbb are labels, lines aaaa through bbbb, inclusive, are deleted.

If aaaa and bbbb are subprograms, aaaa and bbbb and any subprograms between them (in line-number order) are deleted.

The LIST Statement

| | | |
|--------------------------|--|-------------------|
| Syntax Form: | [line-no.] LIST [strexpr],[numexp | [TO numexp]] |
| | [subname | [TO subname]] |
| Descriptive Form: | [line-no.] LIST [stream-spec],[line-number | [TO line-number]] |
| | [subprogram | [TO subprogram]] |

PURPOSE

The LIST statement sends a list of the current BASIC program to the designated stream spec in human-readable (i.e., "pretty-printed") form.

EXPLANATION

If no stream spec is included in the LIST statement, the program is listed on the system console device.

If no line numbers or subprogram names are included in the LIST statement, the entire program is listed.

If one line number is included in the LIST statement, only that line is listed.

If two line numbers are included in the LIST statement, all program statements with line numbers greater than or equal to the first and less than or equal to the second are listed.

Subprogram names include the names of subprogram segments defined by SUB or FUNCTION statements, as well as the special keyword MAIN (indicating the MAIN program segment).

If one subprogram name is included in the LIST statement, that subprogram is listed.

If two subprogram names are included in the LIST statement, both subprograms and all intervening subprograms are listed.

If a line number TO a subprogram name is given in the LIST statement, all program lines from that line number through the last line of the subprogram are listed.

If a subprogram TO a line number is given in the LIST statement, all program lines from the first line of the subprogram through the specified line number are listed.

EXAMPLES

```
list super to 1200
```

All program lines from line Super (if Super is a line label) or the first line of subprogram Super (if Super is a subprogram) to line 1200 are listed on the system console device.

```
list "prin:",120 to super
```

All program lines from line 120 to line Super (if Super is a line label) or the last line of subprogram Super (if Super is a subprogram) are listed on the thermal printer.

NOBREAK**The NOBREAK Statement**

Syntax Form: [line-no.] NOBREAK {numexp[,numexp]...}
{ALL}

Descriptive Form: [line-no.] NOBREAK {line-number[,line-number]...}
{ALL}

PURPOSE

The NOBREAK command clears some or all of the breakpoints set using the BREAK statement.

EXPLANATION

The NOBREAK statement clears breakpoints set using the BREAK statement.

Entering NOBREAK followed by a line number clears a breakpoint at that line number.

Entering NOBREAK followed by a succession of line numbers clears breakpoints at each of those line numbers.

Entering NOBREAK followed by the keyword ALL clears all breakpoints.

Attempting to clear a breakpoint where none has been set has no effect.

EXAMPLES

```
nobreak 1000
```

This command clears a breakpoint set at line 1000.

```
nobreak 500,600,700,1000
```

This command clears breakpoints at lines 500, 600, 700, and 1000.

```
nobreak all
```

This command clears all breakpoints.

The NOTRACE Statement

| | |
|--------------------------|--|
| Syntax Form: | <pre> [[line-no.] NOTRACE {FLOW} {PROGRAM} {SUB {subname[,subname] . . .}} {ALL} {VAR {var[,var] . . .}} {ALL} {VIEW} </pre> |
| Descriptive Form: | <pre> [[line-no.] NOTRACE {FLOW} {PROGRAM} {SUB {subname[,subname] . . .}} {ALL} {VAR {var[,var] . . .}} {ALL} {VIEW} </pre> |

PURPOSE

The NOTRACE command clears some or all of the flags set by the TRACE command.

EXPLANATION

TRACE flags are used when running a program in DEBUG mode to monitor changes in values of variables, to monitor branch points within the program, to monitor the progression of the program throughout execution, and to direct information about the program to the system debug device.

TRACE flags are set by means of the TRACE command. The NOTRACE command "turns off" flags set by TRACE.

NOTRACE FLOW turns off the FLOW flag.

NOTRACE PROGRAM turns off the PROGRAM flag.

NOTRACE SUB followed by a list of subprogram names (separated by commas) turns off all TRACE information during execution of the specified subprograms or user-defined functions.

NOTRACE SUB ALL turns off the FLOW, PROGRAM, or VAR flags during execution of a specified subprogram. The VAR flag is only turned off for variables local to that subprogram; global variables whose values are changed within a subprogram designated by NOTRACE SUB ALL are still traced (i.e., information about these variables is displayed on the system debug device).

NOTRACE VAR followed by a list of variables (separated by commas) turns off the TRACE flags for those variables.

NOTRACE VAR ALL turns off the TRACE flags for all variables.

NOTRACE VIEW stops the display of TRACE information on the system debug device. This can be useful when the user only wishes to examine trace information from a section of program under development. NOTRACE VIEW only allows the "breakpoint" messages to be displayed on the system debug device; thus, execution can proceed using the "DEBUG" and "CONTINUE" commands until the desired breakpoint is reached. The TRACE VIEW command can then be used to re-display trace information.

NOTRACE

EXAMPLES

```
notrace var a,b,c,str1$,str2$
```

This command turns off the TRACE flag for the listed variables.

```
notrace sub sub1,sub2,sub3
```

This command turns off the FLOW, PROGRAM, and VAR flags (for local variables) during execution of subprograms Sub 1, Sub 2, and Sub 3. Global variables whose values are set or changed during execution of these subprograms continue to be traced.

THE REM OR ! STATEMENT

The REM Statement

Syntax Form: line-no. REM message

Descriptive Form: line-no. REM any-message-goes-here;it-is-ignored-by-the-4041

The ! Statement

Syntax Form: statement!message

Descriptive Form: statement!any-message-goes-here;it-is-ignored-by-the-4041

PURPOSE

The REM statement allows programmers to add comments ("remarks") to programs.

The ! statement functions the same way as the REM statement, except that ! statements can appear on the same lines as other statements.

EXPLANATION

REM and ! statements are used to insert comments into a program in order to clarify what the program is doing.

When the 4041 encounters the REM keyword immediately after a line identifier, the remainder of the line is ignored. The 4041 goes immediately to the next line.

Any part of a line that follows an exclamation point (unless the exclamation point is contained within a string literal) is similarly ignored. Thus, the ! statement can be used on the same line as other statements.

EXAMPLE

```
100 Y=x^2+15*x-3 !calculate y
110 If y>1000 then goto 200 !branch if y>1000
```

All text to the right of the exclamation point on each line is ignored by the system.

The RENUMBER Statement

| | |
|--------------------------|---|
| Syntax Form: | RENUMBER [numexp [,numexp]] [TO numexp[,numexp]] [subname [,subname]] [ALL] |
| Descriptive Form: | RENUMBER [beglin [,endlin]] [TO newlin[,maxinc]] [subname [,subname]] [ALL] |
| where: | beglin = first program line to be renumbered endlin = last program line to be renumbered newlin = number that beglin is renumbered to maxinc = maximum increment between renumbered lines subname = name of subprogram or user-defined function |

PURPOSE

The RENUMBER command renumbers program lines. It can also be used to transfer sections of code from one point in a program to another.

NOTE

The RENUMBER command is only available on 4041 units equipped with Option 30 (P/D ROMs and carrier).



The RENUMBER statement in 4041 BASIC may do more than the RENUMBER statement in other BASICs you may be familiar with. Make sure you read and understand RENUMBER's capabilities before attempting to use it.

EXPLANATION

The RENUMBER statement renumbers program lines. The user may specify the first and last lines to be renumbered. The user may also specify a number to which the lines will be renumbered, as well as the maximum number by which the renumbered lines are to be incremented.

The RENUMBER statement can also be used to transfer lines of code from one position in the program to another.

The possible forms of the RENUMBER statement and the results of each form are listed in Table 4-1.

Table 4-1
TABLE OF DEFAULTS FOR RENUMBER^a

| Arguments Entered | Beginning Line | Ending Line | New Line |
|-------------------|-----------------|----------------|-----------------|
| none | first line | last line | 100 |
| ALL | first line | last line | 100 |
| lineXX | lineXX | lineXX | lineXX (no-op) |
| subX | first line subX | last line subX | first line subX |
| lineXX,lineYY | lineXX | lineYY | lineXX |
| subX,lineYY | first line subX | lineYY | first line subX |
| subX,subY | first line subX | last line subY | first line subX |
| lineXX,subY | lineXX | last line subY | lineXX |
| ALL TO lineZZ | first line | last line | lineZZ |
| linXX TO linZZ | lineXX | lineXX | lineZZ |
| subX TO lineZZ | first line subX | last line subX | lineZZ |
| XX,YY TO ZZ | lineXX | lineYY | lineZZ |
| subX,YY TO ZZ | first line subX | lineYY | lineZZ |
| subX,subY TO ZZ | first line subX | last line subY | lineZZ |
| XX,subY TO ZZ | line XX | last line subY | line ZZ |

^a“MAIN” can be used as a “subprogram” name, and denotes the main program segment.

Destination Line Restrictions

If the line number of an existing line is specified as the "newline" for a RENUMBER statement, the existing line must be in the range from "beginning-line" to "ending-line" for the RENUMBER to occur.

Examples:

```

100      A=1
110      B=2
120      C=3
130      D=4
*renumber 100,110 to 130
*** ERROR # 23

```

Line 130 (newline) already exists, and was not within the range from line 100 (beginning line) to line 110 (ending line). The correct statement to move lines 100 and 110 after line 130 is

```

*renumber 100,110 to 140
*list
120      C=3
130      D=4
140      A=1
150      B=2

```

If the specified destination line does not already exist, the destination line can have any value.

Maximum Increment Calculations

The increment included in the RENUMBER statement (if one is given) specifies the maximum increment that the 4041 will use.

If the increment specified is too large (because there are too many lines to "fit" into the space provided), the 4041 calculates the maximum increment that will fit the lines into the space, using the formula:

$$\text{increment} = \text{INT}((\text{maxline} - \text{newline} + 1) / \# \text{-of-lines-to-renomber})$$

If the increment thus calculated is equal to 0, the RENUMBER operation does not take place and an error is generated. Otherwise, the RENUMBER operation is performed using the calculated increment.

Example:

```

100      ! this is line 1
110      ! this is line 2
120      ! this is line 3
130      ! this is line 4
140      ! this is line 5
150      ! this is line 6
*renumber 120,150 to 101
*list
100      ! this is line 1
101      ! this is line 3
103      ! this is line 4
105      ! this is line 5
107      ! this is line 6
110      ! this is line 2

```

Since the lines to be renumbered will not fit in the space allotted with an increment of 10, the 4041 calculates a new increment for the renumber operation.

Renumbering Line References

The RENUMBER statement renumbers line references in GOTO and GOSUB statements.

The RENUMBER statement does not renumber line references in DELETE LINE, LIST, SAVE, or APPEND statements.

Unresolved Line References

If a line to be renumbered contains an unresolved line reference, the line containing the unresolved line reference is renumbered but the reference itself remains unchanged.

Renumbered lines containing unresolved line references are displayed on the system syntax device (the device specified in the most recently executed SET SYNTAX statement; default:system console).

RENUMBER**Examples:**

In Example 4-1, line 127 contains an unresolved line reference. The new line number and the contents of the line are displayed when the line is renumbered.

Unresolved line references that become resolved by a RENUMBER operation are also displayed on the system syntax device.

In Example 4-2, line 127 contains an unresolved line reference that becomes resolved by the renumber operation. The renumbered line 127 (now, line 130) therefore appears on the system syntax device.

Unresolved line references are checked only in the portion of the program being renumbered and in the program segment to which a line or lines are being moved, except for unresolved BRANCH statements, which are checked across segment boundaries.

Restrictions on Renumbering Subprograms and User-Defined Functions

The same basic restriction applies to RENUMBERing Subprograms and user-defined functions as to appending to them and deleting lines from them: no operation may leave any portion of a Subprogram or user-defined function without a SUB or FUNCTION statement to accompany it. RENUMBER commands that would leave a portion of a subprogram without a SUB or FUNCTION statement are not performed, and cause an error.

Re-starting After RENUMBER

If a RENUMBER operation is performed when the 4041 is PAUSEd (i.e., after the PAUSE key was pressed on the front panel or P/D keyboard, or the BREAK key on the P/D keyboard or a user terminal, or after a STOP statement has been executed), the program must be re-started with a RUN or DEBUG statement and not with the CONTINUE statement.

```

96      Init
101     Integer a,b,c
105     Input prompt "enter date and time:":time$
127     Goto 300
*renumber
130 Goto 300

```

Example 4-1.

```

96      Init
101     Integer a,b,c
105     Input prompt "enter date and time:":time$
127     Goto 110
*renumber
130 Goto 110

```

Example 4-2.

The SET DEBUG Statement

Syntax Form: [[line-no.] SET DEBUG stexp

Descriptive Form: [[line-no.] SET DEBUG stream-spec

PURPOSE

SET DEBUG selects the device that debugging information (i.e., information generated by TRACE and BREAK flags) is sent to. This device becomes known as the "system debug device".

EXAMPLE

```
set debug "frtp:"
```

Information generated by TRACE and BREAK flags will appear on the front panel alphanumeric display.

EXPLANATION

The system debug device defaults to the system console. See the description of SET DEBUG in Section 5, *Environmental Control*, for more information.

The SET SYNTAX Statement

Syntax Form: [line-no.] SET SYNTAX strexp

Descriptive Form: [line-no.] SET SYNTAX stream-spec

PURPOSE

The SET SYNTAX statement selects the device to which error messages are sent. This device becomes known as the "system syntax device".

EXPLANATION

The system syntax device defaults to the system console. See the description of SET SYNTAX in Section 5, *Environmental Control*, for more information.

EXAMPLE

```
set syntax "frtp:"
```

This statement sends all syntax error messages to the front panel alphanumeric display. Once there, the line can be edited with the P/D keyboard editing keys.

When using the P/D keyboard to edit syntax errors, you may wish to delete an erroneous line and start all over again. To do this, simply press the CLEAR and RETURN keys on the P/D keyboard, or the CLEAR and PROCEED keys on the front panel.

The SLIST Statement

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] SLIST [strexpr,][numexp [TO numexp]] [subname [TO subname]] |
| Descriptive Form: | [line-no.] SLIST [stream-spec,] [line-number [TO line-number]] [subname [TO subname]] |

PURPOSE

The SLIST command lists the current BASIC program (or portions thereof) to the system console or a specified device.

EXPLANATION

The SLIST command works exactly the same as the LIST command, except it sends an un-formatted listing instead of a "pretty-printed" one to the output device.

The TRACE Statement

Syntax and

Descriptive Forms: [line-no.] TRACE [FLOW]
 [PROGRAM]
 [SUB (subname[,subname]..)]
 {ALL}
 [VAR (var[,var]..)]
 [ALL]
 [VIEW]

PURPOSE

The TRACE command sets flags used during program debugging. When a program is executed using the DEBUG command or the STEP key on the P/D keyboard, the TRACE flags are checked after each line is executed, and the appropriate action is performed.

EXPLANATION

TRACE information is only displayed when a program is run using the DEBUG command or the STEP key on the P/D keyboard. All TRACE information appears on the system debug device, unless a NOTRACE VIEW command has been executed.

TRACE FLOW displays the originating and destination line numbers whenever a branch is performed (GO TO, GO SUB, CALL, or user-defined functions, or returns from GOSUBs, Subprograms, and user-defined functions).

TRACE PROGRAM displays the line number of each line after it is executed. This flag is automatically set on power-up. TRACE PROGRAM does not trace non-executable statements (DATA, FUNCTION, IMAGE, REM, or SUB). TRACE PROGRAM is a global flag (affects all program segments).

TRACE SUB followed by a list of subprogram names causes the 4041 to trace information throughout the execution of a subprogram or user-defined function. TRACE SUB ALL causes the 4041 to trace information throughout the execution of all currently defined Subprograms and user-defined functions.

TRACE SUB ALL is set as a power-up default. The user overrides this default with NOTRACE SUB commands (see the description of the NOTRACE statement, earlier in this section). TRACE SUB is used to negate a previous NOTRACE SUB command.

TRACE SUB followed by a carriage return returns the names of all program segments for which the TRACE SUB flag is set.

TRACE VAR followed by a list of variable names causes the line number, the variable name, and the value of the variable to be displayed each time the value of one of the specified variables is changed.

TRACE VAR ALL displays this information for all variables currently defined in this environment.

The TRACE VAR flag only affects variables currently defined in the "environment" to which the 4041 is connected (see the CONNECT statement, earlier in this section, for more information). Thus, if the 4041 is connected to the MAIN program environment and a TRACE VAR ALL command is executed, all currently defined global variables are traced. If the 4041 is connected to Subprogram AAAA and a TRACE VAR ALL command is executed, all of Subprogram AAAA's currently defined local variables are traced.

TRACE VAR followed by a carriage return lists the environment to which the 4041 is connected and the names of all variables being traced in that environment.

TRACE VIEW displays TRACE information on the device specified by the SET DEBUG statement. This flag is automatically set upon power-up. The command is used to turn the flag on again after a NOTRACE VIEW command has been executed.

TRACE VIEW and NOTRACE VIEW affect only the environment to which the 4041 is connected when either command is executed.

TRACE followed by a carriage return lists the names of the enabled flags from among FLOW, PROGRAM, and VIEW.

EXAMPLES

For purposes of this example, assume that the following program has been entered, and that the TRACE flags are set to their power-up defaults. Assume also that the system debug device is the system console.

```

100   A=1
110   B=2
120   Call aaaa
130   Call bbbb
140   End
150 Sub aaaa
160   C=3 ! c is a global variable
170   D=4 ! d is a global variable
180   Return
190   End
200 Sub bbbb local e,f
210   E=5 ! e is a local variable
220   F=6 ! f is a local variable
230   Return
240   End

```

To find out which TRACE flags are set, use the TRACE command:

```

*trace
TRACE FLAGS= PROGRAM VIEW

```

To find out which variables are being traced in the current environment, use the TRACE VAR command:

```

*trace var
MAIN PROG VAR=

```

This response indicates that no variables are being traced in the main program. To trace all variables in the

main program, use the TRACE VAR ALL command:

```

*trace var all
*trace var
MAIN PROG VAR= a b c d

```

The preceding commands set the TRACE VAR flag for global variables only. To trace local variables as well (such as those in Subprogram bbbb), use the CONNECT command, followed by a TRACE VAR ALL:

```

*conn bbbb
*trace var all
*trace var
bbbb SUBVAR= e f

```

Now debug the program:

```

*debug
Line 100 a=1.0
Line 110 b=2.0
Line 120
Line 160 c=3.0
Line 170 d=4.0
Line 180
Line 130
Line 210 e=5.0
Line 220 f=6.0
Line 230
Line 140

```

The NOTRACE VAR ALL statement now turns off the TRACE VAR flag for all global variables. However, it leaves the TRACE VAR flag on for "local" variables. Similarly, the NOTRACE VIEW command turns the TRACE VIEW flag off for the main program environment, but leaves it on for Subprograms and user-defined functions.

```

*notrace var all
*notrace view
*debug
Line 160
Line 170
Line 180
Line 210 e=5.0
Line 220 f=6.0
Line 230

```

To trace program flow and execution in the main program only, use the TRACE FLOW and NOTRACE SUB commands (among others):

```

*trace view
*trace flow
*notrace sub aaaa,bbbb
*debug
Line 100
Line 110
Line 120 Branch taken to 150
Line 130 Branch taken to 200
Line 140

```

Section 5

ENVIRONMENTAL CONTROL

INTRODUCTION

The INIT and SET commands and the ASK and ASK\$ functions allow the programmer to initialize, modify, and inquire

into the status of the operating environment, all under program control.

THE ASK AND ASK\$ FUNCTIONS

The ASK Function

Syntax and

Descriptive Forms: ASK("ASK-function-name"[,argument])

where ASK-function-name = {ANGLE}
{AUTOLOAD}
{BUFFER}
{CHPOS}
{IODONE}
{KEY}
{MEMORY}
{PROCEED}
{SEGMENT}
{SPACE}
{TIME}
{UPCASE}

The ASK\$ Function

Syntax and

Descriptive Forms: ASK\$("ASK\$-function-name"[,argument])

where ASK\$-function name = {CONSOLE}
{DRIVER}
{ERROR}
{ID}
{LU}
{PATH[,ALL]}
{ROMPACK}
{SELECT}
{SELFTTEST}
{SYSDEV}
{TIME}
{VAR}
{VOLUME}

Purpose

ASK and ASK\$ are general purpose functions used to interrogate the 4041 about the state of the system.

Explanation

The ASK function returns a numeric value. The ASK\$ function returns a string. Using ASK with an ASK\$ function, or ASK\$ with an ASK function, results in a syntax error.

ASK and ASK\$ can be used either in immediate mode or within a program. When used in a program, the numeric value or string returned by the ASK or ASK\$ function may be stored in a variable.

The ASK or ASK\$ function name must be a literal string; it cannot be a string variable or expression.

ASK FUNCTIONS

ASK ("ANGLE")

**Syntax and
Descriptive Forms:** ASK("ANGLE")

ASK("ANGLE") returns an integer value indicating the current coordinate system for trigonometric functions.

ASK("ANGLE") returns

- 0 if the current coordinate system is radians
- 1 if the current coordinate system is degrees
- 2 if the current coordinate system is grads

Example:

```
110 Trig=ask("angle")
```

This statement stores a value indicating the current coordinate system for trigonometric functions in numeric variable Trig.

ASK ("AUTOLOAD")

**Syntax and
Descriptive Forms:** ASK("AUTOLOAD")

ASK("AUTOLOAD") returns an integer value that tells whether or not the system AUTOLOAD function is enabled.

ASK("AUTOLOAD") returns

- 1 if the system AUTOLOAD function is enabled
- 0 if the system AUTOLOAD function is disabled

The 4041 powers up with the AUTOLOAD function enabled. The user invokes the AUTOLOAD function by pressing the appropriate key on the system console device: the AUTOLOAD key on the front panel or P/D keyboard, or CTRL-V on a computer terminal connected to the 4041 through an RS-232-C interface port.

Invoking the AUTOLOAD function loads and runs a file named "AUTOLD" from the DC-100 tape.

Example:

```
120 Atld_e=ask("autoload")
```

After this statement is executed, numeric variable AtldE will contain a value (1 or 0) indicating whether or not the AUTOLOAD function is enabled.

ASK ("BUFFER")

| | |
|--------------------------|-------------------------------------|
| Syntax Form: | ASK("BUFFER"[,numexp]) |
| Descriptive Form: | ASK("BUFFER"[,logical-unit-number]) |

ASK("BUFFER") returns the starting position of unused data in a buffer string after execution of a GETMEM or INPUT statement that includes a BUFFER clause.

The returned value reflects the point in the string where the internal buffer pointer was positioned when all variables in the variable list were satisfied. This value can be used in a string function to discard all used data from the buffer variable before executing a subsequent GETMEM statement to extract additional data from the buffer.

If a numeric variable was the last list element to be satisfied, then the buffer position returned is the position of the last character of the ASCII string representing the numeric value (if the input list was satisfied), or the position of the numeric delimiter character that terminated input.

If a string variable was the last list element to be satisfied, then the buffer position returned is the position of the last character placed in the string (if the string was completely filled), or the position of the delimiter character that terminated input.

The programmer may also specify a logical unit number for the request by using the form ASK("BUFFER",lunum). In this case, the function returns the starting position of unused data in a buffer string after execution of the last GETMEM or INPUT statement with the logical unit number included in the BUFFER clause.

If the logical unit is not open, the value 0 is returned.

Example:

```
110 Buff$="1,2,3,4,5,6,7,8,9,10"  
120 Integer num(5),num2(5)  
130 Getmem buffer buff$:num  
140 Buf2$=seg$(buff$,ask("buffer")+1,20)  
150 Getmem buffer buf2$:num2  
160 Print "NUM1=";num  
170 Print "NUM2=";num2  
180 End  
*run  
NUM=1 2 3 4 5  
NUM2=6 7 8 9 10
```

Line 130 reads the first five numeric values from Buff\$ into array variable Num.

The Ask("Buffer") function returns the starting position of unused data from Buff\$ (in this case, 10).

Line 140 reads 20 characters (or characters up to the end of the source string) into Buf2\$, starting with the 11th character of Buff\$.

Line 150 reads five numeric values from Buf2\$ into array variable Num2.

ASK ("CHPOS")

Syntax and**Descriptive Forms:** ASK("CHPOS")

ASK ("CHPOS") returns the position of the last character scanned by a VAL or VALC function. This is the position of the first non-numeric character after the number scanned. The value can be one more than the length of the last string scanned by VAL or VALC (indicating that the entire string was scanned). A value of 0 indicates that the VAL/VALC functions have not been invoked.

Example:

```

100 String$="JOHN SMITH,11,85"
110 Integer grade,score
120 Getmem buffer string$ dels ",":name$
130 Grade=val(string$)
140 Score=valc(string$,ask("chpos")+1)
150 Print using 180:"NAME:",name$
160 Print using 180:"GRADE:",grade
170 Print using 180:"SCORE:",score
180 Image 6a,n
190 End
*run
NAME: JOHN SMITH
GRADE:11
SCORE:85

```

Line 120 reads all characters up to the first comma in String\$ into Name\$.

Line 130 reads the first legal numeric value in String\$ into numeric variable Grade.

The Ask("chpos") function then returns the position of the last character scanned by a VAL or VALC function; in this case, the value returned is 14.

Line 140 reads the first legal numeric value found in String\$, starting with the character after the last character scanned by a VAL or VALC function, into numeric variable Score.

ASK ("IODONE")

Syntax Form: ASK("IODONE"[,numexp])

Descriptive Form: ASK("IODONE"[,logical-unit-number])

ASK("IODONE", logical-unit-number) returns an integer value reflecting the status (busy or not busy) of a particular logical unit. ASK("IODONE") returns an integer value reflecting the status of the entire 4041 I/O system.

ASK("IODONE", logical-unit-number) returns:

- 1 if the logical unit is NOT busy performing I/O
- 0 if the logical unit IS busy performing I/O
- 1 if the logical unit is not open

ASK("IODONE") returns:

- 0 if I/O is active on some driver
- 1 if no I/O is active on any driver

The ASK("IODONE") functions are usually used when the 4041 is in proceed mode, to ensure that I/O operations on a given device are complete before proceeding to a new section of code.

See Section 8, *Input/Output*, for more information about using INPUT and PRINT in proceed-mode.

Example:

In Example 5-1, line 300 checks to see that the proceed-mode I/O operation on logical unit 1 is complete before jumping to line "GoAhead". If it isn't, the 4041 waits 5 seconds, then tries again.

```
100 Set proceed 1
110 Open #1:"outfil(open=new,size=20000)"
.
.
200 Print #1:bigaray1,bigaray2,bigaray3
.
.
300 If ask("iodone",1)=1 then goto goahead else wait 5
310 Goto 300
.
.
500 Goahead:!execution continues from here
```

Example 5-1.

ASK ("KEY")

**Syntax and
Descriptive Forms:** ASK("KEY")

ASK("KEY") returns the number of the next user-definable function awaiting service. A value of 0 indicates that no user-definable function is awaiting service.

When the user-definable functions are disabled, the 4041 queues one user-definable key, which is executed the next time user-definable functions are enabled.

The system powers up with user-definable functions disabled. User-definable functions are automatically disabled when the system executes a function key handler, and automatically re-enabled when the function key handler completes execution.

For more information about user-definable functions, see Section 12, *Interrupt Handling*.

ASK ("MEMORY"[,ALL])

**Syntax and
Descriptive Forms:** ASK("MEMORY"[,ALL])

ASK("MEMORY") returns the size in bytes of the largest free block in memory.

ASK("MEMORY",ALL) returns the amount in bytes of all free space in memory.

ASK("PROCEED")

**Syntax and
Descriptive Forms:** ASK("PROCEED")

ASK("PROCEED") returns an integer reflecting the current setting of the system proceed mode parameter.

ASK("PROCEED") returns:

- 0 if the 4041 is NOT operating in proceed mode
- 1 if the 4041 IS operating in proceed mode

The 4041 is NOT in proceed mode at power-up.

The 4041 is put into proceed mode when a SET PROCEED 1 statement is executed.

The 4041 returns from proceed mode when a SET PROCEED 0 statement is executed.

ASK("SEGMENT")

**Syntax and
Descriptive Forms:** ASK("SEGMENT"[,ALL])

ASK("SEGMENT") returns an integer value giving information about the program segment currently executing.

Segments indicated by different values of ASK("SEGMENT") are as follows:

- 1 main program, subprogram, or function
- 2 error handler
- 3 function key handler
- 4 GPIB function handler
- 5 ABORT handler
- 6 IODONE handler

ASK("SEGMENT",ALL) returns a bit-encoded value giving information about all program segments currently executing. The rightmost bit of this value (the 1's place in its binary representation) is always 0. Other components of this value are as follows:

- 2 error handler
- 4 function key handler
- 8 GPIB function handler
- 16 ABORT handler
- 32 IODONE handler

Example:

```
1050 Print ask("segment",all)
```

Suppose a value of 44 is printed when line 1050 is executed. Since $44 = 32 + 8 + 4$, this indicates that an IODONE handler, GPIB function handler, and function key handler are active. (A simple way to determine which handlers are active is to print the result of the ASK("SEGMENT",ALL) function using an image operator of "6B".)

ASK("SPACE")

**Syntax and
Descriptive Forms:** ASK("SPACE")

ASK("SPACE") returns an estimate of the amount of memory, in bytes, required to save the current program in ASCII.

(The estimate equals 72 times the number of lines in the program.)

ASK("TIME")

**Syntax and
Descriptive Forms:** ASK("TIME")

ASK("TIME") returns a real number that represents the time in seconds since power-up.

(Be careful not to confuse this function with ASK\$("TIME"), which returns the date and time.)

ASK("UPCASE")

**Syntax and
Descriptive Forms:** ASK("UPCASE")

ASK("UPCASE") returns an integer reflecting the status of the system UPCASE parameter.

When the UPCASE parameter is set, lower-case letters are considered equal to upper-case letters for string comparisons.

ASK("UPCASE") returns

The 4041 powers up with the UPCASE parameter set.

- 1 if the system UPCASE parameter is set
- 0 if the system UPCASE parameter is NOT set

ASK\$ FUNCTIONS

ASK\$("CONSOLE")

**Syntax and
Descriptive Forms:** ASK\$("CONSOLE")

ASK\$("CONSOLE") returns a string containing the stream spec of the system console device.

This string can be saved in a string variable and analyzed (using string functions) for information.

Example:

Assume that the last SET CONSOLE statement executed was

```
Set console "comm:"
```

(setting the standard RS-232-C interface port to be the system console device).

Then, ASK\$("CONSOLE") returns a string as shown in Example 5-2.

```
COMMO(BAU=2400, IBA=0, BIT=8, PAR=NO, STO=2, FOR=ASC, TYP=1.00000E+2,  
#TY=0.00000E+0, FLA=OUT, DSR=ON, CTS=ON, DCD=ON, RTS=OFF, DTR=OFF, ERR=  
LOG, #ER=0.00000E+0, ECH=YES, CON=NO, EDI=RAS, CR=CRL, LF=LF, TIM=  
2.14748E+7, TMS=2.14748E+7, EOA=<0>, EOH=<0>, EOM=""):
```

Example 5-2.

ASK\$(“DRIVER”)

**Syntax and
Descriptive Forms:** ASK\$(“DRIVER”)

ASK\$(“DRIVER”) returns a string containing the names of all device drivers in the system.

ASK\$(“ERROR”)

**Syntax and
Descriptive Forms:** ASK\$(“ERROR”)

ASK\$(“ERROR”) returns a string containing information about an error currently being handled. This information includes:

1. The error number.
2. The line number on which the error occurred.
3. The logical unit number on which the error occurred.
4. The number of times the same error occurred without a different error intervening (useful to indicate situations where an error-handler isn't taking care of the real cause of an error).

A value of “-1” in the logical unit's position indicates that the error is not associated with a logical unit.

For proceed-mode errors (error #999), a string of eight numbers separated by commas is returned. The first four numbers are “999,0,0,0,” followed by a string containing four numbers as described above.

Example:

```
1060 Print ask$("error")
```

Suppose the message printed in response to line 1060 is

```
999,0,0,0,854,130,1,1
```

This indicates that error #854 (write-after-read) occurred during proceed-mode execution of line 130. The error occurred during I/O involving logical unit 1. This is the first time this error has occurred since either the start of the program or a different error.

ASK\$("ID")

**Syntax and
Descriptive Forms:** ASK\$("ID")

ASK\$("ID") returns a string containing the 4041's ID information.

This information includes the manufacturer (Tektronix, naturally), instrument identification number, version of the Tektronix Codes and Formats standard to which the firmware conforms, and firmware version number.

Example:

```
1000 Print ask$("id")
```

This statement prints a string of the form

```
ID TEK/4041,V79.1,2.0
```

on the system console device, indicating that the instrument is a TEKTRONIX 4041 supporting version 79.1 of the Tektronix Codes and Formats standard, and having firmware version 2.0.

ASK\$("LU")

Syntax Form: ASK\$("LU",numexp)
Descriptive Form: ASK\$("LU",logical-unit-number)

ASK\$("LU") returns a string containing the complete stream spec for a specified logical unit.

A null string is returned if the logical unit is not open.

Example:

Suppose logical unit 24 is opened to the device at primary address 24 on the standard GPIB interface port (all other GPIB parameters are at their default values). Then, the statement

```
1100 Print ask$("1u",24)
```

prints the information shown in Example 5-3 on the system console.

```
GPIB0(MA=30,SC=YES,CIC=5,TL=0,ENA=0,PEN=0,PRI=24,SEC=32,  
EOA=<44>,EOH=<32>,EOM=<255>,EOQ=<0>,EQU=;,TIM=2.14748E+7,  
SPE=1.00000E-2,TRA=NOR,PNS=0,IST=FAL,SRQ=0,TC=SYN):
```

Example 5-3.

ASK\$("PATH")

Syntax and

Descriptive Forms: ASK\$("PATH"[,ALL])

ASK\$("PATH") returns a string containing the sequence of subprogram names currently executing. This string is of the form "SubC,SubB,SubA,MAIN,...", where MAIN indicates the MAIN program and SubA, SubB, SubC, etc., are subprogram names.

ASK\$("PATH",ALL) returns a string containing information about the entire current activation path. This string contains the names of all subprograms and user-defined functions currently executing, along with the numbers of the lines that called them into execution. The string also contains the line numbers of GOSUB statements currently in the activation path.

In addition, the string contains information about all condition handlers currently being executed; entries denoting condition handlers are preceded by an appropriate letter followed by a dash, as follows:

| Letter | Condition |
|--------|---------------------------------------|
| A | Abort handler |
| E | Error handler |
| K | Function key handler |
| P | Iodone handler |
| I | Other interrupt (e.g., GPIB function) |

Example:

In Example 5-4, the string returned by the ASK\$("PATH",ALL) function indicates that the current activation path is as follows (reading from right to left):

1. The path starts with the main program segment (this is always true).
2. Line 170 in the main program segment transferred control to a subprogram or user-defined function called Printit.
3. Control transferred to a gosub-type function key handler after line 320 of subprogram/function Printit was executed.
4. Control was transferred to a call-type abort handler (subprogram Abohnd) during execution of line 230.
5. Line 410 of subprogram Abohnd was a GOSUB statement.
6. After line 470 of the subroutine was executed, control was transferred to a call-type iodone handler (subprogram Donhnd).
7. Execution of line 510 caused control to transfer to a gosub-type error handler.

```

100 Print ask$("path",all)
E-510,P-470 donhnd,410,A-230 abohnd,K-320,170 printit,main

```

Example 5-4.

ASK\$("ROMPACK")

**Syntax and
Descriptive Forms:** ASK\$("ROMPACK")

ASK\$("ROMPACK") returns a string containing the names of all ROMpacks in the system.

ASK\$("SELECT")

**Syntax and
Descriptive Forms:** ASK\$("SELECT")

ASK\$("SELECT") returns a string containing the currently SELECTed stream spec. If no SELECT statement has been

executed and no primitive I/O operation has been performed, a null string is returned.

ASK\$("SELFTEST")

**Syntax and
Descriptive Forms:** ASK\$("SELFTEST")

ASK\$("SELFTEST") returns a string containing the result of the last self-test performed (either upon power-up or as a

result of an INIT SELFTEST command).

ASK\$("SYSDEV")

**Syntax and
Descriptive Forms:** ASK\$("SYSDEV")

ASK\$("SYSDEV") returns a string containing the current SYSDEV driver spec. See the description of the SET

SYSDEV statement, later in this section, for more information about the SYSDEV driver spec.

ASK\$("TIME")

**Syntax and
Descriptive Forms:** ASK\$("TIME")

ASK\$("TIME") returns a string giving the date and time of day. This date and time will only be accurate if a SET TIME statement has been executed (see the description of the SET TIME statement later in this section). Otherwise, the date and time will be shown as the time since the 4041 was powered up, with power-up time considered to be "01-JAN-81 00:00:00".

Be careful not to confuse this function with the ASK("TIME") function, which returns the time in seconds since the 4041 was powered up.

ASK\$('VAR')

Syntax and**Descriptive Forms:** ASK\$('VAR',variable-name)

ASK\$('VAR',variable-name) returns a string containing information about the variable requested. Information returned includes a type code, an attribute code, number of rows (array variables), number of columns (array variables), dimensioned length (string variables and string arrays), and current length (string variables). If a field is not applicable, a value of 0 is returned.

Values returned for the type code are:

- 0 short floating point scalar
- 0L label
- 1 long floating point scalar
- 2 integer scalar
- 3 string scalar
- 4 short floating point array
- 5 long floating point array
- 6 integer array
- 7 string array
- 8 subprogram
- 9 function

Attribute code values returned are:

- 1 defined
- 0 undefined.

Specifying a single array element in an ASK\$('VAR') function returns information about the entire array. For example, ASK\$('VAR',Number(1)) returns the same information as ASK\$('VAR',Number).

Invoking the ASK\$('VAR') function prevents any function subprograms from executing in the same line. Attempting to invoke a function subprogram in the same line as an ASK\$('VAR') function results in an error.

Example:

```
100 Long a(20)
110 Print ask$("var",a)
```

The "5" indicates that A is a long floating point array; the first "1" indicates that the array is defined; the "20" indicates that the array has 20 rows; the second "1" indicates that the array has one column.

The two "0's" are not applicable to A, indicating the dimensioned and current lengths of a string variable.

ASK\$('VOLUME')

Syntax and**Descriptive Forms:** ASK\$('VOLUME')

ASK\$('VOLUME') returns the volume ID (label) of the current DC-100 tape. A null string is returned if no tape is in the

DC-100 tape drive.

The INIT Statement

```
Syntax and  
Descriptive Forms: [line-no.] INIT      [VAR {var[,var]...}]  
                                     {ALL}  
                                     [ALL]  
                                     [SELFTEST]
```

PURPOSE

The INIT command initializes system environmental parameters. INIT VAR initializes all variables (including undefined ones). INIT ALL performs an INIT followed by an INIT VAR. INIT SELFTEST re-performs the self-testing the 4041 does upon power-up.

WARNING

The INIT SELFTEST command deletes all program lines and variables from memory. Do not execute this command without first storing any required programs or data on tape.

EXPLANATION

If the INIT command is executed in the main program segment, the command has effect over the entire program. If INIT is executed within a subprogram segment, the command only has effect within that segment (except for closing logical units and re-setting interrupt conditions, which always have global effect).

INIT performs the following functions:

- Clears pending operations in current program segment (FOR.. NEXT loops, GOSUBs)
- Closes open logical units
- Closes open files
- Deletes handler linkages
- Resets ENABLE and DISABLE on all conditions to their power-up defaults
- Resets the ANGLE, AUTOLOAD, FUZZ, and UPCASE parameters to their power-up defaults

- Resets DATA statement pointer in current program segment
- Clears pending interrupts
- Clears "last error" values and repetition count values on ASK\$("ERROR") function
- Clears last result from VAL function
- Takes 4041 out of proceed mode (if in proceed mode)
- Clears function key queue
- Clears front panel display

INIT VAR followed by one or more variable names sets the value of any specified numeric variables to 0, and that of any specified string variables to "null".

Defined variables retain their type and dimensions.

Undefined and deleted variables are set to default types (short-floating-point scalars for numeric variables, string-scalars for string variables).

INIT VAR followed by a label name initializes the label, giving it a value of 0.

INIT VAR ALL sets the values of all numeric variables in memory (including undefined ones) to 0, and the values of all string variables in memory (including undefined ones) to null.

INIT VAR ALL has no effect on deleted variables.

INIT VAR ALL has no effect on labels.

INIT ALL performs an INIT followed by an INIT VAR ALL.

INIT SELFTEST erases the program and all variables in memory, and re-performs the 4041's power-up self-test.

The SET Statement

| | | |
|--------------------------|-----------------|--|
| Syntax Form: | [[line-no.] SET | {ANGLE numexp} {AUTOLOAD numexp} {CONSOLE strexp} {DEBUG strexp} {DRIVER strexp} {FUZZ numexp,numexp,numexp,numexp} {PROCEED numexp} {SYNTAX strexp} {TIME strexp} {UPCASE numexp} |
| Descriptive Form: | [[line-no.] SET | {ANGLE value of ANGLE parameter} {AUTOLOAD value of AUTOLOAD parameter} {CONSOLE stream spec of system console device} {DEBUG stream spec of system debug device} {DRIVER stream spec giving physical parameters for GPIB or COMM driver} {FUZZ number of digits to compare for short floating point, number to consider zero for short floating point, number of digits to compare for long floating point, number to consider zero for long floating point} {PROCEED value of PROCEED parameter} {SYNTAX stream spec of system syntax device} {TIME "DD-MMM-YY HH:MM:SS"} {UPCASE value of UPCASE parameter} |

PURPOSE

The SET command changes features of the 4041's programming environment.

SET ANGLE

Syntax Form: [[line-no.] SET ANGLE numexp

Descriptive Form: [[line-no.] SET ANGLE current coordinate system for trigonometric functions

SET ANGLE chooses a coordinate system for trigonometric functions. The ANGLE parameter takes a value of 0 for radians, 1 for degrees, and 2 for grads. The default is 0 (radians).

The numeric expression in the SET ANGLE statement is rounded to the nearest integer. A result other than 0, 1, or 2 results in an error.

SET AUTOLOAD

Syntax Form: [[line-no.] SET AUTOLOAD numexp

Descriptive Form: [[line-no.] SET AUTOLOAD current setting of AUTOLOAD function

SET AUTOLOAD sets the AUTOLOAD parameter, which enables or disables the 4041's AUTOLOAD function. An AUTOLOAD parameter value of 1 enables the AUTOLOAD function, while an AUTOLOAD parameter value of 0 disables it.

The numeric expression in the SET AUTOLOAD statement is rounded to the nearest integer. A result other than 0 or 1 results in an error.

The user invokes the AUTOLOAD function by pressing appropriate keys on the system console device: either the AUTOLOAD key on the front panel or P/D keyboard, or the CTRL-V keys on a computer terminal attached to the 4041 via an RS-232-C interface port.

The AUTOLOAD function clears memory, then loads and runs a file called "AUTOLD" from the DC-100 tape. If no tape is in the drive, or if the tape does not contain an "AUTOLD" file, the AUTOLOAD function has no effect.

SET CONSOLE

| | |
|--------------------------|---|
| Syntax Form: | [line-no.] SET CONSOLE strexp |
| Descriptive Form: | [line-no.] SET CONSOLE stream spec of system console device |

The SET CONSOLE statement specifies the stream spec for the system console device.

The 4041 powers up with the front panel as the system console device.

The SET CONSOLE statement is used to designate another driver as the system console device. COMM0 (and COMM1, in systems configured with Option 1) are legal system console devices in addition to FRTP.

NOTE

The SET CONSOLE statement affects only logical parameters of the system console device. Physical parameters (while accepted within the stream spec) are not affected. Physical parameters can only be changed by means of the SET DRIVER statement.

After a SET CONSOLE statement is executed, the system debug and system error devices revert back to the system console device, if no SET DEBUG or SET SYNTAX statement has been executed previously. If the user intends that

devices other than the console be the system debug and syntax devices, a SET DEBUG or SET SYNTAX statement must be executed.

Example:

```
200 Set console "comm0(edi=sto):"
```

This command makes the standard RS-232-C interface port the system console device, and sets the "EDIT" logical parameter to "Storage" (echoes "Rubout" as a backslash-character stringbackslash sequence).

```
200 Set console "frtp(vie=3, rat=0.75):"
```

The front panel and P/D keyboard are made the system console device. Lines appearing on the front panel will be displayed for three seconds. Lines longer than 20 characters will "scroll" across the display at the rate of one character every 3/4 second.

SET DEBUG

| | |
|--------------------------|----------------------------------|
| Syntax Form: | [line-no.] SET DEBUG strexp |
| Descriptive Form: | [line-no.] SET DEBUG stream-spec |

SET DEBUG selects the device that debugging information (i.e., information generated by TRACE and BREAK flags) is sent to.

The SET DEBUG stream spec defaults to the system console device upon power-up. After a SET DEBUG statement has been executed, subsequent SET CONSOLE statements will not change the system debug device.

Example:

```
Set debug "frtp:"
```

Information generated by TRACE and BREAK flags will appear on the front panel alphanumeric display.

SET DRIVER

Syntax Form: [line-no.] SET DRIVER strexp

Descriptive Form: [line-no.] SET DRIVER stream-spec

The SET DRIVER statement sets physical parameters for the GPIB, COMM, and OPT2 drivers.

Physical parameters for the GPIB0 and GPIB1 drivers include:

- My Address (MA)
- System Controller (SC)
- Polled With Nothing To Say (PNTs)

For more information about these parameters, see Section 9, *Instrument Control With GPIB*, and Appendix D.

Physical parameters for the COMM0 and COMM1 drivers include:

- Baud Rate (BAUd)
- Integer Baud (IBAUd)
- No. of bits transmitted (BITs)
- Parity (PARity)
- No. of stop bits (STOp)
- ASCII or Item format (FORmat)
- Size of typeahead buffer (TYPE)
- Type of flagging in effect (FLAg)
- Data Set Ready Modem Control Line (DSR)
- Clear To Send Modem Control Line (CTS)
- Data Terminal Ready Modem Control Line (DTR)
- Error Reporting (ERR)

For more information about these parameters, see Section 10, *RS-232-C Data Communications*, and Appendix D.

Physical parameters for the OPT2 driver (available on 4041 units equipped with Option 2) include:

- IREG: the number of the register to be written into to turn off the "SRQ" interrupt.
- IVAL: the value to be sent to the IREG register to turn off the "SRQ" interrupt.

For more information on these parameters and on the OPT2 driver, see Section 8 *Input/Output*, Section 12 *Interrupt Handling*, and Appendix D *Stream Specifications*.

Physical parameters for the GPIB0, GPIB1, COMM0, and COMM1 drivers can only be adjusted via the SET DRIVER statement. Although physical parameters may appear in stream specs as part of other statements, the values given for those parameters will be ignored.

Similarly, although logical parameters may appear in the stream spec following the SET DRIVER keywords, logical parameter values given will have no effect.

NOTE

The SET DRIVER statement has different effects on physical parameters not included in the stream spec when executed on the COMM, GPIB, and OPT2 drivers.

When the SET DRIVER statement sets physical parameters for one of the COMM drivers, all physical parameters not included in the stream spec given in the SET DRIVER statement ARE SET TO THEIR DEFAULT VALUES.

When the SET DRIVER statement sets physical parameters for one of the GPIB drivers or the OPT2 driver, all physical parameters not included in the stream spec given in the SET DRIVER statement RETAIN THEIR PREVIOUS VALUES.

ENVIRONMENTAL CONTROL
FUZZ

Example:

```
200 Set driver "gpib(ma=29,sc=no):"
```

The 4041 is assigned a primary address of 29 on the bus connected to the standard GPIB interface port. The 4041 is also told that it is not the system controller.

```
400 Set driver "opt2(ireg=5,ival=7):"
```

This example sets the OPT2 driver's IREG parameter to 5, and the IVAL parameter to 7. This means that, when a

device connected to the Option 2 interface port generates an interrupt, the 4041 will send a value of 7 to the device's register number 5 to turn the interrupt off.

In Example 5-5, the device attached to the optional RS-232-C interface port will communicate with the 4041 at a baud rate of 4800, transmitting 7-bit characters with even parity. When communicating with this device, the 4041 will have a typeahead buffer of 500 bytes. All other physical parameters are set to their default values.

```
300 Set driver "comm1(bau=4800,bit=7,par=eve,typ=500):"
```

Example 5-5.

SET FUZZ

Syntax Form: [line-no.] SET FUZZ numexp,numexp,numexp,numexp

Descriptive Form: [line-no.] SET FUZZ number of digits to compare for short floating point,
number to consider zero for short floating point,
number of digits to compare for long floating point,
number to consider zero for long floating point

SET FUZZ sets the number of digits to be compared and the number to be considered equal to zero for comparisons involving short floating point and long floating point numbers.

SET FUZZ takes four arguments.

Argument 1 is the approximate number of decimal digits used for comparisons of regular floating point numbers. (This number is approximate because the comparison of a decimal number with the binary form in which that number is stored is not always exact.)

Argument 2 is the number to consider equal to zero when comparing short floating point numbers. This argument must be within the range of short floating point numbers (maximum: $\pm 3.40282E38$; minimum: $\pm 2.93874E-39$). Any short floating point number whose absolute value is less than or equal to the absolute value of this argument is considered equal to zero.

Argument 3 is the approximate number of decimal digits used for comparisons of long floating point numbers.

Argument 4 is the number to consider equal to zero when comparing long floating point numbers. This argument must be within the range of long floating point numbers (maximum: $\pm 1.7976931348623E308$; minimum: $\pm 5.562684646269E-309$). Any long floating point number whose absolute value is less than or equal to the absolute value of this argument is considered equal to zero.

The power-up defaults for SET FUZZ are: 6,1E-14,14,1E-64.

When comparing numbers of "unequal" precision (e.g., an integer with a regular floating point number, or a regular floating point with a long floating point), the less-precise number is always converted to the more-precise form before comparison. The FUZZ parameter for the more-precise number representation is then used to determine the outcome of the comparison.

Example:

In Example 5-6, line 100 sets the FUZZ parameters as follows: for regular floating point comparisons, five digits of each number are compared, and any number within the range -1E-16 to +1E-16 is considered equal to zero. For long floating point comparisons, ten digits are compared, and any number within the range -1E-20 to +1E-20 is considered equal to zero.

Lines 110 and 120 set the values of short floating point variables A and B to 3.14159 and 3.14158, respectively.

Line 130 compares A and B and prints a message, depending on whether A is equal to B. (Since only the first five digits of short floating point numbers are to be compared, A is equal to B in this case.)

Line 140 allocates memory space for a long floating point variable C.

Line 150 sets the value of C to 7.5E-21.

Line 160 compares C to zero, and prints a message, depending on whether or not C is equal to zero. (Since any number whose absolute value is less than or equal to 1E-20 is to be considered equal to zero, C is equal to zero in this case.)

```

100 Set fuzz 5,1E-16,10,1E-20
110 A=3.14159
120 B=3.14158
130 If a=b then print "A=B" else print "A<>B"
140 Long c
150 C=7.5E-21
160 If c=0 then print "C=0" else print "C<>0"
170 End
*run
A=B
C=0
*
```

Example 5-6.

SET PROCEED

Syntax Form: [line-no.] SET PROCEED numexp

Descriptive Form: [line-no.] SET PROCEED {1}
{0}

The SET PROCEED statement puts the 4041 into or out of proceed mode.

When the 4041 is in proceed mode, program statements continue to execute after an INPUT or PRINT statement invokes an I/O driver, and while the I/O is being performed.

When the 4041 is not in proceed-mode, program statements execute strictly sequentially, i.e., an INPUT or PRINT operation must be completed before the next statement in the program executes.

The 4041 is NOT in proceed mode after power-up. Executing an INIT, RUN, DEBUG, or DELETE ALL statement also takes the 4041 out of proceed mode.

The numeric expression given after the SET PROCEED keywords must evaluate to a real number rounding to 1 or 0. Any other value results in an error.

Example:

```
300 Set proceed 1
```

This statement puts the 4041 into proceed-mode.

```
400 Set proceed 0
```

This statement takes the 4041 out of proceed-mode.

SET SYNTAX

Syntax Form: [line-no.] SET SYNTAX strexp

Descriptive Form: [line-no.] SET SYNTAX stream-spec

The SET SYNTAX statement selects the device to which system error messages are sent.

SET SYNTAX defaults to the system console device upon power-up.

After a SET SYNTAX statement has been executed, subsequent SET CONSOLE statements do not affect the setting of the system syntax device.

Example:

```
Set syntax "frtp:"
```

This statement sends all system error messages to the front panel alphanumeric display. Syntax errors sent to the front panel can be corrected using the P/D keyboard's editing keys.

SET SYSDEV

| | |
|--------------------------|-----------------------------------|
| Syntax Form: | [line-no.] SET SYSDEV strexp |
| Descriptive Form: | [line-no.] SET SYSDEV driver-spec |

SET SYSDEV sets the default driver spec used in subsequent stream specs that only specify a file spec.

Any TAPE parameter values are legal within the driver specification except OPEn = REPlace, OPEn = UPDate, and PHYsical = YES.

The user may override any SYSDEV parameter value by specifying a different value.

The driver specification given in the SET SYSDEV statement must terminate with a colon.

The SYSDEV driver spec powers up to "TAPE:" by default.

Examples of the use of SET SYSDEV are to enable the user to get long-form tape directories by default, or to suppress directory modification or read-after-write verification in order to save time.

NOTE

The word "SYSDEV" is still a legal name for variables and subprograms in 4041 BASIC.

Example:

```
Set sysdev "tape(lon=yes):"
```

Sets the default value of the LON parameter to "YES". Results in long-form directories when the "DIR" command is executed.

```
Set sysdev "tape(ver=no,cli=yes,lon=yes):"
```

Suppresses read-after-write verification; automatically clips files when closed; sets default value of LON parameter to "yes" (results in long-form directories when the DIR statement is executed).

```
Set sysdev spec$
```

Sets the SYSDEV stream spec to the contents of string variable Spec\$.

SET TIME

Syntax Form: [line-no.] SET TIME stexp

Descriptive Form: [line-no.] SET TIME day:month:year hours:minutes[:seconds]

SET TIME sets the date-and-time parameter for the 4041. The argument following the SET TIME keywords must be a string of the form "DD-MMM-YY HH:MM:SS".

To enter the month into the date portion of the date-and-time string, use the first three letters of that month's name (JAN for January, FEB for February, etc), or use the number of the month (1 for January, 2 for February, etc.). Regardless of which way the month is entered, the ASK\$ ("TIME") function returns the three-letter form.

The TIME parameter uses a 24-hour clock; 2:00 p.m., for example, is entered in the time portion of the date-and-time string as "14:00:00". Seconds are optional, and default to ":00". Midnight is entered as "00:00:00".

The 4041 automatically updates the date portion of the date-and-time string whenever "00:00:00" is sensed in the time portion of the date-and-time string after "23:59:59" is encountered. The 4041 takes leap years into account for changing the date.

If no SET TIME command is executed, date and time are measured from the time the 4041 powers up, with the moment of power-up receiving the date and time "01-JAN-81 00:00:00". It is a good idea to set the date and time whenever the 4041 is powered up.

The system verification tape includes a step to set the date and time; another way to guarantee that date and time are set whenever the 4041 is powered up is to create an "AUTOLD" file on a DC-100 tape with instructions prompting the user to set a date and time. This file is run whenever the user presses the AUTOLOAD key on the front panel or P/D keyboard (or CTRL-V on a user terminal, if "COMMO:" is the system console device).

Example:

```
2000 Set time "6-JAN-82 14:45:00"
```

This command sets the date and time to 2:45 p.m., January 6th, 1982. This parameter is automatically updated by the 4041 as long as it remains powered up; thus, the response to the function ASK\$("TIME") five minutes after time had been set by the above command would be "6-JAN-82 14:50:00".

SET UPCASE

Syntax Form: [line-no.] SET UPCASE numexp

Descriptive Form: [line-no.] SET UPCASE {1}
{0}

SET UPCASE determines whether or not upper case letters are considered equal to lower case letters for string comparisons.

The UPCASE parameter takes a value of 0 for false (upper case NOT equal to lower case), and 1 for true (upper case equal to lower case). The default is 1 (true). Attempting to assign a value to the UPCASE parameter that does not round to 0 or 1 results in an error.

Example:

NOTE

Example 5-7 requires the use of either the P/D keyboard or a computer terminal capable of producing both upper case and lower case characters.)

In Example 5-7, line 100 sets the UPCASE parameter to 0. When the program branches to line 200, "ABC" is not considered equal to "abc", and the message "NOT EQUAL" is printed on the system console device.

Line 120 sets the UPCASE parameter to 1. When the program branches to line 200, "ABC" is considered equal to "abc", and the message "EQUAL" is printed on the system console device.

```
100 Set upcase 0
110 Gosub 200
120 Set upcase 1
130 Gosub 200
140 Goto 300
200 If "ABC"="abc" then print "EQUAL" else print "NOT EQUAL"
210 Return
300 End
*run
NOT EQUAL
EQUAL
*
```

Example 5-7.

Section 6

MEMORY MANAGEMENT

INTRODUCTION

The commands described in this chapter are used to allocate and de-allocate memory space for variables and program lines.

These commands include: COMPRESS, DATA, DELETE ALL, DELETE VAR, DIM, INTEGER, LET, LONG, READ, and RESTORE.

The COMPRESS Statement

**Syntax and
Descriptive Forms:** [[line-no.] COMPRESS [ALL]

PURPOSE

The COMPRESS statement compresses blocks of unused memory into one block.

EXPLANATION

Deleting variables or program lines, either during execution or during program development, leaves blocks of fragmented memory. The COMPRESS statement gathers unused memory into one contiguous block. The 4041 can then service larger requests for memory.

The ALL keyword is optional.

The DELETE ALL Statement

**Syntax and
Descriptive Forms:** DELETE ALL

PURPOSE

The DELETE ALL statement deletes all program lines and variables from memory and closes all open logical units.

DELETE ALL waits for any proceed mode I/O operations to complete before executing.

The DIM Statement

| | |
|--------------------------|---|
| Syntax Form: | [line-no.] DIM {numvar(numexp[,numexp])} [. .] {strvar[(numexp[,numexp])][TO numexp]} |
| Descriptive Form: | [line-no.] DIM {numeric-array(no.-of-rows[,no.-of-columns)]} [. .] {string[(no.-of-rows[,no.-of-columns])[TO max-string-length]} |

PURPOSE

The DIM statement declares a variable to be an array and allocates storage for it.

EXPLANATION

The DIM statement does not change the type (integer, short-floating-point, long-floating-point) of an existing variable. If a variable named in a DIM statement has not been referenced previously, the DIM statement assigns it a type of short-floating-point by default.

If a variable has been declared as a scalar prior to execution of the DIM statement, the variable's scalar value is copied into each element of the array when the DIM statement is executed.

If a variable has been previously declared as an array prior to execution of the DIM statement, the DIM statement performs a row-major mapping from the "old" array size into the "new" array size.

If the new array contains fewer elements than the old array, the extra elements from the old array are truncated, after the old array elements have been copied in row-major fashion.

If the new array contains more elements than the old array, the old array is copied into the new array in row-major fashion, then the extra elements in the new array are assigned a value of 0 (for numeric arrays) or null (for string arrays).

Array Subscripts

The lower bound of array subscripts is always 1. The upper bound of array subscripts is the rounded value of the numeric expression contained in the latest executed DIM statement for that array. The maximum allowable array subscript is 32,767.

String Arrays

Arrays of strings may be declared, where each element of the array is a string. A maximum string length may be set by means of the TO keyword. The default string length for a string array is 72 characters. The string size may be decreased, but not increased, without deleting the current values. The maximum length of any string or string array element is 32,767 characters.

Array Initialization

When the DIM statement is used on a previously undeclared numeric variable, each element of the numeric array is initialized to 0.

When the DIM statement is used on a previously undeclared string variable, each element of the string array is initialized to null.

EXAMPLE

```
2030      Dim ab(5,10),b$(100) to 80
```

This statement reserves storage space for a 50-element, two-dimensional numeric array Ab and a 100-element, one-dimensional string array B\$. Each element of B\$ may be up to 80 characters long. If variable Ab was declared previously, it retains its type (integer, short-floating-point, long-floating-point), else it becomes a short-floating-point variable by default.

The INTEGER Statement

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] INTEGER numvar{(numexp[,numexp])] [,numvar{(numexp[,numexp])}]...} |
| Descriptive Form: | [line-no.] INTEGER numvar{(no.-rows[,no.-columns])] [,numvar{(no.-rows[,no.-columns])}]...} |

PURPOSE

The INTEGER statement declares a variable to be of type integer and (optionally) reserves storage for an integer array.

EXPLANATION

The INTEGER statement declares a variable to be of type integer. If the variable has been previously declared and has a different type, or if the variable was previously undeclared, the INTEGER statement sets the value of the variable to 0.

An integer variable may be dimensioned as an array in the INTEGER statement itself or in a separate DIM statement. Subscript rules are as described in the "DIM" statement.

If an INTEGER statement declares a previously-declared integer scalar variable to be an array variable, the scalar's value is copied into each element of the integer array.

If an INTEGER statement changes the dimensions of a previously-declared integer array variable, the "old" array values are mapped into the "new" array values in row-major fashion.

If the new array contains fewer elements than the old array, extra elements from the old array are ignored. If the new array contains more elements than the old array, the row-major mapping is performed, and the remaining elements in the new array are assigned a value of 0.

EXAMPLE

```
3040      Integer inum(5,10)
```

This statement sets up a 50-element array Inum of values to be stored in integer format.

Section 7

CONTROL STATEMENTS

INTRODUCTION

This section describes statements in 4041 BASIC that start or stop program execution or change the normal order in which statements are executed.

The most commonly used statement to start program execution is the RUN statement. The DEBUG statement also starts program execution, but differs from RUN in that DEBUG starts execution with the 4041 in "debug" mode. The DEBUG statement is described in Section 4, "Program Editing, Debugging, and Documentation".

Normal order of execution for a 4041 BASIC program is sequential, from the statement with the lowest line number to that with the highest. Statements that can change this order include the CALL statement, the FOR and NEXT statements, the GO TO and GO SUB statements, and the IF . . THEN . . ELSE statement.

The normal order of execution can also be changed by the sensing of an interrupt condition and transfer of control to a handler for that condition. Interrupts and how they are handled are described in Section 12, "Interrupts and Errors".

The CONTINUE Statement

**Syntax and
Descriptive Forms:** CONTINUE

PURPOSE

The CONTINUE statement is used to resume program execution after the 4041 is PAUSEd, or to begin execution without a re-start from a line number specified by a GOTO statement.

NOTE

The CONTINUE statement is only available on 4041 units equipped with Option 30 (Program Development ROMs).

EXPLANATION

The 4041 becomes PAUSEd when:

1. A breakpoint is encountered in DEBUG mode;
2. The front-panel or P/D keyboard PAUSE keys are pressed while the "FTRP:" driver is the system console device;
3. CTRL-B is pressed on a user terminal while one of the "COMM:" drivers is the system console device; or
4. A STOP statement is executed.

Continuing from a WAIT

When the 4041 becomes PAUSEd during execution of a WAIT statement, executing a CONTINUE statement resumes execution of the WAIT. The exception to this rule occurs when an immediate-mode statement transfers control elsewhere while the 4041 is PAUSEd.

Executing Without a Re-Start

The CONTINUE statement can be used in immediate mode along with the GO TO or GO SUB statements to start a program at a specified line. Thus, the commands

```
*goto 1000  
*continue
```

start program execution at line 1000. When executed this way, the program is not initialized as it would be if the program were started using "RUN line-number". (See the description of the RUN command, later in this section, for more information.)

The END Statement

**Syntax and
Descriptive Forms:** line-no. END

PURPOSE

The END statement has two meanings in 4041 BASIC. During program development, entering an END statement marks the end of a program segment to the translator. During program execution, encountering an END statement terminates the program.

EXPLANATION

A 4041 BASIC program consists of a series of program segments. The first program segment is the main program. The second and succeeding program segments are subprograms or user-defined functions. Every program segment besides the main program starts with a SUB or FUNCTION statement and ends with an END statement. The END statement indicates the end of a program segment.

An END statement also terminates the program when encountered during execution. END closes all open files, closes all logical units, and flushes all waiting tasks.

The EXIT Statement

Syntax Form: [[line-no.] EXIT [numexp] TO numexp

Descriptive Form: [[line-no.] EXIT [no.-of-loops-to-exit-from] TO destination

PURPOSE

The EXIT statement is used to clean up the 4041's run-time stack when exiting prematurely from a FOR..NEXT loop.

EXPLANATION

The EXIT statement clears a specified number of FOR statements from the run-time stack.

The 4041 has an internal run-time stack for storing operations requiring more than one statement to complete (e.g., FOR, GO SUB, and CALL statements, and invocations of user-defined functions).

An operation is added to the stack whenever one of these statements is encountered in the program. The operation is deleted from the stack when its "partner" statement (e.g., the RETURN statement for GO SUBS and subprograms) is executed.

FOR..NEXT loops present a special case, however, since not every FOR statement may be cleared off the run-time stack. If the program exits the FOR..NEXT loop prematurely, the FOR statement is not cleared off the run-time stack. Instead, the FOR statement remains on the stack, taking up a small amount of memory space that would otherwise be available to the user.

This process is illustrated below. Suppose the 4041 is executing the following section of code:

```

150   For i=1 to 10
200   If (condition) then goto jumpout
210   Next i
220 Jumpout:  ! execution continues from here
230   For i=1 to 10
300   Next i
400   End

```

When line 150 is executed, a FOR statement is added to the run-time stack.

If the condition specified in line 200 is never met, the FOR..NEXT loop will be executed ten times and will exit normally, and the FOR statement will be deleted from the run-time stack.

If the condition specified in line 200 is met, however, the FOR..NEXT loop is terminated prematurely, and the FOR statement is not cleared from the run-time stack.

This accumulation of FOR statements on the stack is not necessarily a problem as long as the program doesn't try to reenter a loop already on the stack.

When a FOR statement is reached, the stack is checked for uncleared FOR statements that use the same index variable. Thus, in the above example, line 230 cannot be executed because another FOR statement using I as an index variable cannot be put on the stack. Similar reasoning shows why FOR..NEXT loops that use the same index variable cannot be nested.

The EXIT statement provides the solution. Suppose the example code was re-written as:

```

150   For i=1 to 10
200   If (condition) then exit to jumpout
210   Next i
220 Jumpout:  ! execution continues from here
230   For i=1 to 10
300   Next i
400   End

```

Line 200 tells the 4041 to "clean up" (i.e., delete) the last FOR statement from the run-time stack and continue execution from the line labeled "JumpOut". JumpOut can now be executed, because the conflicting FOR statement is removed by the exit statement.

Entering a numeric expression between the keyword EXIT and the keyword TO tells the 4041 to clean up the specified number of FOR statements from the run-time stack before continuing execution.

EXIT

When using the EXIT statement, take care to exit outside the FOR..NEXT loop. Otherwise the EXIT statement clears away the FOR statement from the run-time stack, and the program produces an error when the NEXT statement is encountered.

The EXIT statement does not need to be placed within a FOR..NEXT loop. Whenever it is encountered, it removes the last FOR statement from the stack. If a FOR statement is not already on the stack, the EXIT statement causes an error.

EXAMPLE

```
100   For i=1 to n
110     For j=i to i+50
120       For k=j to j+10
200         If (condition) then exit 3 to jump
250         Next k
260       Next j
270     Next i
300 Jump: !execution continues from here
```

Line 200 clears three FOR statements off the run-time stack, and resumes execution from the line labeled "Jump". Had line 200 read ".EXIT 2 TO Jump", one FOR statement would have remained on the run-time stack.

The FOR Statement

Syntax Form: line-no. FOR numvar = numexp TO numexp [STEP numexp]

Descriptive Form: line-no. FOR index-variable = starting-value TO limiting-value [STEP increment]

The NEXT Statement

Syntax Form: [(line-no.) NEXT numvar

Descriptive Form: [(line-no.) NEXT index-variable

PURPOSE

The FOR and NEXT statements work together to control the number of times a section of program is repeatedly executed.

EXPLANATION

When a FOR statement is executed, an index variable is specified, and a starting value, limiting value, and increment (default: 1) are evaluated.

The index variable is assigned the starting value. Succeeding program lines are then executed until a NEXT statement with the same index variable is encountered.

At this point the increment value is added to the current value of the index variable. The new value of the index variable is then compared with the ending value specified by the FOR statement.

If the increment value is positive, control passes to the line following the NEXT statement if the new value of the index variable is greater than the limiting value given in the FOR statement. Otherwise, the line after the FOR statements for the index variable is executed again.

If the increment value is negative, control passes to the line following the NEXT statement if the new value of the index variable is less than the limiting value given in the FOR statement. Otherwise, the line after the FOR statement for the index variable is executed again.

If the starting value is very large relative to the increment value, the index variable may not be incremented when the 4041 encounters the NEXT statement (e.g., FOR I= 1E38 TO 2E38 STEP 1).

NOTE

The final value of the index variable need not equal the limiting value specified in the FOR statement to terminate the loop. The program should not rely upon a specific value for the final index variable.

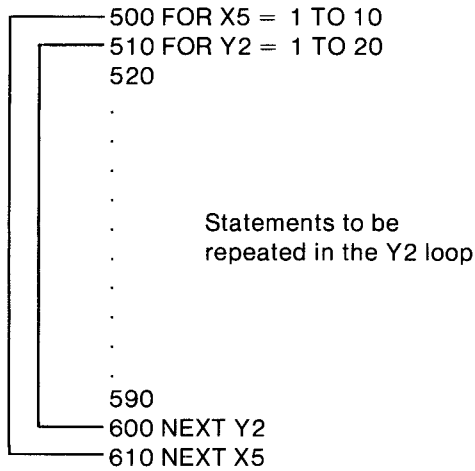
NOTE

After the FOR statement is evaluated, the starting value, limiting value, and step increment for the index variable are placed in temporary storage and are not evaluated again. These values can only be changed by re-entering the loop through the FOR statement.

FOR, NEXT

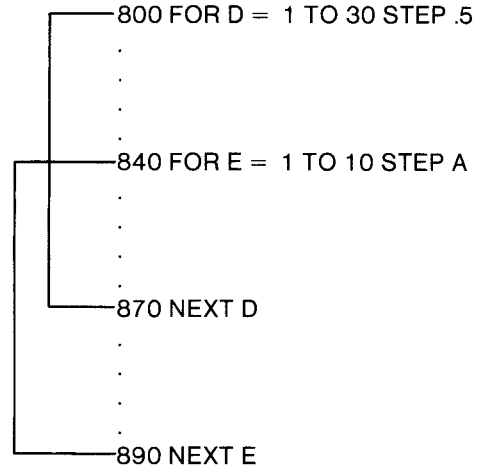
“NESTING” FOR/NEXT LOOPS

FOR/NEXT loops can be nested inside each other, as shown below:



In this example, lines 520 through 590 in the Y2 loop are executed 20 times for each pass through the X5 loop. When the program exits the X5 loop, the statements in the Y2 loop will have been executed 200 times.

FOR/NEXT loops cannot “cross”. The following is illegal:



NOTE

Two loops that use the same control variable cannot be nested. If two loops do use the same control variable, one cannot be entered until the other is completed or exited with the EXIT statement.

Branching Into and Out of a FOR/NEXT Loop

Branching out of a FOR/NEXT loop using GO TO or GO SUB statements is legal, but is not recommended. This is for two reasons: (1) the current value of the index variable may be unknown to the programmer unless the program completes the loop in the normal fashion; and (2) the next attempt to execute a FOR/NEXT loop with the same index variable will cause an error.

Branching into a FOR/NEXT loop from another point in a program is dangerous programming practice. If a NEXT statement is encountered without a corresponding FOR statement having been executed first, an error results.

The GO SUB and GO TO Statements

| | |
|--------------------------|---|
| Syntax Form: | <pre> [[line-no.] {GO SUB} {numexp} {GOSUB} {numexp OF numexp[,numexp]... } {GO TO} {GOTO} </pre> |
| Descriptive Form: | <pre> [[line-no.] {GO SUB} {line-number} {GOSUB} {index OF target[,target]... } {GO TO} {GOTO} </pre> |

PURPOSE

The GO SUB and GO TO statements transfer control to specified target lines. The GO TO statement branches to its target line with no return anticipated; the GO SUB statement branches to the target line and returns to the line following the GO SUB when it encounters a RETURN statement.

EXPLANATION

When the OF keyword is not used, the GO TO and GO SUB statements branch to the specified target lines directly. The GO TO statement branches directly to the target line, with no provision for returning to the point of branch. The GO SUB statement branches to the target line, and returns to the line following the GO SUB when a RETURN statement is encountered.

When the OF keyword is used in the statement, the GO TO and GO SUB statements compute the value of an arithmetic expression, and use that value to choose from a list of N possible destinations. If the expression evaluates to 1 (i.e., $0.5 \leq \text{expression} < 1.5$), the first destination in the list is chosen; if the expression evaluates to 2, the second destination is chosen; and so on for N specified destinations. If the value of the expression is less than 0.5 or greater than $N + 0.5$, no branch occurs; execution continues with the line following the GO TO or GO SUB statement.

Destinations for GOTO and GOSUB statements must be in the same program segment as the GOTO or GOSUB statement itself, i.e., the target line for a GOSUB or GOTO in the main program cannot be in a subprogram or user-defined function.

The GOTO and GO SUB statements may be used in immediate mode and followed by a CONTINUE statement to start execution from a specified line. When a program is started in this way, the 4041 does not initialize the program as it would if the program were started with "RUN line-number". (See the description of the RUN statement, later in this section, for more information.)

EXAMPLES

```
1000      Goto a+b of br1,br2,br3
```

The expression $A + B$ is evaluated. If $\text{ROUND}(A + B) = 1$, the program branches to line Br1; if $\text{ROUND}(A + B) = 2$, to line Br2; if $\text{ROUND}(A + B) = 3$, to line Br3. If $A + B < 0.5$ or $A + B > 3.5$, execution continues with the line after line 1000.

```
*goto 1500
*continue
```

These two immediate-mode statements start program execution at line 1500. (Available only on 4041 units equipped with Option 30, Program Development ROMs).

GO SUB AND GO TO

“Recursive” GOSUBs

As a feature for advanced programming applications, GOSUBs in 4041 BASIC are recursive, i.e., you can GOSUB to a line from within its GOSUB subroutine.

Note, however, that when GOSUB is used this way, only the line sequencing and “stacking/unstacking” of the GOSUB calls are supported. No “copies” of variables are made, and changes made to variables within the subroutine affect the values of the same variables as the GOSUBs “unstack”. Use with care.

The IF Statement

Syntax Form: line-no. IF numexp THEN statement [ELSE statement]

Descriptive Form: line-no. IF condition THEN consequence [ELSE consequence]

PURPOSE

The IF statement evaluates a condition, then takes action depending on the result of that evaluation.

EXPLANATION

The expression following the IF keyword is evaluated and rounded to an integer. If the expression evaluates to other than 0, the statement following the THEN clause is executed. If the expression evaluates to 0, the statement following the ELSE clause, if present, is executed.

If the IF statement has not transferred control to another portion of the program, execution continues with the statement following the IF statement.

The statements contained in THEN and ELSE clauses may not be IF, FOR, NEXT, SUB, FUNCTION, REM, IMAGE, or DATA statements.

EXAMPLES

```
1500 If x>0 then goto greater else goto lessoreq
```

The condition "X>0" is tested. If the condition is true, control goes to the line labeled "Greater"; if the condition is false, control goes to the line labeled "Lessoreq".

```
4500 If a>1 or b<1 then mar=55 else mar=65
```

The condition "A>1 OR B<1" is tested. If the condition is true, the variable "Mar" is set to 55; if the condition is false, "Mar" is set to 65. Execution continues with the next statement in sequence.

Invoking User-Defined Functions

| | |
|--------------------------|--|
| Syntax Form: | function-name [{(numexp) [,numexp] . . .}] {strex} [,strex] |
| Descriptive Form: | function-name [(argument[,argument] . . .)] |

PURPOSE

Invoking a user-defined function transfers control to a segment of the program given by the specified function name. Upon encountering a RETURN statement, the 4041 returns control to the line in which the function was invoked.

EXPLANATION

A user-defined function is a form of subprogram that (a) transfers control to a segment of the program defining the function, and (b) returns a value to the line that invoked the function.

A user-defined function begins with a FUNCTION statement and ends with an END statement. It differs from a subprogram begun with a SUB statement in that a function returns a value in place of the function name, and returns control to the line invoking the function. (See Section 11, Subprograms and User-Defined Functions, for more information.)

The RCALL Statement

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] {RCALL strexp [,exp]...} { strexp [exp[,exp]...]} |
| Descriptive Form: | [line-no.] {RCALL routine-name [,parameter]...} { routine-name [parameter[,parameter]...]} |

PURPOSE

The RCALL statement calls a ROM pack routine.

EXPLANATION

RCALL "OPT2IN"

A call to the "OPT2IN" routine (used on 4041 units equipped with Option 2) takes the form:

```
[line-no.] RCALL strexp,numexp,numexp,numvar
```

The string expression must evaluate to the string "OPT2IN".

The first numeric expression is the number of the starting register in the device connected to the Option 2 (TTL) interface. This must evaluate to a number from 0 to 127, inclusive.

The second numeric expression is the number of registers to concatenate. This must evaluate to a number from 1 to 4, inclusive.

The numeric variable is the location in which the incoming numeric value will be stored.

The 4041 converts the binary values in the registers specified by the RCALL statement into an integer, then stores the result in the specified numeric variable.

RCALL "OPT2OUT"

A call to the "OPT2OUT" routine (used on 4041 units equipped with Option 2) takes the form:

```
[line-no.] RCALL strexp,numexp,numexp,numexp
```

The string expression must evaluate to the string "OPT2OUT".

The first numeric expression is the number of the starting register to output to in the device connected to the TTL interface. This must evaluate to a number from 0 to 127, inclusive.

The second numeric expression is the number of registers to concatenate. This must evaluate to a number from 1 to 4, inclusive.

The third numeric expression is the data element to be output through the TTL interface.

The 4041 converts the data element into a binary integer with length equal to 8 times the number of registers to be concatenated, and stores the result starting with the specified starting register. The 4041 then outputs the result through the TTL interface port.

EXAMPLES

```
1050      Rcall "opt2in",3,4,nerp
```

Inputs a value through the TTL interface port from registers 3, 4, 5, and 6 of the device connected to the TTL interface port, converts the result into the 4041's internal format and stores it in numeric variable Nerp.

```
1060      Rcall "opt2out",3,4,zoom
```

Takes the contents of variable Zoom, converts it into a 32-bit binary integer, outputs the value through the TTL interface port and stores it in registers 3, 4, 5, and 6 of the device connected to the TTL interface.

RCALL**INVISIBLE RCALLS**

The word RCALL may be omitted with romcalls less than nine characters long. Romcalls longer than eight characters must be abbreviated to exactly eight characters in order to omit the RCALL keyword.

If the keyword RCALL is omitted, the comma following the romcall name must be omitted also.

When the RCALL keyword is omitted, rompack routines that contain a space may be called by substituting an underscore for the space. Note that certain errors usually returned at translation time are now delayed until run time because of this new capability. For example, entering the line

```
100 Foo x + 1
```

with Version 1 firmware would have returned a syntax error (indicating that the user probably forgot an equal sign) when the user tried to enter the line into the program. Under Version 2, the 4041 will accept the line and attempt to execute a romcall routine "Foo". The 4041 will then return error #491 (Romcall routine not found) at execution time if it does not find a romcall routine named "Foo" in the available rom packs.

Note also that variables can have the same names as romcall routines. Variable names of "Opt2in", "Opt2out", "Move", "Draw", etc., are legal, and will not conflict with the romcall routines of the same name.

When the RCALL keyword is omitted in an RCALL statement, the romcall routine name is printed in upper case when the statement is listed. This helps distinguish romcall routine names from variable names in the program.

Example: (using R01 Graphics Rompack)

```
100 GINIT 1,4025,1
110 MOVE 50,50
120 DRAW 75,75
130 ASK_WIND lox,hix,loy,hiy
```

These statements are equivalent to

```
100 Rcall "ginit",1,4025,1
110 Rcall "move",50,50
120 Rcall "draw",75,75
130 Rcall "ask_window",lox,hix,loy,hiy
```

The RETURN Statement

**Syntax and
Descriptive Forms:** [line-no.] RETURN

PURPOSE

The RETURN statement returns control to the line following a GOSUB or CALL statement, or to the line in which a user-defined function was invoked.

EXPLANATION

Executing GO SUB and CALL statements or invoking user-defined functions causes the program to branch to a designated target line. The program continues execution from the target line until it encounters a RETURN statement. The RETURN statement transfers control back to the statement following the GO SUB or CALL, or to the statement that invoked the user-defined function.

EXAMPLE

```
250 Gosub 1000
260 Var1=2*var2
.
.
1000 ! gosub subroutine starts here
.
.
1100 Return
```

Line 250 transfers control to line 1000. The program continues from line 1000, until it encounters the RETURN statement in line 1100. After the RETURN statement is executed, control transfers to the line following the GO SUB statement.

The RUN Statement

Syntax Form: RUN [numexp]

Descriptive Form: RUN [starting-line-number]

PURPOSE

RUN is an immediate-mode command that starts (or re-starts) program execution. RUN starts execution from the first line of the program; RUN followed by a line number starts execution from the specified line.

NOTE

The RUN command is only available on 4041 units equipped with Option 30 (Program Development ROMs).

EXPLANATION

RUN executes a program without breakpoints or trace flags enabled. (See Section 4, "Program Editing, Debugging, & Documentation", for more information about breakpoints and trace flags).

The RUN command may be entered when the 4041 is in idle mode, after a BREAK command has been executed (in debug mode), after the PAUSE key has been pressed on the front panel or P/D keyboard, or after the BREAK or CTRL-B keys have been pressed on a user terminal connected to an RS-232-C interface port.

The RUN keyword may be followed by an optional numeric expression designating the line at which execution is to re-start. This line must be in the main program segment.

The RUN statement does several "housecleaning" functions to ensure that a newly started or restarted

program is not hampered by assignments or settings made previously. These housecleaning functions of the RUN statement are:

- Delete all handler linkages;
- Reset interrupt conditions to their power-up enabled/disabled states;
- Reset ANGLE, AUTOLOAD, UPCASE, and FUZZ parameters to their power-up settings;
- Reset the DATA statement pointer to the first DATA statement in the program;
- Clear pending operations;
- Close all open files;
- Close all open logical units;
- Clear pending interrupts, last error values, last result from VAL and VALC functions, and the function key queue;
- Clear the front panel display;
- SET PROCEED 0.

EXAMPLES

```
*run
```

Starts program execution from the first line of the program.

```
*run 1250
```

Starts program execution from line 1250 of the main program segment.

Section 8

INPUT/OUTPUT

INTRODUCTION

This section discusses the statements in 4041 BASIC that exchange data between the 4041's memory and other devices.

The section starts by defining the concept of an I/O driver and presenting the names of the drivers available on the 4041.

A discussion of the concept of "System Console Device" and its relationship to 4041 I/O operations follows.

The section continues with a discussion of stream specifications ("stream specs") and logical units, which are the means by which the programmer specifies alternate data paths for I/O operations. Stream specs and logical units are also used to vary parameters that control driver operations.

There follows a discussion of "proceed-mode" I/O, which allows an I/O driver to perform INPUTs, PRINTs, RBYTEs, and WBYTEs in parallel with one or more other drivers.

The I/O commands are then described, in alphabetical order.

The commands presented are:

CLOSE Returns one or more logical unit numbers to their default stream specs.

COPY Transfers data from one device or file to another.

DATA Stores data values within a 4041 program block.

GETMEM Transfers data from a buffer string into string variables or numeric variables in memory.

IMAGE Specifies data format for USING clauses.

INPUT Transfers data from a device into memory.

OPEN Associates a logical unit with a stream spec; assigns values to logical parameters.

PRINT Transfers data from memory to a device.

PUTMEM Transfers numeric or string data into a string variable.

RBYTE Transfers "primitive" (8-bit) bytes from a device into memory.

READ Transfers values from DATA statements into memory.

RESTORE Sets the DATA statement pointer to the first element of the first DATA statement in a program block.

SELECT Specifies the default stream spec for RBYTE, WBYTE, and POLL statements.

WBYTE Transfers "primitive" (8-bit) bytes from memory to a device.

I/O DRIVERS

A driver is a firmware module that controls the operation of an I/O device. The drivers available on the 4041 are:

| | |
|--------------|--|
| F RTP | Accepts input from the Program Development (P/D) keyboard and front-panel function/numeric keypad, and controls operation of the 20-character alphanumeric display (output). |
| COMM (COMM0) | Controls data transfers through the standard RS-232-C interface port. |
| COMM1 | Controls data transfers through the optional RS-232-C interface port (available on 4041 units equipped with Option 1). |
| GPIB (GPIB0) | Controls data transfers through the standard GPIB interface port. |

| | |
|--|---|
| GPIB1 | Controls data transfers through the optional GPIB interface port (available on 4041 units equipped with Option 1). |
| OPT2 | Controls data transfers through the Option 2 (TTL) interface port (available on 4041 units equipped with Option 2). |
| PRIN | Controls operation of the 20-character thermal printer on the front panel. |
| TAPE | Controls operation of the DC-100 tape drive. |
| All I/O operations on the 4041 either use default drivers or use drivers specified by the user within the I/O command. | |
| DISK | Controls operation of the SCSI disk drive. |

THE SYSTEM CONSOLE DEVICE

The system console device, or "console", is the primary driver through which the user communicates with the 4041. This driver must be either the F RTP driver, the COMM0 driver, or (on 4041 units equipped with Option 1 or Option 3) the COMM1 driver.

For example, the user might communicate with the 4041 by means of a computer terminal connected to an RS-232-C interface port designated as the system console device.

When the 4041 powers up, the system console device defaults automatically to "F RTP:". To make the standard RS-232-C interface port the system console device, the 4041 must execute the following SET CONSOLE statement:

```
Set console "comm:"
```

To make the front panel display/keyboard the system console device (if another device is system console), the 4041 must execute the following SET CONSOLE statement:

```
Set console "frtp:"
```

For more information, see the description of the "SET CONSOLE" statement in Section 5, *Environmental Control*.

NOTE

In order to use an RS-232-C computer terminal as the system console device on 4041 units NOT equipped with Options 30 and 31 (P/D ROMs and P/D keyboard), the user MUST have a DC-100 tape or disk containing an ITEM file named "AUTOLD" that includes the statement.

```
Set console "comm:".
```

This tape must either be inserted at power-up, or the 4041 must be powered up and the AUTOLOAD button on the front panel must be pressed.

INPUTTING DATA ITEMS FROM THE CONSOLE

When no other data path is specified, the 4041 always expects input from the console when it executes an INPUT statement.

Thus, when executing the command

```
2000 Input num1,num2,num3
```

the 4041 expects to receive three numeric values from the system console device (P/D keyboard or RS-232-C computer terminal).

Similarly, the command

```
2500 Input num1,string1$,num2,string2$
```

causes the 4041 to expect to receive a numeric value, a string, another numeric value, and another string (in that order) from the system console device.

OUTPUTTING DATA ITEMS TO THE CONSOLE

When no other data path is specified, the 4041 always outputs data to the system console device when it executes a PRINT statement.

Thus, the command

```
3000 Print num1,num2,num3
```

causes the 4041 to output three numeric values, separated by EOU characters, to the system console device. Similarly, the command

```
3500 Print num1,string1$,num2,string2$
```

causes the 4041 to output a numeric value, a string, another numeric value, and another string, separated by EOU characters, to the system console device.

SPECIAL CAPABILITIES OF THE SYSTEM CONSOLE DEVICE

Immediate-mode Statements

Only the system console device can accept immediate-mode statements.

Function Keys

Function keys are only honored when entered from the system console device with the KEYS condition enabled. (See Section 12, *Interrupt Handling*, for more information about the KEYS condition.)

When the front panel is the system console device, pressing keys 1 through 9 invokes the corresponding user function; pressing key 0 invokes user function 10.

When the COMM is the system console device, pressing CTRL-F followed by keys 1 through 9 invokes user functions 1 through 9; pressing CTRL-D followed by keys 1 through 9 invokes user functions 11 through 19; pressing CTRL-F followed by key 0 invokes user function 10; and pressing CTRL-D followed by key 0 invokes user function 20.

Abort/CTRL-C

When the ABORT condition is enabled, the ABORT keys on the front panel and P/D keyboard always cause the program to transfer to the current ABORT handler. In addition, if the COMM is the system console device, pressing CTRL-C also causes the program to transfer to the current ABORT handler. The "current ABORT handler" may be either a user-defined handler or the system handler.

Pause/Continue Functions

When the FRTP is the system console device, pressing the PAUSE key on the front panel or P/D keyboard pauses the system, and pressing the PROCEED key on the front panel or the CONTINUE key on the P/D keyboard causes program execution to continue.

When the COMM is the system console device, pressing CTRL-B pauses the system, and typing the CONTINUE command (if the P/D ROMs are available) or pressing CTRL-^ followed by "C" causes program execution to continue.

SPECIFYING DATA PATHS

STREAM SPECIFICATIONS

Data can be transferred between the 4041 and several different devices (e.g., the front panel or program development keyboard, a computer terminal, the alphanumeric display, the thermal printer, etc.).

A firmware driver (a short program that constitutes part of the 4041's operating system) controls data transfers between the 4041 and each device.

A set of parameters specific to each driver controls the way data are interpreted and I/O operations are executed.

The programmer selects a driver for an I/O operation and specifies parameters for the driver to use by means of stream specifications (a.k.a., "stream specs", for short).

Complete information on stream specs and on the parameters that can be used with each driver are given in Appendix D, "Stream Specifications".

INPUT and PRINT statements that do not include a stream spec signify that input is to come from or output to go to the system console device.

LOGICAL UNIT NUMBERS

Logical unit numbers are "shorthand" ways of referring to stream specs in I/O commands. Legal logical unit numbers are integers in the closed interval 0..32767.

Opening Logical Units: the OPEN Statement

A logical unit number is assigned to a stream spec by means of the OPEN statement. Once the logical unit is OPENed, the stream spec may be referenced in I/O commands by its associated logical unit number. (See the description of the OPEN statement, later in this section, for more information.)

Example . The statement

```
1500 Open #1:"frtp(view=2.5,rate=.2):"
```

assigns logical unit number 1 to the stream spec "FRTP-(VIEW = 2.5, RATE = .2)". I/O can be directed to this stream spec by referencing its logical unit number (e.g., "PRINT #1:X,Y" prints the values of X and Y on the alphanumeric display).

Closing Logical Units: the CLOSE Statement

The CLOSE statement performs the opposite function of the OPEN statement; it is used for "housekeeping" purposes (e.g., to make sure that all contents of I/O buffers have been transferred, to update tape directories, etc.).

Example. The statement

```
2000 Close 1,2,a+b
```

closes logical units 1, 2, and A + B.

Default Stream Specifications for Logical Unit Numbers

All logical unit numbers are assigned to certain stream specs by default. Unless the user reassigns a logical unit to a different stream spec, the stream spec associated with each is as follows:

- Logical unit numbers 0 through 30 represent primary addresses 0 through 30 on the standard GPIB. The default stream spec is "GPIB(PRI = logical-unit-number, TIM = 4)".
- Logical unit numbers 31 through 32767 are assigned to the system console device. The console stream spec powers up to "FRTP:" by default. The console stream spec may be reset by the user via the SET CONSOLE command.

“PROCEED-MODE” I/O

The 4041 has a special I/O mode called “proceed mode” that allows it to continue executing program statements while an I/O driver simultaneously completes execution of a previous INPUT, PRINT, RBYTE, or WBYTE statement. The program can set up handlers for completion of proceed-mode input or output (see the description of the IODONE interrupt condition in Section 12, *Interrupt Handling*).

This mode is called “proceed mode” because, in effect, the 4041 gives the driver controlling the I/O device instructions on how to execute the statement, then “proceeds” to execute subsequent program statements (which may be I/O statements involving other drivers).

When not in proceed mode, the 4041 is said to be in “sequential” mode, because it executes all statements sequentially, waiting for all driver activity caused by one statement to be completed before proceeding with the next statement.

TOGGLING BETWEEN PROCEED AND SEQUENTIAL I/O MODES

The 4041 is put into proceed mode when it executes the statement “SET PROCEED 1”.

The 4041 returns to sequential mode when it executes the statement “SET PROCEED 0”, or when it executes an INIT, RUN, DEBUG, or DELETE ALL statement.

PROCEED MODE INPUT RESTRICTIONS

When the 4041 executes an INPUT statement in proceed mode, the INPUT statement must call for one data item, which must be input into a string variable. Multiple data items cannot be input by the same input statement in proceed mode, nor can numeric data items be input in numeric form in proceed mode.

Numeric data items can be input indirectly in proceed mode by inputting the characters of the number into a string variable, then including a GETMEM statement in the IODONE handler for the proceed-mode input.

PROCEED MODE PRINT RESTRICTIONS

Any number and type (string or numeric) of data items may be output by means of a single PRINT statement when the 4041 is in proceed mode. However, the 4041 does not “proceed” (begin executing the next program statement) until the number of characters remaining to be output can fit within the 4041’s proceed-mode output buffer.

When a proceed-mode PRINT statement calls for more characters to be output than will fit in the 4041’s proceed-mode output buffer, the 4041 outputs one “buffer-full” of characters without proceeding to the next statement. It then refills the buffer. The 4041 only proceeds to the next statement when all characters to be output can fit within the buffer.

The 4041’s default proceed-mode output buffer can hold up to 512 ASCII characters. The programmer can specify that a larger (or smaller) buffer is to be used by including a BUFFER clause in the proceed-mode PRINT statement.

PROCEED MODE RBYTE/WBYTE RESTRICTIONS

In proceed-mode, an RBYTE statement can only input a single string variable or string array element per statement.

In proceed-mode, a WBYTE statement may only output a single string expression per statement.

PROCEED MODE CAVEAT WHEN USING A WINCHESTER HARD DISK

Since the response from a hard disk is fast, a problem exists in 4041 Basic when using an “IODONE” handler. The problem is when using a subprogram or gosub routine via an “IODONE”, and that segment of code starts another I/O to the disk, the subprogram or gosub routine may never execute a return statement prior to the next “IODONE” call/gosub. If a call to a subprogram is being used, the user may get an error message stating an active subprogram is being called. If a gosub is being used, the user may eventually see a run-time stack overflow error.

RESTRICTED ACCESS TO VARIABLES FROM PROCEED MODE INPUT/RBYTE STATEMENTS

Statements referring to values obtained from uncompleted proceed mode INPUT and RBYTE statements are automatically delayed from executing until the values required become available.

Example.

```
100 Set proceed 1
110 Open #1:"datafi"
120 Dim bigstr$ to 500
130 Input #1: bigstr$
.
.
.
200 Num1=val(bigstr$)
```

Execution of line 200 is delayed until the proceed-mode INPUT statement of line 130 has been completed.

THE IODONE CONDITION

The 4041 signals the completion of proceed-mode statements by setting the IODONE interrupt condition true. The 4041 can thus be programmed to start a large data transfer in proceed mode, perform other operations while the data transfer is going on, interrupt those operations when the data transfer is complete (IODONE becomes true), and return to its previous operations when the IODONE condition has been handled.

One example of the use of an IODONE interrupt routine is to process large amounts of incoming data. The 4041 can process other data until the input is complete, interrupt its processing to handle the new data, then return to its other processing.

The status of proceed-mode I/O on a logical unit can be queried via the ASK("IODONE") function, and an interrupt can be generated by means of the ON IODONE statement.

See Section 12, *Interrupt Handling*, for more information about the IODONE interrupt condition.

The CLOSE Statement

| | |
|--------------------------|---|
| Syntax Form: | <code>[line-no.] CLOSE {numexp [,numexp ...] {ALL}}</code> |
| Descriptive Form: | <code>[line-no.] CLOSE {lunum [,lunum ...] {ALL}}</code> |

PURPOSE

The CLOSE statement returns one or more logical unit numbers to their default stream specs. The CLOSE ALL statement closes all open logical units. This is also done implicitly when an END, RUN, DEBUG, or INIT statement is executed.

EXPLANATION

The CLOSE statement performs the opposite function of the OPEN statement.

CLOSE automatically performs any "housekeeping" necessary to discontinue operations on a given logical unit (e.g., closing files, ensuring that data transfers are completed, updating directories, etc.).

Memory required to support an open logical unit is returned to the free memory pool and is available for storing commands or data after a CLOSE statement is performed.

Attempting to close an un-opened logical unit number yields no result.

EXAMPLE

```
3020 Close 1, 9+if3
```

This statement returns logical unit numbers 1 and 9 + IF3 to their default stream specs.

The COPY Statement

Syntax Form: [line-no.] COPY stream-spec TO stream-spec

Descriptive Form: [line-no.] COPY source TO destination

PURPOSE

The COPY statement copies data from one device or file to another.

DESCRIPTION

Data is copied from one device to another until an end-of-file is encountered (source:TAPE), two successive null lines are entered (source:FRTP, COMM, or GPIB), the EOI line is asserted (source:GPIB), the user presses ABORT on the front panel or CTRL-C on a user terminal (if the COMM is the system console device), or an error occurs.

If either stream spec given in the COPY statement is a file spec only (no driver name appearing to the left of a colon), the current SYSDEV driver spec is used by default.

Attempting to copy information from a non-existent file or device results in an error.

TAPE Driver

COPY statements using the TAPE driver as a source copy data from a DC-100 tape file exactly as stored. For instance, if a program was SAVEd onto the tape, copying that program to the COMM would produce a listing of the program in its shortest possible form (four-character keywords, no unnecessary spaces, etc.)

COPY statements using the TAPE driver as a destination should specify relevant file parameters (OPEN, CLIP, etc.).

FRTP and COMM Drivers

COPY statements using the FRTP or COMM drivers as a source accept data and copy it to the destination until the user types two null lines (i.e., types 2 <cr>'s without entering any characters).

GPIB Driver

COPY statements using the GPIB driver as a source transfer information from the addressed device's output buffer to the destination until two successive null messages are encountered, or until the EOI line is asserted.

PRIN Driver

The PRIN driver can only be used as a destination in a COPY statement.

EXAMPLES

```
3030 Copy "gpib(pri=4):" to "gpib(pri=5):"
```

This statement copies data from one device on the GPIB to another device on the GPIB.

```
3040 Copy "prog" to "prin:"
```

This statement copies a file named "Prog" from the DC-100 tape drive to the front-panel thermal printer.

```
3050 Copy "file1" to "file2(open=new,clip=yes)"
```

This statement takes a file called "File1" from the DC-100 tape, creates a copy of it on the tape, and calls the copy "File2".

The IMAGE Statement

Syntax Form: [line-no.] IMAGE any-characters-except-CR

Descriptive Form: [line-no.] IMAGE format-string-for-INPUT-USING-or-PRINT-USING-statement

PURPOSE

The IMAGE statement specifies the input or output format to be used with INPUT, PRINT, GETMEM, or PUTMEM statements containing USING clauses.

EXPLANATION

The string contained in an IMAGE statement is called an "image string".

An image string is a special string of characters that regulates the format for input or output when an INPUT, PRINT, GETMEM, or PUTMEM statement containing a USING clause is executed.

The image string consists of a combination of field operators and field modifiers that defines the format to be followed for the PRINT or INPUT operation. Image strings can be specified in either of two ways: (1) as part of the INPUT, PRINT, GETMEM, or PUTMEM statement's USING clause, or (2) in an IMAGE statement.

IMAGE statements can appear anywhere in a program, before or after the statements that refer to them. The same IMAGE statement can be referred to any number of times in the same program.

See the descriptions of USING clauses in the PRINT and INPUT statements for more information on field operators and modifiers for image strings.

EXAMPLE

In Example 8-1, line 200 tells the 4041 to use the image string contained in line 1000 to execute the PRINT operation.

Line 1000 tells the 4041 to print the string "A = ", then to print the value of the first numeric variable in the print list in the next six character positions, with two digits to the right of the decimal point. The 4041 is then to print spaces until it reaches character position 16.

It then prints the string "B = ", followed by the value of the next numeric variable in the print list, printed in six character positions with one digit to the right of the decimal point. The 4041 then prints spaces until it reaches character position 31.

It then prints the string "C = ", followed by the value of the third numeric variable in the print list, printed in six character positions with no digits to the right of the decimal point.

See the descriptions of the USING clause under the INPUT and PRINT statements for more information about image strings.

```
200 Print using 1000:a,b,c
.
.
1000 Image 'A=',6.2g,16t,'B=',6.1g,31t,'C=',6.0g
```

Example 8-1.

The INPUT Statement

Syntax Form: [line-no.] INPUT[clause-list:] {numvar}[,numvar] . .
{strvar} [,strvar]

where:

clause-list = [#numexp] [ALTER strexp] [BUFFER strvar][DELN strexp] [INDEX {numexp;}][USING {numexp;}]
[#strex] [PROMPT strexp] {strex} {strex}

Descriptive Form: [line-no.] INPUT [input clauses:][input list]

PURPOSE

The INPUT statement transfers data from a peripheral device into variables in the 4041's memory.

EXPLANATION

An INPUT statement without a USING clause transfers data into memory using default data conversion rules explained below. An INPUT statement containing a USING clause transfers data into memory using conversion rules provided by the user via the USING clause.

Default Data Input Rules

If no USING clause is specified, the procedures used by 4041 to satisfy the INPUT command are as follows:

1. When a numeric data item is input, an entire ASCII number is accumulated until a default terminator is seen (see the description of the DELN clause, later in this section) or the end-of-message character is recognized.

The number is then converted to internal form and stored in the numeric variable specified by the INPUT statement.

Non-numeric characters preceding the first numeric character in the input stream are discarded.

2. String variables cause data bytes to be input and stored in the string variable specified by the INPUT command until either the end-of-message character is recognized, the string is filled, or an error occurs. Null strings are allowed.
3. Numeric arrays are input element-by-element until the entire array is filled. Each array element is input using the rule for numeric variables (#1).
4. String arrays are input element-by-element until the entire string array is filled. Each element is input using the rule for string variables (#2).
5. If enough data is not provided to satisfy the argument list, the INPUT statement does not terminate until all arguments in the argument list have been satisfied.

EXAMPLES

```
3070 Input avar, astr$
```

This command calls for a numeric value and a string to be input from the system console device.

```
3071 Input #5:hdr$,val1
```

This command transfers data from the device open as logical unit number 5 into the 4041's memory. Characters are input to string variable Hdr\$ until the end-of-message-unit character is seen; a numeric value is then input to Val1.

USE OF INPUT STATEMENT WITH DIFFERENT DEVICE DRIVERS

If no “#” clause is included in an INPUT statement, the 4041 assumes that input is to be received from the system console device.

Input Via COMM Driver

When inputting a string via the COMM driver, control characters can be inserted into the string by preceding each one with an ESCAPE character. (On some terminals, this character is sent by the ALT MODE key.)

During input, the control character thus inserted will appear on the terminal in its “caret-uppercase-letter” form.

When the string is output, the control character itself will be printed.

Example.

```

100 Set console "comm:"
110 Input a$
120 Print a$
run
(user input) ABCJDEF
(output)   ABC
          DEF

```

The “\$” represents the ESCAPE character; the “^J” is the upper-case character representation of the LINE FEED character.

NOTE

No character output by the 4041 can affect the operation of the 4041 itself. However, the character may affect the device attached to the 4041.

Thus, a CTRL-C imbedded within a string will not affect the 4041 when the string is output.

Input Via FRTP Driver

When an INPUT statement is being executed via the FRTP driver, the user-definable function keys on the front panel denote digits to be input, and not user-definable functions.

User-definable functions can still be invoked via the P/D keyboard’s user-definable function keys.

Control characters may be inserted into a string being input via the P/D keyboard by preceding each control character with an ESCAPE character (the same as the COMM driver). However, CTRL-M’s cannot be imbedded into a string being input via the FRTP driver.

Input Via GPIB Driver

Use of the INPUT statement with GPIB drivers is described under *High-Level Message Transfers* in Section 9, *Instrument Control With GPIB*.

Input Via OPT2 Driver

The OPT2 driver does not use the INPUT statement for input. Instead, it uses the “OPT2IN” romcall, described along with the RCALL statement in Section 7, *Control Statements*.

Input Via Opt3 (SCSI) Driver

Using the INPUT statement to transfer data from the disk files over the SCSI interface is almost identical to the tape driver. The only exception is the difference in the stream specification.

Example:

```

100 Open #1:"disk(DEV=3,UNI=0):FILE NAME"
110 Dim a1(5),a2(10)
120 Input #1:a1,a2,a3

```

Input Via TAPE Driver

The file used for input from the DC-100 tape must be opened with an OPEN parameter of OLD (OLD is the default value).

The 4041 keeps track of its position as it “moves through” a file. Thus, successive INPUT statements read successive data elements from the input file.

Example.

```

100 Open #1:"datafi"
110 Dim a1(5),a2(10)
120 Input #1:a1,a2,a3

```

A file called “DataFi” is opened (with OPEN parameter of OLD, by default) on the DC-100 tape.

Line 110 dimensions two arrays of short-floating-point numbers, A1 (containing five elements) and A2 (containing ten elements).

Line 120 reads five elements from DataFi to fill array A1, ten more elements to fill array A2, and a single element to fill array A3. The next INPUT statement calling for a value from DataFi will read the seventeenth element in the file.

A file may be opened for input on more than one logical unit at a time. The “pointer” into the tape file is maintained separately for each logical unit.

INPUT/OUTPUT

INPUT

Example.

```
100 Open #1: "datafi"
110 Open #2: "datafi"
120 Input #1:a,b,c
130 Input #2:d,e
```

Line 120 reads the first three data elements from file "DataFi" on the DC-100 tape.

Line 130 reads the first two data elements from file "DataFi" on the DC-100 tape.

The next INPUT statement using logical unit 1 will read data starting at the fourth data element in file "DataFi".

The next INPUT statement using logical unit 2 will read data starting at the third data element in file "DataFi".

No file can be open for both input and output at the same time.

See Section 14, "DC-100 Tape", for more information on use of the INPUT statement with tape files.

PROCEED MODE INPUT

The 4041 enters proceed mode upon execution of a SET PROCEED 1 statement.

In proceed mode, program execution continues while one or more I/O drivers complete the tasks assigned to them by an INPUT or PRINT statement, or until further execution requires that the proceed-mode INPUT or PRINT be complete, or until a proceed-mode I/O error occurs.

Example.

```
100 Set proceed 1
110 Open #1:"datfil"
120 Input #1: string$
130 A=0
140 B=1
.
.
190 Z=1.111E111
200 Num1=val(string$)
```

Line 100 puts the 4041 into proceed mode.

Line 110 opens a file on the DC-100 tape called "DatFil".

Line 120 reads a string from "DatFil" and stores it in string variable String\$.

While line 120 is executing, the 4041 executes lines 130 through 190 "in parallel" with the input operation, i.e., while the input is coming in from the tape.

Line 200, however, requires that the previous input be complete in order to be executed. The 4041 therefore "pauses" to allow the I/O to complete before continuing execution.

Restrictions on INPUT in Proceed Mode

When the 4041 is in proceed mode, INPUT statements can only be used to input one string variable per INPUT statement.

The string variable used as a target variable for a proceed-mode input operation must be a simple string variable; it may NOT be an entire string array or string array element.

The GETMEM statement may be used after completion of an INPUT in proceed mode to "extract" the values of several numeric or string variables from the contents of the string variable that was input. (See the description of the GETMEM statement, elsewhere in this section, for more information.)

The only legal clause that can be used with an INPUT statement when the 4041 is in proceed mode is the "#" clause. The ALTER, BUFFER, DELN, DELS, PROMPT, and USING clauses are illegal.

The ASK("IODONE") Function

The status of I/O operations on a given logical unit at any time can be determined by means of the ASK("IODONE") function. (See the description of the ASK("IODONE") function in Section 5, *Environmental Control*, for more information.)

The IODONE Interrupt

Control can be transferred to a given subprogram or GO-SUB subroutine upon completion of proceed-mode I/O on a given logical unit by means of the ON IODONE statement.

The IODONE interrupt condition is automatically enabled when the 4041 is put into proceed mode.

See Section 14, *Interrupt Handling*, for more information about the ON statement and the IODONE interrupt condition.

The ASK("PROCEED") Function

Whether or not the 4041 is in proceed mode can be determined by means of the ASK("PROCEED") function. ASK("PROCEED") returns a value of 1 if the 4041 is in proceed mode, 0 if not.

Errors During Proceed-Mode INPUT

Input errors occurring during proceed-mode input are handled in a special way. See the description of the ERROR interrupt condition in Section 14, *Interrupt Handling*, for more information.

INPUT CLAUSES

Clauses are "statements within statements" used with GETMEM, INPUT, PRINT, PUTMEM, RBYTE, and WBYTE commands to define a data transfer path or modify execution of an I/O operation. Clauses are separated from data lists by a colon.

The following clauses can be used with INPUT statements:

| | |
|--------|--|
| # | Specifies a logical unit number or stream spec for the INPUT operation. |
| ALTER | The 4041 outputs a string, which is modified by the user or device. The modified string is then sent back to the 4041. Compare PROMPT clause. |
| BUFFER | A string variable is used directly as an I/O buffer. Allows user to control amount of data transferred. |
| DELN | The user specifies delimiter characters for inputting numeric values. |
| DELS | The user specifies delimiter characters for inputting strings. |
| INDEX | For use with future enhancements that support random-access peripheral devices. |
| PROMPT | The 4041 outputs a string to a device. The device's or user's response is then sent back to the 4041. The string originally output by the 4041 is NOT sent back. Compare ALTER clause. |
| USING | Data formatting is under user control with an image string. |

Further explanation and examples showing the use of each clause follow.

The # Clause

Syntax Form:
#numexp OR #strexp

Descriptive Form:
#logical-unit-number OR
#stream-spec

The # clause specifies a data path for an I/O operation.

The # clause is used when I/O is to be performed to other than the default device for an I/O operation, or when the user wishes to specify other than default attributes for a driver or file.

If a numeric expression follows the "#", it is interpreted as the number of the logical unit to be used for the INPUT operation.

If a string expression follows the "#", it is interpreted as the stream specification to be used for the INPUT operation. The string expression must evaluate to a legal stream spec.

The numeric expression for a logical unit must evaluate to an integer in the range $0 \leq \text{lunum} \leq 32767$.

INPUT**The ALTER Clause****Syntax Form:**

ALTER string-expression

Descriptive Form:

ALTER string-to-be-sent-and-modified

The ALTER clause sends a string to a device for user modification prior to an INPUT operation. It is useful for providing a default for user input, or for re-displaying erroneous data for correction.

The string specified by the string expression following the ALTER keyword is output to a peripheral device. The string may then be modified by the device or by the user.

The string is then returned as data to satisfy the input operands specified in the INPUT data list.

The modified string can be stored within a numeric variable if it conforms to the rules for numeric constants, i.e., no letters or special characters (except the letters "E" or "e"), within limits for the type of number being stored, etc.

A number sent to the alphanumeric display can be "modified" on the front panel numeric keypad by pressing the "CLEAR" key on the front panel, then entering a new number and pressing "PROCEED".

A string sent to the alphanumeric display can be modified using the P/D keyboard's editing keys. When the string has been modified to the user's satisfaction, it can be entered into the 4041 by pressing "RETURN" on the keyboard or "PROCEED" on the front panel.

Example.

```
1234 Input alter "Name:":name$
```

This statement sends the string "Name:" to the system console device. The user types his/her name and presses CR. String variable Name\$ then contains a string consisting of the characters "Name:" plus the user's name.

The BUFFER Clause**Syntax Form:**

BUFFER strvar

Descriptive Form:

BUFFER string-variable-for-input-data

The BUFFER clause allows a user to specify a string variable to use as a buffer. The buffer can be used to store incoming data in its "pre-processed" form. It can also be used to tailor the size of the buffer space to the amount of incoming data, e.g., specifying a buffer 1,000 characters long allows 1,000 characters to be input during a single INPUT operation. (The default buffer size is 514 bytes.)

When no BUFFER clause is included in an INPUT statement, the 4041 creates a temporary buffer 514 bytes long to hold incoming data. The 4041 translates the data in the temporary buffer into the form(s) required by the INPUT arguments, stores the data, then deletes the buffer.

The BUFFER clause allows a user to specify a string variable in which to store the incoming data. After the data transfer is complete, the buffer's contents are not deleted, but remain available for editing or other uses.

Example.

```
90 Dim a$ to 20
100 Input buffer a$:num1,num2,num3
110 Print num1,num2,num3
120 Print a$
run
(user input) 1,2,3
(output) 1.0<tab>2.0<tab>3.0
1,2,3
```

Line 100 inputs three numbers, Num1, Num2, and Num3, and saves the incoming data string in a buffer, A\$.

When the INPUT operation is complete, Num1, Num2, and Num3 are saved as short-floating-point numbers. In addition, A\$ contains the exact string that was input for Line 100.

The DELN Clause

Syntax Form:
 DELN strexp

Descriptive Form:
 DELN delimiter-list-for-numeric-data-items

The DELN clause specifies the delimiters for numeric data items when an INPUT statement is executed.

Default numeric data item delimiters are space, tab, comma, semicolon, colon, and the end-of-message (EOM) character.

The EOM character is always a delimiter for numeric data items, regardless of which other characters may be delimiters.

Sequential spaces and tabs are treated as one delimiter (i.e., on multiple inputs, you can type one value followed by any number of spaces, followed by the second value).

Inputting the EOM character in place of a value causes the 4041 to ask for that value again.

Inputting any other delimiter in place of a value causes the 4041 to set that value to 0.

Example.

```
5678 Input deln " ":num1,num2,num3
```

This statement calls for three numeric values to be input from the system console device (logical unit 32). Delimiters for the numeric values include space and carriage-return (EOM character).

Therefore, typing

```
1.414<sp>2.323<sp>4.567<cr>
```

in response to this command stores the value 1.414 in variable Num1, 2.323 in variable Num2, and 4.567 in variable Num3.

The DELS Clause

Syntax Form:
 DELS string-expression

Descriptive Form:
 DELS delimiter-list-for-string-variables

The DELS clause specifies the delimiter characters for strings when an INPUT statement is executed.

The only default delimiter for string data items is the end-of-message character (usually carriage-return, but can be set as a parameter in a GPIB or COMM stream spec).

The EOM character is always a delimiter for string data items, regardless of what other characters are delimiters.

Inputting a delimiter in place of a string sets the string variable being input to null.

The DELS clause treats multiple spaces and tabs in the same way the DELN clause does.

Example.

```
1000 Input dels "!":string1$,string2$,string3$
```

This command tells the 4041 to accept three strings as input from the system console device, using either an exclamation point (!) or the EOM character as a delimiter.

Thus, typing

```
AAA!BBB!CCC<cr>
```

in response to this command would store "AAA" in String1\$, "BBB" in String2\$, and "CCC" in String3\$.

Typing

```
AAA!CCC<cr>
```

sets String1\$ to "AAA", String2\$ to "" (null), and String3\$ to "CCC".

The INDEX Clause

Syntax Form:

```
INDEX {numexp}  
      {strex}
```

The INDEX clause will be included in a future enhancement, for use with peripheral devices that support random access.

The word "INDEX" can be used as a line label, variable, or subprogram name in 4041 programs.

The PROMPT Clause

Syntax Form:

```
PROMPT strexp
```

Descriptive Form:

```
PROMPT prompt-message
```

With this clause the 4041 sends a string to a device prior to receiving input from that device. Unlike the ALTER clause, the string does not become part of the input data. This clause is normally used to request input from a user, or to query for input from an instrument attached to the GPIB.

The PROMPT clause can be used to "set up" input operations from devices on the GPIB and to interrogate instruments on the GPIB for their current settings or readings. When used in this way, the 4041 automatically sends the End-Of-Query character (EOQ; default is no output) following the string specified after the PROMPT clause.

Examples.

```
1000 Input prompt "Name":name$
```

This statement asks for the user's name, then stores the reply in string variable Name\$.

```
2010 Dc=19  
2020 Open #dc:"gpib(pri=&str$(dc)&",eoq=?):"  
2030 Input #dc prompt "Func":func$
```

Line 2030 sends the string "Func?" to an instrument at the primary address specified by the value of numeric variable DC. (The EOQ character "?", as defined by the OPEN statement, is output after the string given in the PROMPT clause.) If the device is a Tektronix digital counter it returns a string indicating which measurement function is currently selected. This string is stored in string variable Func\$.

The USING Clause

Syntax Form:

USING {numexp} {strexp}

Descriptive Form:

USING {line-number-of-IMAGE-statement}
{image-string}

When used with the INPUT statement, the USING clause allows the user to check the format of input data.

The USING clause either contains an image string, which defines a format for input data checking, or refers to an IMAGE statement that contains an image string.

Legal image strings for INPUT USING consist of any combination of INPUT USING operators, spaces, and commas. Spaces and commas may be included as needed to improve readability, but are ignored by the BASIC interpreter.

Legal image strings for INPUT USING consist of any combination of INPUT USING operators, modifiers, spaces, and commas. INPUT USING modifiers appear before the operators they modify in the image string. Spaces and commas may be included within an image string as needed to improve readability, but are ignored by the 4041.

Legal INPUT USING operators are listed in Table 8-1. Legal INPUT USING modifiers are listed in Table 8-2.

Further explanations and examples of the use of INPUT USING operators and modifiers follow.

Table 8-1

INPUT USING OPERATORS

| Operator | Meaning |
|----------|--|
| nA | Accept string data item of length n characters. |
| nB | Accept numeric data item in binary representation. |
| nD | Accept numeric data item in decimal representation. |
| nE | Accept numeric data item in scientific notation. |
| n.mG | Accept numeric data item in floating point representation. |
| nH | Accept numeric data item in hexadecimal representation. |
| I | Accept item in binary item format. |
| J | Accept numeric data item in engineering notation. |
| K | Accept numeric data item in any format. |
| N | Accept numeric or string data item. |
| nO | Accept numeric data item in octal representation. |
| nX | Skip n characters. |
| n% | Accept the leftmost n bits of a GPIB byte or pair of bytes as the rightmost n bits of a 4041 integer, using binary block format. |
| n@ | Accept the leftmost n bits of a GPIB byte or pair of bytes as the rightmost n bits of a 4041 integer, using binary block format. |

Table 8-2

INPUT USING MODIFIERS

| Modifier | Meaning |
|----------|--|
| n() | Replicate enclosed image string n times. |
| F | Free-format: accept characters until a delimiter is received. |
| R | ("E" and "G" operators only) Alternate radix: use comma for radix symbol instead of decimal point (must use elements from a DELN clause that cannot include a comma as a delimiter). |
| V | ("%" operator only) accept data in "data-driven" mode. |

INPUT

The "A" Operator. Accept one data item of length n characters.

Example:

```

100 Input using "5a":string$
110 Print string$
run
(user input) ABCDEFGHIJ
(output)     ABCDE

```

The "B" Operator. Accept an n -digit data item in binary representation.

The value input is interpreted as a binary number, right-justified (with left-zero fill) within an n -space field.

The user may input a sign (default: +) and up to 32 digits for the number.

Numbers that can be input using the "B" operator range from $-(2^{32}-1)$ to $+(2^{32}-1)$.

Numbers input in the range from 2^{31} to $2^{32}-1$ are stored in the range from $-(2^{31})$ to -1 , respectively.

Example:

```

*long x
*input using "32b":x
10000000000000000000000000000000 (user input)
*print x
-2.147483648E+9
*input using "32b":X
11111111111111111111111111111111 (user input)
*print x
-1.000000

```

Numbers input in the range from $-(2^{31} + 1)$ to $-(2^{32})$ are stored in the range from $+(2^{31})$ to 1 , respectively.

Example:

```

*long x
*input using "33b":x
-10000000000000000000000000000001 (user input)
*print x
2.147483647E+9
*input using "33b":x
-11111111111111111111111111111111 (user input)
*print x
1.000000

```

This input scheme allows the user to express binary numbers in either signed two's complement or unsigned 32-bit notation.

The "D" Operator. Accept an n -digit data item in integer format.

The "D" operator delimits input when n characters have been received, when a character other than a digit, monadic plus, or monadic minus is input, or on the delimiters specified in a DELN clause (default: space, tab, comma, semicolon, colon, and the EOM character).

Example:

```

100 Input using "3d":num
110 Print num

```

NOTE

The "D" operator treats decimal points in the input stream in a special way. A decimal point is considered a terminator, not a delimiter, by the "D" operator.

Example:

```

*delete var num1,num2
100 Input using "2(3d)":num1,num2
110 Print num1
120 Print num2
RUN
(user input) 123.456
(result)    *** ERROR # 751 LINE # 100

```

The decimal point terminated the input in line 100. The solution is to use the "X" operator to skip over the decimal point in the input stream. Change line 100 to read

```

100 Input using "3d,x,3d":num1,num2
run
(user input) 123.456
(output)    123.0
           456.0

```


The “E” Operator. Accept an n-digit data item in scientific notation.

The “E” operator delimits input when n characters have been received, when a “non-scientific-notation” character is received (scientific notation characters include digits 0-9,., monadic + or –, and “E” or “e”), or on the delimiters specified in a DELN clause (default: space, tab, comma, semicolon, colon, and the EOM character).

If the nth character of the input string is an E, or if no E appears in the input string, the number is interpreted as a floating-point number (long-floating-point, if more than six digits have been input).

Example:

```

100 Input using "7e":num
110 Print num
run
(user input) 1.234E9
(output)     1.234E+9

```

The “G” Operator. Accept a numeric data item in floating point representation. The command format is “n.mG”, where “n” and “m” signify to accept at most n characters (total) and at most m characters after the radix symbol.

The “G” operator delimits data items when n characters (total) have been received, or when m characters have been received after the radix symbol, or when a non-numeric character is input (where the numeric characters are digits 0-9,., and monadic + or –), or on the delimiters specified in the DELN clause (default: space, tab, comma, semicolon, colon, and the EOM character).

The “G” INPUT USING operator truncates values received on input, according to the value of the m modifier.

Example:

```

100 Input using "6.3g":num
110 Print num
run
(user input) 12.34567
(output)     12.345

```

The “H” Operator. Accept an n-digit data item in hexadecimal representation.

The value input is interpreted as a hexadecimal number, right-justified (with left-zero fill) within an n-space field.

The user may input a sign (default: +) and up to 8 digits for the number.

Numbers that can be input using the “H” operator range from $-(2^{32}-1)$ to $+(2^{32}-1)$.

Numbers input in the range from 2^{31} to $2^{32}-1$ are stored in the range from $-(2^{31})$ to -1 , respectively.

Example:

```

*long x
*input using "8h":x
80000000 (user input)
*print x
-2.147483648E+9
*input using "8h":x
FFFFFFFF (user input)
*print x
-1.000000

```

Numbers input in the range from $-(2^{31} + 1)$ to $-(2^{32})$ are stored in the range from $+(2^{31})$ to 1 , respectively.

Example:

```

*long x
*input using "9h":x
-80000001 (user input)
*print x
2.147483647E+9
*input using "9h":x
-FFFFFFFF (user input)
*print x
1.000000

```

This input scheme allows the user to express hexadecimal numbers in either signed two’s complement or unsigned 32-bit notation.

INPUT

The "I" Operator. Accept a data item in ITEM format. (All data elements in the input list must be in ITEM format.)

ITEM format is the internal representation used by the 4041 to store data. Data stored on external devices in ITEM format require less room to store than ASCII files. Data input in ITEM format also takes less time to store inside the 4041, since the data need not be translated from ASCII into the 4041's internal representation.

ITEM-format data exchanges are strictly type-checked. The data formats of incoming and outgoing data must match exactly, or an error is generated. Thus, data stored on a DC-100 tape in integer format cannot be input directly to short-floating-point variables.

The TYPE function, which returns a number indicating the type of the next data item to be read in from a tape file, can be used to check ITEM data before it is input. (See the description of the TYPE function in Section 14, *DC-100 Tape*, for more information.)

Example 8-2 writes five data items in item-format into a file called "Dat", then reads them back out again.

The "J" Operator. Accept data item in engineering notation.

The suffixes used for engineering notation are as follows:

- P — pico (1E-12)
- N — nano (1E-9)
- U — micro (1E-6)
- M — milli (1E-3)
- K — kilo (1E3)

Example:

```

100 Input using "j":a
110 Print a
(user input) 1.23M
(output) 0.00123

```

The "K" Operator. Accept data until a non-decimal character is received. Decimal characters are digits 0-9, radix symbol, monadic + or -, and "E" or "e" used to denote exponent.

Binary, octal, and hexadecimal data can be entered if the following notation is used:

```

Binary = [d..dB]
Octal = [d..dO]
Hex = [d..dH]

```

The "K" operator delimits input on non-decimal characters, or on the "]" of a binary, octal, or hexadecimal number, or on the delimiters specified in a DELN clause (default: space, tab, comma, semicolon, colon, and the EOM character).

Example:

```

100 Integer a
110 Input using "k,k":a,b
120 Print a,b
130 End
run
(user input) 56,3.106E5
(output) 56<tab>3.106E+5

```

The "N" Operator. Similar to the "K" operator except that string data can be entered as well as numeric data. The delimiters specified in a DELS clause delimit input for string data items (default: the EOM character).

Example 8-3 inputs three data items using the "n" operator.

Note that the exclamation points used to delimit string data elements do not become part of the inputted strings.

```

100 ! PRINT THE ITEM-FORMAT DATA TO A TAPE FILE
110 Open #1: "dat(open=rep,clip=yes,for=ite)"
120 Print #1 using "5(i)": "HELLO",1,2,3,"GOOD-BYE"
130 Close all
140 ! NOW READ IT BACK OUT AGAIN
150 Open #1: "dat(for=ite)"
160 Integer a,b,c
170 Input #1 using "5(i)":str1$,a,b,c,str2$
180 Print str1$,a,b,c,str2$
190 End
run
(output) HELLO<tab>1<tab>2<tab>3<tab>GOOD-BYE

```

Example 8-2.

The "O" Operator. Accept an n-digit data item in octal representation.

The value input is interpreted as an octal number, right-justified (with left-zero fill) within an n-space field.

The user may input a sign (default: +) and up to 11 digits for the number.

Numbers that can be input using the "O" operator range from $-(2^{32}-1)$ to $+(2^{32}-1)$.

Numbers input in the range from 2^{31} to $2^{32}-1$ are stored in the range from $-(2^{31})$ to -1 , respectively.

Example:

```
*long x
*input using "11o":x
2000000000 (user input)
*print x
-2.147483648E+9
*input using "11o":x
3777777777 (user input)
*print x
-1.000000
```

Numbers input in the range from $-(2^{31} + 1)$ to $-(2^{32})$ are stored in the range from $+(2^{31})$ to 1 , respectively.

Example:

```
*long x
*input using "12o":x
-20000000001 (user input)
*print x
2.147483647E+9
*input using "12o":x
-3777777777 (user input)
*print x
1.000000
```

This input scheme allows the user to express octal numbers in either signed two's complement or unsigned 32-bit notation.

The "X" Operator. Skip past the next n characters in the input stream.

In Example 8-4, one numeric value is input, then two characters of the input stream are shipped before the next numeric value is input.

```
100 Input using "n,n,n" dels "!":str1$,num,str2$
110 Print str1$
120 Print num
130 Print str2$
140 End
run
(user input) HELLO THERE!123,GOOD-BYE!
(output) HELLO THERE
123.0
GOOD-BYE
```

Example 8-3.

```
100 Input using "3d,2x,3d":num1,num2
110 Print num1,num2
run
(user input) 12345678
(output) 123.0<tab>678.0
```

Example 8-4.

INPUT/OUTPUT

INPUT

The “%” Operator (GPIB). Accept data item in binary block argument format.

Binary block argument format is a data transfer format defined by the Tektronix GPIB Codes and Formats standard to facilitate rapid data transfers between devices on the GPIB.

When information is exchanged using binary block argument format, the talking device sends the following information over the GPIB after ATN becomes unasserted:

1. A percent sign (%).
2. A two-byte binary count, telling the receiving devices how many data bytes will be sent in the upcoming data transfer.
3. A sequence of bytes representing the data.
4. A checksum byte, equal to the two's complement of the modulo-256 sum of all preceding data bytes sent since the first percent sign.

The “%” operator can be used to transfer one or more integer scalars or integer arrays from a device on the GPIB to the 4041. Only integer scalars or integer arrays can be transferred using the “%” operator.

In addition, the “n” modifier can be used to control the way that integers received are stored. The way this is done is explained below.

The 4041 stores integers in 2 bytes of memory (16 bits), in 2's complement representation. Thus, numbers from -32768 to +32767 can be stored in a 4041 integer.

However, the GPIB has only eight data lines. Therefore, transmitting a 16-bit value to the 4041 over the GPIB data lines requires that two GPIB data bytes be transmitted. By convention, the 4041 assumes that the integer's most significant byte is transmitted first.

The “n” modifier, in conjunction with the “%” operator, tells the 4041 how many bits of each byte or pair of bytes to store. The default value for “n” is 16; thus, using the default value causes the 4041 to store each pair of data bytes it receives as a 2-byte integer, most significant byte first.

If the value of the “n” modifier is less than 16 but greater than 8, the 4041 takes the n leftmost bits of each 2-byte data value received, and makes them the n rightmost bits of an integer to be stored in memory.

Example:

```
100 Open #1:"gpib(pri=1):"  
110 Dim bufr$ to 1000  
120 Input #1 using "+12%" buffer bufr$:a
```

These lines produce the following results upon execution:

1. The 4041 sends the UNLISTEN interface message over the GPIB (to clear any stray listeners off the bus).
2. The 4041 sends its listen address and the talk address of the device at primary address 1 over the GPIB.
3. The 4041 then expects to receive the following values over the GPIB:
 - a. A percent sign
 - b. Two bytes signifying the number of bytes that the talking device will transmit
 - c. The data bytes
 - d. A checksum byte sent with EOI asserted.
4. The 4041 then sends the UNTALK and UNLISTEN interface messages over the GPIB.
5. The 4041 takes the first data byte received and the leftmost four bits of the second data byte received and stores these as the rightmost twelve bits of integer variable A. If the talking device sent more than two data bytes, the remaining data bytes are stored in string variable Bufr\$.

If the “n” modifier is less than or equal to 8, the 4041 stores the n leftmost bits of each GPIB data byte received as the n rightmost bits of a 4041 integer. (Note that when $n > 8$, the 4041 reads the values of two GPIB data bytes for each integer; when $n \leq 8$, it only reads one.)

If the input list contains an integer array variable, the entire integer array is input using the format described above, i.e., a percent sign is received, followed by two bytes signifying the number of bytes to be transferred, followed by the array values, followed by a checksum sent with EOI asserted.

In addition to the “n” modifier, the value of the “±” modifier determines the way in which the 4041 stores the data received over the bus.

If the “+” modifier is included in the USING clause, the n leftmost bits of the data received are stored as the n rightmost bits of a 16-bit 4041 integer. Bits to the left of the nth position from the right are given a value of 0. (Thus, incoming data bytes are stored as positive integers.)

If a “+” value is not included in the USING clause, the value of the “±” modifier defaults to “-”. In this case, the n leftmost bits of the data received are stored as the n rightmost bits of a 16-bit 4041 integer, AND the value of the nth bit from the right is copied into all bits to the left of the nth bit. (Thus, incoming data bytes are stored as signed integers in 2’s complement representation.)

Example:

```
100 Open #1:"gpib0(pri=1):"
110 Integer a
120 Input #1 using "8%":a
130 Print a
```

When the 4041 receives the input called for in line 120, it stores the first byte received after the binary byte count as a signed integer (default). The value of the 8th bit (from the left) of the first byte received over the GPIB will be replicated in all bits to the left when the 4041 stores the number.

To transfer values into an input list more than one element long, use the “n” modifier with parentheses around the “%” operator to indicate the number of input elements to be transmitted using binary block argument format. When inputting more than one element through one INPUT statement using binary block argument format, the 4041 treats each element of the input list as if it were a separate INPUT statement.

Example:

```
150 Open #16:"gpib0(pri=16):"
160 Integer a(10),b(5)
170 Input #16 using "2(%)":a,b
```

The 4041 treats line 170 as if it were executing the following lines:

```
input #16 using "%":a
input #16 using "%":b
```

The 4041 does not send talk addresses when the logical unit from which it is receiving data is open to no primary address (as, for example, with the statement OPEN #1:“GPIB0:”). In this case, the 4041 simply listens for bytes being put onto the bus by a previously talk-addressed device.

When the “%” operator is used in combination with other INPUT USING operators, the binary block data should be preceded by and followed by the EOA character, unless the binary block data item is the last element on the input list, in which case it should be followed by the EOM character.

Thus, if the first data byte received over the GPIB is HEX E8, the 4041 stores the value HEX FFE8 in the two bytes reserved for integer variable A.

Line 130 would then print the value -24 on the system console device, since HEX FFE8 is the 16-bit 2’s complement representation of -24.

NOTE

If you get an error #757 (Invalid Header), the instrument your 4041 is connected to is probably not sending the correct EOA character to delimit the header before the “%” sign. Check the documentation on your instrument to find out what it sends to delimit the header, then make that character the EOA character with an OPEN statement.

The following program segment reads a block of 1000 integers preceded by a 10-character ASCII header from a device on the GPIB, using the binary block argument format.

In Example 8-5, the binary-block transmission returned by an instrument in response to the prompt “curve?” is stored as a sequence of positive integers in numeric array B. Each integer transmitted contains eight bits.

```
100 Dim a$ to 10 ! a$ receives the header
110 Integer b(1000) ! array b receives the data
120 Dim c$ to 1100 ! c$ is a temporary buffer
130 Open #1:"gpib0(pri=1):"
140 ! input the header in response
150 ! to the GPIB message "curve?"
160 Input #1 prompt "curve?" buffer c$ using "fa,+8%":a$,b
```

Example 8-5.

INPUT

The “@” Operator. Accept a data item in end block argument format.

End block argument format is a data transfer format defined by the Tektronix GPIB Codes and Formats standard to facilitate rapid data transfers between devices on the GPIB.

When information is exchanged using end block argument format, the talking device sends the following information over the GPIB after ATN becomes unasserted:

1. An “at” sign (@).
2. A sequence of bytes representing the data. The last data byte is sent with EOI asserted, to indicate the end of the data transfer.

Like the “I” operator, data transmitted using the “@” operator should be preceded by the EOA character. However, the “@” operator does not require that the EOA character follow the data.

Modifiers for the “@” INPUT USING operator are identical to those for “%” INPUT USING operator.

The “n” Modifier. Designates the number of character positions in an input field, or the number of times a parenthesized image string segment is to be repeated.

When used to designate the number of times a parenthesized image string is to be repeated, n must be an integer < = 511.

In Example 8-6, the USING operator “5(2d)” indicates that five two-digit numeric values are to be input.

The “F” Modifier. Accept characters until a delimiter is received. The default delimiters for numeric data elements are: space, comma, tab, semicolon, colon, and the EOM character. The default delimiter for string data elements is the EOM character.

The only limits on the number of characters that can be received for a single data element using the “F” modifier are the dimension of the string (for string data elements), the size of the number that can be accommodated (for numeric data elements), and the size of the input buffer (default: 514 bytes).

Example:

```

100 Input using "2(fg)":num1,num2
110 Print num1
120 Print num2
run
(user input) 123.456,789.012
(output)     123.456
              789.012

```

```

100 Input using "5(2d)": num1,num2,num3,num4,num5
110 Print num1
120 Print num2
130 Print num3
140 Print num4
150 Print num5
run
(user input) 1234567890
(output)     12.0
              34.0
              56.0
              78.0
              90.0

```

Example 8-6.

The “R” Modifier. Accept a numeric data item using comma for a radix symbol instead of decimal point.

The “R” modifier can be used only with the “E” and “G” operators.

NOTE

When using the “R” modifier for input, the user should include a DELN clause in the INPUT statement to specify that the comma is NOT a delimiter for numeric input elements.

In Example 8-7, line 100 requests numeric input with a comma (not a decimal point) used as a radix symbol.

Line 110 prints the number with a decimal point for a radix symbol, and line 120 prints the number with a comma for a radix symbol.

Note that the “G” operator truncates (not rounds) excess characters on input.

The “V” Modifier. (“%” operator only) accept data in “data-driven” mode.

By default, binary block data exchanges occur in “argument-driven” mode. This means that the next argument in the argument list is not filled with data until the current argument is filled with data.

In the data-driven mode, the next argument is filled with data when the 4041 completes the binary block transfer for the current argument.

Example:

Suppose the data consist of two 512-byte chunks, to be transferred in binary block format. The following statements:

```
100 Integer a(768),b(512)
110 Input using "%,%":a,b
```

fills all of array A and half of array B (the first 512 bytes are transferred to A; since A is not yet full, 256 bytes from the next chunk are transferred to A to fill it, and the remaining bytes are transferred to B).

With the V-modifier, however, the following statements:

```
100 Integer a(768),b(512)
110 Input using "v%,v%":a,b
```

put the first chunk into array A (filling 512 of its 768 bytes), and the second chunk into array B (filling it completely).

```
100 Input using "r6.4g" deln " ":num
110 Print num
120 Print using "r6.4g":num
run
(user input) 1,23456
(output) 1.2345
1,2345
```

Example 8-7.

The OPEN Statement

Syntax Form: [line-no.] OPEN #numexp:strex[,strvar]

Descriptive Form: [line-no.] OPEN #lunum:stream-spec [,open-result-for-tape-files]

PURPOSE

The OPEN statement associates a logical unit number with a stream spec for subsequent I/O operations.

EXPLANATION

When a logical unit number is associated with a stream spec, all subsequent I/O statements using that logical unit number perform their I/O function as directed by the stream spec. The logical unit number becomes a "shorthand" form of the stream spec.

The logical unit number must evaluate to an integer between 0 and 32767 inclusive.

If no driver spec is included in a stream spec, the current SYSDEV driver spec is used by default.

If a file is being opened on the DC-100 tape, a string variable may be specified after the stream spec (and separated from it by a comma) to receive the directory entry for the opened file. If a string variable is specified for an OPEN on any other driver, the string is set to null.

Attempting to open a logical unit that is already open closes the logical unit, then re-opens it with new stream attributes.

Example

```
3110 Open #1:"gpib(pri=21,sec=4):"
```

This statement designates an instrument at primary address 21 and secondary address 4 on the currently selected GPIB as logical unit 1. This number can be used to direct subsequent I/O to this device.

```
4781 Open #2:"test",openres$
4782 Print openres$
```

Line 4781 designates a file called "Test" on the DC-100 tape as logical unit 2, and stores the directory entry for file "Test" in OpenRes\$.

Line 4782 directs the 4041 to print the value of OpenRes\$ on the system console device.

Defining Stream Spec Parameters During Execution

It is often desirable to define stream spec parameters during execution. To do this, the user must concatenate several substrings that together make up a legal stream spec. The resulting string can be used directly in I/O statements or associated with a logical unit number by means of the OPEN statement.

Example

In Example 8-8, line 100 asks the user to input a number representing a primary address of a device, and stores it in numeric variable Primary.

Line 110 concatenates the string "GPIB(PRI = " with the string equivalent of numeric variable Primary and with the string "):", then associates the stream spec thus formed with the logical unit given by the value of numeric variable Primary.

Thus, if the number input in line 100 were "1", line 110 would associate logical unit 1 with the stream spec "GPIB-(PRI = 1):".

Opening Tape Files

The OPEN parameter of the file spec determines how a tape file is handled when it is opened.

When OPEN = REPLACE, the file is opened for writing. The file is deleted (if it exists already) from the tape directory in memory, and a new file is created with the tape drive's read/write head positioned at the beginning of the file. Data items output to the file over-write the previous contents of the file (previous contents are lost).

When OPEN = NEW, the file is treated in exactly the same way as OPEN = REPLACE, except that an error is generated if the file already exists on the tape.

When OPEN = UPDATE, the file is opened for writing. The tape drive's read/write head is positioned at the end of the data in the file. Data items output to the file are added to the end of the file (both previous and new file contents are preserved).

When OPEN = OLD (default), the file is opened for reading as well as writing. File contents are read sequentially, until a data item is output to the file. At that time, the file is "closed" for reading; data items output to the file over-write the previous contents of the file after the point at which writing began (previous contents of the file are lost after the first data item output).

Error Recovery

If you mistakenly open a file with OPEN parameter equal to REPLACE when you meant OLD, the file will be deleted from the copy of the tape directory stored in memory. All is not lost, however.

BEFORE you start writing to the new file, "pop" the tape (remove it from the tape drive).

NOTE

NEVER "pop" a tape while data is being written to it. You might make one record irrecoverable (if the 4041 is not writing out the tape header or directory), or make the data on the WHOLE TAPE irrecoverable (if the 4041 is writing the tape header or directory).

After popping the tape, execute a DISMOUNT statement in immediate mode. You are now back where you started.

```
100 Input prompt "Primary Address:":primary
110 Open #primary:"gpib(pri=" & str$(primary) & "):"
```

Example 8-8.

The PRINT Statement

| | |
|-------------------------|---|
| Syntax Form | [line-no.] PRINT [clause-list:] {numexp} [,numexp] . . . {strex} [,strex] . . . |
| where | clause-list = [# {numexp}] [BUFFER strvar] [INDEX {numexp}] [USING {numexp}; {strex} {strex} {strex} |
| Descriptive Form | [line-no.] PRINT [print-clauses:]print-list |

PURPOSE

The PRINT statement transfers data from variables in memory to a device.

EXPLANATION

The PRINT statement transfers data items using default data formatting rules (explained below). The PRINT USING statement allows the programmer to specify the printing format by means of an image string.

The PRINT statement sends data to the system console device by default. The user can specify which device is to be the system console and set its logical parameters by means of the SET CONSOLE statement (described elsewhere in this section).

Data is sent to devices other than the system console by means of the “#” clause (described later in this section).

If no USING clause is specified, the PRINT statement converts data items into ASCII character strings and outputs them to a peripheral device according to the following rules:

1. Numeric expressions are evaluated and reduced to a numeric constant. The constant is converted to ASCII, using the most abbreviated format possible for the specific data type, and sent to the peripheral device.
2. String expressions are evaluated and reduced to a string constant. The string constant is sent directly to the peripheral device.
3. Numeric arrays are converted to ASCII element-by-element and sent to the peripheral device.
4. String arrays are sent to the peripheral device element-by-element. Each element is treated as a string expression.

5. Commas are used to separate message units. A comma in a print list sends the End-of-Message-Unit character (EOU) to the output device. The default EOU character for the GPIB is a SEMICOLON; the default EOU character for other devices is a TAB.

A print element must appear on either side of a comma in a print list to be syntactically correct. Beginning or ending a print list with a comma or putting two or more consecutive commas in a print list results in an error.

6. Semicolons separate print elements within message units, or suppress output of the End-of-Message (EOM) character.

The first semicolon in a message unit sends the End-of-Header (EOH) character to the output device. The second and succeeding semicolons send the End-of-Argument (EOA) character to the output device.

A semicolon after an array name in a print list sends EOA characters between array elements. If the array is the last element in the print list, the semicolon suppresses output of the EOM character(s) after the last element.

EOH defaults to SPACE for the GPIB driver, and defaults to no output for all other devices. EOA defaults to COMMA for the GPIB driver, and defaults to no output for all other devices.

If a semicolon concludes a print list, it sends neither EOH nor EOA, but suppresses the EOM normally output after a PRINT statement is executed. (The effect on a computer terminal, for example, is to suppress carriage-return/line-feed and leave the cursor at the end of the printed line.)

A semicolon concluding a print list in a statement outputting data to a GPIB device also suppresses assertion of the EOI line indicating end-of-message.

- After all print list elements are processed, the End-of-Message (EOM) string is sent to the output device. The default EOM string for the GPIB is <cr> <lf> & <EOI> (carriage return followed by line feed, with the EOI line on the GPIB asserted). The default EOM character for other devices is <cr> (carriage return).

If a semicolon terminates the print list, no End-of-Message string is set.

Examples

```
3130 Print
```

This statement outputs one end-of-message string to the system console device. If the system console device is a user's terminal connected through an RS-232-C interface port, this statement prints a carriage-return/line-feed; if the system console device is the front panel/keyboard, this statement erases the contents of the front panel.

```
3131 Print ;
```

This line generates no output (semicolon terminating print list containing no data elements).

```
3132 Print 1,"word";2
```

This line produces the following output:

```
1 <tab> word2
```

```
3133 Print 1;" word ";3
```

This line produces the following output on the system console device:

```
1 word 3
```

USE OF PRINT STATEMENT WITH GPIB, OPT 2, OPT 3, OR DRIVERS

The use of the PRINT statement to output to GPIB devices is described under *High-Level Message Transfers* in Section 9, *Instrument Control With GPIB*.

The OPT2 driver does not use the PRINT statement for output. Instead, it uses the "OPT2OUT" romcall, described along with the RCALL statement in Section 7, *Control Statements*.

Using the PRINT statement to output to a tape file has varying results depending on the value of the OPEN parameter in the tape file stream spec.

If OPEN = NEW or OPEN = REPLACE, the tape's read/write head is positioned at the beginning of the file. Output to the file writes over information previously in the file; data in the file from before the PRINT operation are lost.

If OPEN = UPDATE, the tape's read/write head is positioned at the end of the data presently in the file. Output to the file adds information to the end of the file; data in the file from before the PRINT operation remain available for later retrieval.

If OPEN = OLD, the tape's read/write head is positioned at the beginning of the file. The file is then open for reading until the first PRINT statement calls for output to the file, at which time the file is closed for reading and opened for writing. (All other logical units that may have opened the file for reading must be closed before the first output to the file occurs, or an error is generated.)

A file can only be opened for output on one logical unit at a time. A file cannot be opened for input and output at the same time.

PROCEED MODE PRINT

The 4041 enters proceed-mode upon execution of a SET PROCEED 1 statement. In proceed-mode, program execution continues while one or more drivers complete the tasks assigned to them by an INPUT or PRINT statement, or until a subsequent statement requires that a previous I/O operation be complete, or until a proceed-mode I/O error occurs.

Example

```
100 Set proceed 1
110 Open #1:"datafi(open=replace,clip=yes)"
110 Print #1:array1,array2,array3
120 A=1
130 B=2
140 C=3
150 Print #1:a,b,c
```

Line 110 writes the variables Array1, Array2, and Array3 into file "DataFi" on the DC-100 tape. At the same time that the output is being written to the tape, the 4041 executes lines 120 through 140. Since line 150 calls for output to logical unit 1, the 4041 stops at line 150 until the PRINT operation caused by line 110 is complete.

PRINT**Restrictions on PRINT in Proceed Mode**

When using the PRINT statement in proceed mode, the 4041 only "proceeds" when it has one buffer-full or less of data to be output.

If no BUFFER clause is included in the PRINT statement, the 4041 assigns a temporary buffer 514 bytes long to hold output data. Program execution does not "proceed" (i.e., succeeding program statements do not begin to execute) until there are 514 bytes of data or less to be output.

Therefore, if there were 1,000 bytes of data to be output by a PRINT statement, the 4041 would output the first 514 bytes in "sequential" mode, then proceed to execute other statements as it output the last 486 bytes.

The way to overcome this restriction is to dimension a buffer large enough to contain the entire string of output data, then use that buffer in a BUFFER clause within the PRINT statement.

Example. Suppose we wish to output three arrays containing 1100 characters of data, total. The following program segment would allow the 4041 to proceed during the output operation:

```
100 Set proceed 1
110 Dim bufr$(1200)
120 Print buffer bufr$:array1,array2,array3
130 ! PROGRAM PROCEEDS FROM HERE
```

Line 120 reads the characters to be output into string variable Bufr\$, then proceeds to execute succeeding program statements while printing the contents of Bufr\$.

The 4041 will not proceed past a statement calling for proceed mode I/O on a driver that is already busy (even if the second statement invokes a different logical unit from the first).

The ASK("IODONE") Function

The status of I/O operations on a given logical unit at any time can be determined by means of the ASK("IODONE") function. See the description of the ASK("IODONE") function in Section 5, *Environmental Control*, for more information.

The IODONE Condition

Control can be transferred to a given subprogram or GO-SUB subroutine upon completion of proceed-mode I/O on a given logical unit by means of the ON IODONE statement.

The IODONE condition is automatically enabled when the 4041 is put into proceed mode.

See Section 14, *Interrupt Handling*, for more information about the ON statement and the IODONE condition.

The ASK("PROCEED") Function

Whether or not the 4041 is in proceed mode can be determined by means of the ASK("PROCEED") function. ASK("PROCEED") returns a value of 1 if the 4041 is in proceed mode, 0 if not.

Errors During Proceed-Mode PRINT

Errors occurring during proceed-mode PRINT are handled in a special way. See the description of the ERROR interrupt condition in Section 14, *Interrupt Handling*, for more information.

PRINT Clauses

Clauses are "statements within statements" used with GETMEM, INPUT, PRINT, PUTMEM, RBYTE, and WBYTE commands to define a data transfer path or modify execution of an I/O operation. Clauses are separated from data lists by a colon.

The following clauses can be used with PRINT statements:

| | |
|--------|---|
| # | Specifies a logical unit number or stream spec for the PRINT operation. |
| BUFFER | A specified string is used directly as an I/O buffer. |
| USING | Data formatting is under user control. |

Further explanation and examples showing the use of each clause follow.

The # Clause

Syntax Form:

#numexp OR #strexp

Descriptive Form:

#logical-unit-number OR #stream-spec

The # clause specifies a data path for an I/O operation.

The # clause is used when I/O is to be performed to other than the default device for an I/O operation, or when the user wishes to specify other than the default attributes for a driver or file.

If a numeric expression follows the “#”, it is interpreted as the number of the logical unit to be used for the PRINT operation.

If a string expression follows the “#”, it is interpreted as the stream specification to be used for the PRINT operation.

The numeric expression for a logical unit must evaluate to an integer in the range $0 \leq \text{lunum} \leq 32767$.

A string expression following the “#” must evaluate to a legal stream spec. (See the description of stream specs, near the beginning of this section, and Appendix D for more information.)

Example.

```
2048 Print #5:x,y,z
```

This statement sends the values of three numeric variables, X, Y, and Z, to the device opened as logical unit 5.

```
4096 Print #"prin:":zip$;zap
```

This statement sends the values of a string variable, ZIP\$, and a numeric variable, ZAP, to the thermal printer.

The BUFFER Clause

Syntax Form:

BUFFER str-var

Descriptive Form:

BUFFER string-variable-for-output-data

The BUFFER clause allows a user to specify a string variable to use as a buffer during a data transfer.

When no BUFFER clause is included in a PRINT statement, the 4041 makes a temporary 514-byte buffer into which data are read in their output-formatted form. When the buffer is full or all data to be output have been written into the buffer, the buffer's contents are sent to the output device. The contents of the buffer are then erased.

The BUFFER clause allows a user to specify a string variable into which output data is read during a PRINT operation. After the PRINT operation is complete, the buffer's contents are not erased, but remain available for editing or other uses.

In addition, the BUFFER clause can be used to advantage with PRINT statements during proceed-mode I/O. See “Proceed-Mode PRINT”, later in this section, for more information.

Example.

```
100 Read a,b,c,d,e
110 Data 1,2,3,4,5
120 Print buffer bufr$:a,b,c,d,e
130 Print bufr$
run
(output) 1.0<tab>2.0<tab>3.0<tab>4.0<tab>5.0
1.0<tab>2.0<tab>3.0<tab>4.0<tab>5.0
```

Note how string variable Bufr\$ contains the exact sequence of characters that were output.

The INDEX Clause

Syntax Form:

```
INDEX {numexp}  
      {strex}
```

The INDEX clause will be included in a future enhancement, for use with peripheral devices that support random access.

The word "INDEX" can be used as a line label, variable, or subprogram name in 4041 programs.

The USING Clause

Syntax Form:

```
USING {line-reference}  
      {strex}
```

Descriptive Form:

```
USING {line-number-of-IMAGE-statement}  
      {format-string}
```

When used with the PRINT statement, the USING clause allows the user to specify the format in which data are printed, displayed, or otherwise sent to an output device.

The USING clause specifies a format string, which defines a format for output. The format string may be any combination of string literals, string variables, string functions, or operations with the string operator. The string expression must evaluate to a legal format string.

The USING clause either contains a format string, which defines a format for output, or refers to an IMAGE statement that contains a format string. In either case, the format string may be any combination of string literals, string variables, string functions, or operations with the string operator, as long as it forms a string expression that evaluates to a legal format string.

Legal PRINT USING operators and modifiers are listed in Tables 8-3 and 8-4.

Further explanations and examples of the use of each of these operators follow.

Table 8-3
PRINT USING OPERATORS

| Operator | Meaning |
|----------|--|
| nA | Output n characters in string, left-justified, space fill. |
| nB | Output binary representation of number; set n digit positions; do not output square brackets or "B". |
| nD | Integer format: set n digit positions, leading space fill. |
| nE | Scientific notation format: set n digit positions. |
| n.mG | Floating point format; set n total character positions (including radix symbol), with m digits to right of decimal point. |
| nH | Output hexadecimal representation of number; set n digit positions, do not output square brackets or "H". |
| I | Print data in binary item format. |
| nJ | Engineering notation format: set n digit positions, leading space fill. |
| K | Output numeric data item in shortest form. |
| nL | Output n ASCII LF characters. |
| N | Print string or numeric data item in shortest format (identical to K, except N allows strings). |
| nO | Output octal representation of number; set n digit positions; do not output square brackets or "O". |
| S | Suppress carriage return. |
| nT | Output spaces until character position n reached. |
| nX | Output n spaces. |
| ' | Begin/end imbedded string. |
| n() | Replicate enclosed image string n times. |
| n/ | Output n end-of-message strings. |
| : | Output colon. |
| ; | Output semicolon. |
| n% | Output the rightmost n bits of a 4041 integer as the leftmost n bits of a GPIB byte or pair of bytes, using binary block format. |
| n@ | Output the rightmost n bits of a 4041 integer as the leftmost n bits of a GPIB byte or pair of bytes, using end block format. |

Table 8-4
PRINT USING MODIFIERS

| Modifier | Meaning |
|----------|--|
| n | Specifies (a) number of character positions in print field, (b) number of times a field operator is repeated, or (c) number of rightmost bits to "pack" into GPIB byte(s) in binary or end block format. |
| F | Set a print field (string or numeric) just large enough to accommodate data item. |
| R | Output comma as radix symbol. |
| + | Output sign: + if positive, - if negative. |
| - | Output sign: SPACE if positive, - if negative (default). |
| Z | Leading zero fill. |
| * | Leading "*" fill. |
| < | Left-justify data in a field ("A" format only; default). |
| > | Right-justify data in a field ("A" format only). |
| = | Center data within a field ("A" format only). |

The “A” Operator. Output n characters in a string, left-justified, space fill.

This operator sets up a field n characters long to receive an alphanumeric string. Characters are written into the field starting with the leftmost character position. Unused field elements are filled with blank spaces. If n is not specified, a field size large enough to hold the entire string is used.

Example:

```
100 A$="Name:"
110 Print using "10a":a$
```

Line 110 produces the following output:

```
Name: <sp> <sp> <sp> <sp> <sp>
```

The “B” Operator. Output the binary representation of a number.

A field of print spaces may be reserved for the data element by means of the “n” modifier. This field must be large enough to accommodate both a sign (if the number is negative) and the digits that make up the number.

The output range for PRINT USING with the “B” operator is from $-(2^{32}-1)$ to $+(2^{32}-1)$.

When a specified length is allotted to a print-list element being printed with the “B” operator, the allotted field is left-zero filled.

Example:

```
*x=5
*print using "8b":x
00000101
*x=-5
*print using "8b":x
-0000101
```

The “D” Operator. Output a number in integer format in n digit positions, leading space fill.

This operator sets up an n-digit print field for a numeric variable. Numbers are rounded to the nearest integer and printed right-justified within this field. If a number does not fit within its designated space, a string of *’s is output.

Example:

```
100 Number=155
110 Print using "5d":number
```

Line 110 produces the following output on the system console device:

```
<sp> <sp> 155
```

The “E” Operator. Output a number in scientific notation. The user may specify a number of digits n (minimum:3) to be printed in the mantissa.

If n = 3, the 4041 outputs one character for the sign of the mantissa, one character for the mantissa’s first digit, and one character for the radix symbol.

If n > 3, n characters are output for the mantissa, including one for the sign and one for the radix symbol.

In addition, the 4041 outputs an upper-case “E”, a sign for the exponent, and as many digits as are required for the exponent.

If n = 0, the entire number is output in floating-point format.

Example:

```
490 A=3.4567E12
500 Print using "3e":a
```

Line 500 causes the following output to appear on the system console device:

```
<sp>3.E+12
```


The “G” Operator. Output a number in floating point format. The user may specify the total number of character spaces for the print field, as well as the number of digits to be output to the right of the decimal point.

When used in an image string, the G operator takes the form “n.mG”, where n is the total number of spaces reserved for a number in the output field, and m is the number of digits to appear to the right of the decimal point. The decimal point takes up one space in the print field.

If a number does not fit within its designated space, a string of *’s is output.

N can have a maximum value of 31.

Example:

```
1050 Print using "4.2g":1.41421
```

This statement produces the following output on the system console device:

```
1.41
```

The “H” Operator. Output the hexadecimal representation of a number.

A field of print spaces may be reserved for the data element by means of the “n” modifier. This field must be large enough to accommodate both a sign (if the number is negative) and the digits that make up the number.

The output range for PRINT USING with the “H” operator is from $-(2^{32}-1)$ to $+(2^{32}-1)$.

When a specific length is allotted to a print-list element being printed with the “H” operator, the allotted field is left-zero filled.

Example:

```
*x=14
*print using "8h":x
0000000E
*x=-14
*print using "8b":x
-0000000E
```

The “I” Operator. Output data in binary item format.

The “I” operator transfers data to a device using the 4041’s internal data representation. This operator can be used to store data on mass-storage devices, such as tapes or disks, in order to speed subsequent data retrieval (because incoming data will not have to be converted to the 4041’s internal format; they’ll already be in it).

The “I” operator cannot be used with the “n” modifier; it must be used by itself or within a repetition count (denoted by parentheses) only.

The “J” Operator. Output data item in “engineering” notation.

The “J” operator functions in the same way as the “E” operator, except:

- The minimum field size that can be specified with the “J” operator is 4.
- If the number’s exponent is between -12 and +3, a letter indicating the exponent’s value is printed. The letters printed for various exponents are as follows:

| Exponent Value | Letter |
|----------------|--------|
| -12 | P |
| -9 | N |
| -6 | U |
| -3 | M |
| +3 | K |

Example:

```
2000 Print using "4j":15000
```

This statement produces the following output:

```
<sp>15K
```

PRINT

The “K” Operator. Output numeric data item in shortest form.

This operator causes the 4041 to find the representation of a data item that takes the fewest print spaces while retaining all the significant digits of a number. The 4041 then outputs the data item in that representation.

NOTE

For purposes of finding the shortest form of a data item, the 4041 uses a “short-form” scientific notation. The exponent is output as an integer without left-zero fill; the exponent sign is output only if the exponent is negative. Likewise, the sign of the characteristic is output only if the characteristic is negative.

Example:

```
1250 Num=1000000
1260 Print using "k":num
```

This example produces the following output:

```
1E6
```

The integer representation of variable NUM takes up seven print spaces; the floating point representation takes up eight. The scientific representation takes up only three print spaces, so the 4041 prints the data item in that form.

The “L” Operator. Output n ASCII line feed characters.

The “L” operator outputs a line feed. The “L” operator does not print a carriage return along with the line feed; instead, the “carriage” is moved down one line, remaining in the same position relative to either margin.

Example:

```
10 Print using 20:"FE","FI","FO","FUM"
20 Image 2a,1,2a,1,2a,1,3a
```

These statements produce the following output:

```
FE
FI
FO
FUM
```

Each “A” operator sets up an alphanumeric field of appropriate length, while each “L” operator moves the print line down by one line without returning the carriage to the first column. Note the difference between the “L” operator and the “/” operator, described next.

The “/” Operator. Output n end-of-message characters.

The “/” operator is similar to the “L” operator, except it causes printing to start in the first column of a new line.

Example:

```
10 Print using 20:"FE","FI","FO","FUM"
20 Image 2a,/,2a,/,2a,/ ,3a
```

These commands produce the following output (with the default EOM character (CR-LF):

```
FE
FI
FO
FUM
```

The “A” operators create alphanumeric fields of appropriate size for each data item, while the “/” operators cause each data item to be printed on the line below the previous one, starting in the first column.

NOTE

The “/” operator will not cause a line feed to be printed if the EOM character has been changed to (CR). In this case, the four lines shown above would be printed one over the other, with predictable results (an unreadable message, if output was to a printer or DVST terminal; the word “FUM”, if output was to a raster-scan terminal).

The “N” Operator. Output string or numeric data item in shortest format.

The “N” operator functions similarly to the “K” operator, except that “N” can output strings as well as numbers.

The “N” operator follows the same rules for determining the shortest form of a number as the “K” operator.

Strings output in their shortest form have trailing nulls deleted.

Example:

```
1400 Print using 1500: number
1450 Print using 1500: string$
1500 Image n
```

Because “N” is an all-purpose, shortest-format operator, the same IMAGE statement can handle both PRINT commands in lines 1400 and 1450.

The “O” Operator. Output the octal representation of a number.

A field of print spaces may be reserved for the data element by means of the “n” modifier. This field must be large enough to accommodate both a sign (if the number is negative) and the digits that make up the number.

The output range for PRINT USING with the “O” operator is from $-(2^{32}-1)$ to $+(2^{32}-1)$.

When a specific length is allotted to a print-list element being printed with the “O” operator, the allotted field is left-zero filled.

Example:

```
*x=12
*print using "8o":x
00000014
*x=-12
*print using "8o":x
-0000014
```

The “S” Operator. Suppress output of EOM character.

This operator suppresses the EOM character normally output after a PRINT statement is executed.

Example:

```
160 Num=450
170 Num2=125
180 Print using "4d,s":num
190 Print using "4d":num2
*run
```

These statements produce the following output:

```
<sp>450<sp>125
```

Line 180 reserves four print spaces for the value of the variable Num, then prints that value right-justified within the four-space field. The “S” operator then suppresses printing of the carriage-return/line-feed that normally follows execution of a PRINT statement. Thus, when statement 190 is executed, the next data item to be printed appears on the same line as the previous data item.

The “T” Operator. Output SPACES until character position n reached.

The “T” operator moves the print cursor to a specified character position on the print line. The specification “30T”, for example, tells the 4041 to output spaces until the cursor reaches the 30th print space on the current print line.

The “T” operator cannot force the print cursor to move backwards over a print line. Thus, if the cursor is currently positioned in column 20, and a “15T” is encountered in an image string, no action is taken, because the cursor cannot move back to position 15 from position 20 on the current line.

“T” differs from the “X” operator in that “T” specifies the next column that printing will start in, while “X” specifies a number of spaces to be printed from the current cursor position.

Example:

```
200 Able=500
210 Baker=38.875
220 Charli=2.3196E10
230 Print using abc:able, baker, charli
240 Abc: image 4.0g,11t,6.3g,21t,2e
```

These commands produce the following output:

```
500<sp><sp><sp><sp><sp><sp><sp><sp>
<sp>38.875<sp><sp><sp><sp><sp>2.E10
```

The IMAGE statement labeled “ABC” in line 240 tells the 4041 to:

1. Print a numeric data item (Able) in four print spaces, with no digits to the right of the decimal point.
2. Print spaces until the print cursor is positioned in the 11th column of the current print line.
3. Print another numeric data item (Baker) in six print spaces, with three digits to the right of the decimal point.
4. Print spaces until the print cursor is positioned in the 21st column of the current print line.
5. Print a numeric data item (Charli) in scientific notation, with two digits in the characteristic.

PRINT

The “X” Operator. Output n spaces from current cursor position.

The “X” operator outputs n spaces from the current cursor position. It differs from the “T” operator in that “X” does not specify the column in which printing will resume; it simply causes the 4041 to count off n spaces from the current column and prints the next item from there.

Because the “X” operator does not specify which column data items will appear in, the “T” operator is most often used when data are to appear in columns, or in other cases where output formatting is important. The “X” operator, on the other hand, is more convenient to use when neat columnar output formatting is not needed.

Example:

```
200 Able=500
210 Baker=38.875
220 Charli=2.3196E10
230 Print using abc: able, baker, char
240 Abc: image 4.0g,5x,6.3g,5x,2e
```

These statements produce the following output:

```
< sp> 500.< sp> < sp> < sp> < sp> < sp> 38.875<
sp> < sp> < sp> < sp> < sp> 2.E + 10
```

The “X” operators in line 240 cause five spaces to be printed between the last character of variable “Able” and the first character of variable “Baker”, and the last character of variable “Baker” and the first character of variable “Charli”.

The “'” Operator. Begin/end imbedded string.

Strings enclosed in single quotes are reproduced in output.

Example:

```
100 Freq=1000
110 Print using "'FREQ ',4d":freq
run
(output) FREQ 1000
```

```
100 Freq$="FREQ "
110 Freq=1E3
120 Ampl$="AMPL "
130 Ampl=1
140 Image$="n,n;n,n"
150 Print using image$:freq$,freq,ampl$,ampl
run
(output) FREQ 1000.0;AMPL 1.0
```

Example 8-9.

The “:” Operator Output a colon.

Example:

```
100 Nrpt$="NR.PT"
110 Nrpt=512
120 Print using "5a:3d":nrpt$,nrpt
run
(output) NR.PT:512
```

The “;” Operator. Output a semicolon (see Example 8-9).

The “%” Operator (GPIB). Output data item in binary block argument format.

Binary block argument format is a data transfer format defined by the Tektronix GPIB Codes and Formats standard to facilitate rapid data transfers between devices on the GPIB.

When information is exchanged using binary block argument format, the talking device sends the following information over the GPIB after ATN becomes unasserted:

1. A percent sign (%).
2. A two-byte binary count, telling the receiving devices how many data bytes will be sent in the upcoming data transfer.
3. A sequence of bytes representing the data.
4. A checksum byte, equal to the two’s complement of the modulo-256 sum of all preceding data bytes sent since the first percent sign.

The “%” operator can be used to transfer one or more integer scalars or integer arrays from the 4041 over the GPIB. Only integer scalars or integer arrays can be transferred using the “%” operator.

In addition, the "n" modifier can be used to control the way these integers are sent. The way this is done is explained below.

The 4041 stores integers in 2 bytes of memory (16 bits), in 2's complement representation. Thus, numbers from -32768 to +32767 can be stored in a 4041 integer.

However, the GPIB has only eight data lines. Therefore, transmitting the entire contents of a 4041 integer over the GPIB requires that two GPIB data bytes be transmitted. By convention, the 4041 integer's most significant byte is transmitted first.

The "n" modifier, in conjunction with the "%" operator, tells the 4041 how many bits of each integer to put onto the bus. The default value for "n" is 16; thus, using the default value causes the 4041 to transmit each integer in the print list as a sequence of two GPIB data bytes, most significant byte first.

If the value of the "n" modifier is less than 16 but greater than 8, the 4041 takes the n rightmost bits of each integer in the print list, and makes them the n leftmost bits of an integer to be transmitted over the GPIB, most significant byte first.

Example:

```
100 Open #1:"gpib(pri=1):"  
110 Print #1 using "12%":16706
```

These lines produce the following results upon execution:

1. The 4041 sends the UNLISTEN interface message over the GPIB (to clear any stray listeners off the bus).
2. The 4041 sends its talk address and the listen address of the device at primary address 1 over the GPIB.
3. The 4041 takes the twelve rightmost bits of the integer 16706 (HEX 4142), and makes them the twelve leftmost bits of a new integer, 5152 (HEX 1420).
4. The 4041 then sends the following values over the bus:

```
HEX 25 — percent sign  
HEX 0 } these two bytes are  
HEX 3 } the binary byte count  
HEX 14 } these two bytes are  
HEX 20 } the data  
HEX C9 — this is the checksum byte, sent with  
EOI asserted to indicate the end of the  
message.
```

5. The 4041 then sends the UNTALK and UNLISTEN interface messages.

If the "n" modifier is less than or equal to 8, the 4041 sends the n rightmost bits of each integer in the print list as the n leftmost bits of a single GPIB data byte. (Note that when n > 8, the 4041 sends two GPIB data bytes for each integer; when n <= 8, it only sends one.)

If the print list contains an integer array element, the entire integer array is transmitted using the format described above, i.e., a percent sign is transmitted, followed by two bytes signifying the number of bytes to be transferred, followed by the array values, followed by a checksum sent with EOI asserted.

To transmit values from a print list more than one element long, use the "n" modifier with parentheses around the "%" operator to indicate the number of print elements to be transmitted using binary block argument format.

Example:

```
100 Open #1:"gpib(pri=1):"  
110 Print #1 using "2(8%)" :a,b
```

This statement tells the 4041 to transfer two integer data elements A and B to the device at primary address 1 on the standard GPIB interface port, using binary block argument format. The rightmost 8 bits of each integer will be transferred as a single GPIB data byte.

When the print list contains more than one element, the 4041 treats the binary block transmission of each element like a separate PRINT statement, sending talk and listen addresses, binary byte count, data, and checksum for each print list element.

The EOI line is only asserted after the last print list element has been transferred.

The 4041 does not send listen addresses when the logical unit to which it is transferring data is open to no primary address (as, for example, with the statement OPEN #1:"GPIB0:"). In this case, the 4041 simply puts values onto the bus for receipt by any previously listen-addressed devices.

No other PRINT USING operator may be used with the "%" operator. If the 4041 is to send ASCII header information preceding a block of data, the header must be sent via a separate PRINT statement.

INPUT/OUTPUT
PRINT

The “@” Operator (GPIB). Output data item in end block argument format.

End block argument format is a data transfer format defined by the Tektronix GPIB Codes and Formats standard to facilitate rapid data transfers between devices on the GPIB.

When information is exchanged using end block argument format, the talking device sends the following information over the GPIB after ATN becomes unasserted:

1. An “at” sign (@).
2. A sequence of bytes representing the data. The last data byte is sent with EOI asserted, to indicate the end of the data transfer.

In all other respects, the “@” PRINT USING operator is identical to the “%” PRINT USING operator.

The “F” Modifier. Set up a print field just large enough to accommodate all significant digits of a numeric data item, or current length of a string data item.

Example:

```
540 SqrTwo=1.414
550 TwoSqr=4
560 Print using 570:sqrtwo,twosqr
570 Image 2(fg)
```

This program segment produces the following output:

```
1.4144
```

A print field of five spaces is set up to print the value of SqrTwo, then a one-space print field is set up to print the value of TwoSqr.

The “n” Modifier. Designates the number of character positions in a field, or the number of times a parenthesized image string segment is to be repeated.

When used to designate the number of times a parenthesized image string is to be repeated, n must be an integer <= 511.

Example:

```
350 Image 10a
```

Sets up a 10-character alphanumeric print field.

```
400 Image 5(6d)
```

Sets up print fields of six spaces for each of five integers.

The “R” Modifier. Use a comma for a radix symbol instead of a decimal point. Used with the “E” and “G” operators only.

Example:

```
(output) 100 Print using "r5.2g":23.4567
run
23,46
```

The “Z” Modifier. Set the leading fill character to “0”. Used with the “D” and “G” operators only.

Example:

```
(output) 100 Print using "z10.2g":123.456
run
0000123.46
```

The “*” Modifier. Set the leading fill character to “*”. Used with integer and floating point formats only.

Example:

```
(output) 100 Print using "**10.2g":123.456
run
****123.46
```

The “+” Modifier. Print a plus sign before the numeric value in a print field if the number is positive, and a minus sign if the number is negative. Used with the “D”, “E”, and “G” operators only.

Example:

```
(output) 100 Print using "+5.2g":123.456
run
+123.
```

The “-” Modifier. Print a space before the numeric value in a print field if the number is positive, and a minus sign if the number is negative. This is the default modifier for the “D”, “E”, and “G” operators. Used with the “D”, “E”, and “G” operators only.

The “<” Modifier (default). Left-justify string data within a field. Used with the “A” operator only.

Example:

```
(output) 100 Print using "<9a":"WATER"
run
WATER<sp><sp><sp><sp>
```

The “>” Modifier. Right-justify string data within a field. Used with the “A” operator only.

Example:

```
(output) 100 Print using ">9a":"WATER"
run
<sp><sp><sp><sp>WATER
```

The “=” Modifier. Center string data within a field. Used with the “A” operator only.

Example:

```
(output) 100 Print using "=9a":"WATER"
run
<sp><sp>WATER<sp><sp>
```

The PUTMEM Statement

| | |
|-------------------------|--|
| Syntax Form | <code>[line-no.] PUTMEM BUFFER strvar [clause list]: {numexp} [,numexp ...] {strex} [,strex ...]</code> |
| where | <code>clause-list = [#numexp] OR [#strex] AND [USING numexp] OR [USING strexp]</code> |
| Descriptive Form | <code>[line-no.] PUTMEM BUFFER target-var[clauses]:data[,data ...]</code> |

PURPOSE

The PUTMEM statement transfers numeric or string data into a string variable.

EXPLANATION

PUTMEM does the exact opposite of GETMEM; it takes numeric or string variables from a list and stores them in a string.

The string in which PUTMEM stores the result of its operation must be either a simple string variable or a string array element; it cannot be a string array.

The PUTMEM statement can use the same USING clause operators and modifiers as the PRINT statement. In addition, the PUTMEM statement allows the "%", "@", and "I" operators to be used in the same image string as other operators (unlike the PRINT statement).

Semicolons and commas can be used to separate elements of a PUTMEM list.

Commas separate message units in a PUTMEM list. A comma between two elements of a PUTMEM list stores the End-of-Message-Unit (EOU) character in the result string.

Commas in a PUTMEM list must have data elements to either side; a comma starting or ending a PUTMEM list or two or more commas in a row within a PUTMEM list cause an error.

The default EOU character for the GPIB is a COMMA; the default EOU character for other devices is a TAB.

Semicolons separate elements within a message unit. The first semicolon in the message unit stores the End-of-Header (EOH) character in the result string, and the second and succeeding semicolons in the message unit store the End-of-Argument (EOA) character in the result string.

EOH defaults to SPACE for the GPIB driver, and defaults to null for all other devices. EOA defaults to COMMA for the GPIB driver, and defaults to null for all other devices.

EXCEPTIONS:

1. If a semicolon separates two elements of a PUTMEM list after a comma has appeared in the same list, the first semicolon after the comma stores the EOH character in the result string, the second and succeeding semicolons store EOA.
2. A semicolon concluding a PUTMEM list has no effect.

EXAMPLE

```
5000 Num1=2
5010 Num2=4
5020 Num3=6
5030 Putmem buffer a$:num1,num2,num3
```

Line 5030 stores the string "2<tab>4<tab>6" in string variable A\$ (assuming that the system console device is the COMM or FRTP).

Using the same values for Num1, Num2, and Num3, the command

```
5030 Putmem #"gpiB:" buffer a$:num1;num2;num3
```

stores the string "2<sp>4,6" in string variable(A\$ (the GPIB driver's default EOH and EOA characters are used to output the characters of the message unit).

The RBYTE Statement

Syntax Form (non-record-read format)

```
[line-no.] RBYTE [#numexp:] {numvar[,numvar]...}  
                [#strexp:] {strvar[,numvar][,strvar[,numvar]] ... }
```

Descriptive Form (non-record-read format)

```
[line-no.] RBYTE [#lunum:]      {numeric-var[,numeric-var] . . . }  
                [#stream-spec:] {string-var[,state-of-EOI][,string-var [state-of-EOI]] . . . }
```

Syntax Form (record-read-format)

```
[line-no.] RBYTE[#numexp:] numexp,strvar  
                [#strexp:]
```

Descriptive Form (record-read-format)

```
[line-no.] RBYTE [#numexp:] physical-record-to-transfer, string-  
                [#strexp:]      variable-in-which-to-store-contents
```

PURPOSE

In non-record-read format, the RBYTE statement transfers "literal" 8-bit bytes into the 4041's memory.

In record-read format, the RBYTE statement transfers entire physical records into the 4041's memory.

String data items in the RBYTE list are delimited by the EOI line's becoming asserted (for GPIB data transfers) or by the transfer of a number of characters equal to the dimensioned length of the string variable. If the string variable is undimensioned when the RBYTE statement is executed, the variable is assigned a default dimensioned length of one byte.

EXPLANATION

Non-record-read Format

Any number of string or numeric data items may be transferred using a single RBYTE statement.

When transferring a string data item via RBYTE, a numeric variable immediately following a string variable in the RBYTE list will store the state of the GPIB's EOI control line at the end of the data transfer (1 = EOI line asserted; 0 = EOI line unasserted).

Record-read Format

In record-read format, the 4041 reads an entire physical record from a driver into a string variable.

The numeric expression in record-read format specifies the number of the physical record to be read (e.g., from a DC-100 tape).

Data read from the physical record is stored in the string variable specified in the RBYTE list. If the string variable has not been dimensioned previously, it is dimensioned to a length of 256 characters by the RBYTE statement.

NOTE

If the data path for the RBYTE operation does not use a GPIB driver, the value of a numeric variable immediately following a string variable in an RBYTE list will always be 0.

NOTE

Record-read format is intended for use with the TAPE driver. If an RBYTE statement conforming to record-read format is executed on other than the TAPE driver, the numeric expression denoting the record number is ignored, and the 4041 inputs a string variable from whatever device is called for by the RBYTE statement.

SPECIFYING DATA PATHS FOR RBYTE

The 4041 uses the currently SELECTed stream spec as the default data path for RBYTE. The 4041 powers up with a selected stream spec of "GPIB0:".

If all RBYTE and WBYTE commands are to be executed using a stream spec other than "GPIB0:", a SELECT statement selecting another stream spec should be executed.

If the user wants the standard GPIB interface port to remain as the selected stream spec, but wishes to execute one or more RBYTE or WBYTE commands using a different stream spec, a "#" clause should be included in every RBYTE or WBYTE statement so executed.

The ASK\$(SELECT) function may be used to determine the currently SELECTed stream spec. (For more information, see the description of this function in Section 5, *Environmental Control*.)

COMM

When transferring numeric data values via the COMM driver, RBYTE stores the ASCII decimal equivalent of each key into a numeric variable in the RBYTE list.

Example

```

100 Rbyte #"comm":x,y,z
110 Print x,y,z
run
(user input) 123
(output) 49.0<tab>50.0<tab>51.0

```

When transferring strings via the COMM driver, RBYTE delimits each string when the number of characters received equals the number of characters to which each string is dimensioned.

Example

```

100 Dim a$ to 2,b$ to 2,c$ to 2
110 Rbyte #"comm":a$,b$,c$
120 Print a$,b$,c$
run
(user input) ABCDEF
(output) AB<tab>CD<tab>EF

```

FRTTP

Using the FRTTP to transfer data bytes via RBYTE is similar to using the COMM driver, except that bytes representing system function keys may be transferred as well as bytes representing numbers or characters.

The system function keys and the values they transfer are as follows:

| Function Key | Value |
|--------------|-------------|
| 160 | <CURSOR |
| 161 | CURSOR> |
| 162 | SCROLL< |
| 163 | SCROLL> |
| 164 | DELETE< |
| 165 | DELETE> |
| 166 | CLEAR |
| 167 | INSERT |
| 168 | STEP |
| 169 | CONTINUE |
| 170 | AUTO-LOAD |
| 173 | RUN |
| 174 | LIST |
| 175 | AUTO-NUMBER |
| 176 | RECALL |
| 177 | RECALL PREV |
| 178 | RECALL NEXT |
| 179 | DELETE |

INPUT/OUTPUT
RBYTE

GPIB

Use of the RBYTE statement with the GPIB driver is described in Section 9, *Instrument Control with GPIB*, under "Low Level Data Transfers."

PRIN

The RBYTE statement cannot be used with the PRIN driver.

Tape

RBYTEs from the DC-100 tape drive read a string from a specified physical tape record.

To read data from the tape using RBYTE, the tape's PHYSICAL parameter must be set to a value of "YES". In addition, ALL TAPE FILES MUST BE CLOSED AT THE TIME OF THE RBYTE OPERATION.

In addition, the RBYTE statement must include a stream spec or logical unit number denoting that the TAPE driver is to be used, or the TAPE driver must be selected by means of the SELECT statement.

The numeric expression must evaluate to an integer greater than 1 and less than the number of physical records on the tape. This integer is the physical record of the tape that will be read from.

(To find the number of physical records on the tape: execute a DIR command; add up the total number of bytes on the tape; divide by 255; add 4.)

The string variable should be dimensioned to a length of 256 characters. If the string variable is dimensioned to less than 256 characters, characters after the current dimensioned size of the string are lost.

Example.

```
100 Dim string$ to 256
110 Close all !close all tape files
120 Select "tape(phy=1):"
130 Rbyte 5,string$
```

Line 130 reads the contents of physical record #5 from the DC-100 tape, and stores the contents in string variable String\$.

DISK

RBYTEs from the disk drive read a string from a specified sector.

To read data from the disk using RBYTE, the disk's PHYSICAL parameter must be set to a value of "YES".

In addition, the RBYTE statement must include a stream spec or logical unit number denoting that the disk driver to be used, or the disk driver must be selected by means of the SELECT statement.

The numeric expression must evaluate to an integer greater than 1 and less than the number of physical sectors on the disk. This integer is the physical sector of the disk that will be read from.

To find the number of physical sectors on the disk: execute a DIR command; note last start sector/rec #, then add used and un-used length of that file plus the amount free in last block. Then, divide by 512 and add the last start sector/rec #.

Example:

```
2720 (last Start Sector/Rec #)
51200 (Used Length)
1933312 (Un-used Length)
7268352 (Amount Free)
```

$$9252864 / 512 = 18072 + 2720 = 20792 \text{ (physical sectors)}$$

The string variable should be dimensioned to a length of 512 characters. If the string variable is dimensioned to less than 512 characters, characters after the current dimensioned size of the string are written as zero's (NULL).

Example:

```
100 Dim string$ to 512
110 Select "disk (phy=YES):"
120 Rbyte 5, string$
```

Line 120 reads the contents of sector #5 from the disk, and stores the contents in string variable String\$.

PROCEED MODE RBYTE

The RBYTE statement may be used when the 4041 is in proceed mode. In proceed-mode, an RBYTE statement can only input a single string variable per statement.

The READ Statement

Syntax and Descriptive Forms [line-no.] READ var[,var...]

PURPOSE

The READ statement reads values into memory from DATA statements, starting at the current position of the DATA pointer within the current program block.

EXPLANATION

When a program is executed, values contained in DATA statements are "converted" into an internal "file" within each program block.

At the start of execution, a "data pointer" is set up for each block, pointing to the first value of this internal file, which corresponds to the first value contained in the lowest-numbered DATA statement in the program block.

Whenever a value is read from this internal data file by means of the READ statement, the data pointer is advanced to the next value in the file.

If an array variable is specified in the READ statement, values are read from DATA statements until the entire array is filled.

The data pointer can be repositioned to the first item in the first DATA statement in the block, or to the first item in any given DATA statement in the block, by means of the RESTORE statement. (See the description of the RESTORE statement, later in this section, for more information.)

Attempting to read data items past the last DATA statement in the current program block results in an error.

EXAMPLES

```
1000 Dim aardvark (10)
1010 Read aardvark, hello$
1020 Data 1,2,3,4,5,6,7,8,9,10,"Hello"
1030 Data 11,12,13,14,15
```

Line 1000 reserves storage for a 10-element numeric array called Aardvark.

Line 1010 reads 10 values from DATA statement 1020. If the data statement did not contain sufficient data to fill the array, the contents of other DATA statements in the current program block would be read into Aardvark until the array was filled.

```
2000 DATA "Hello", "LAST", 1,2,3
2010 READ Hello$, Last $, a,b,c
```

Line 2010 reads the two string in the data statement into the two string variables. The three numeric data values are then read into the numeric variables.

If a numeric data value is found when a string is expected or the reverse situation happens, then an error will occur.

The RESTORE Statement

| | |
|-------------------------|-------------------------------------|
| Syntax Form | [line-no.] RESTORE [numexp] |
| Descriptive Form | [line-no.] RESTORE [line-reference] |

PURPOSE

The RESTORE statement sets the DATA pointer to the first item in the first (or alternately, any specified) DATA statement in the current program block.

EXPLANATION

When a program is executed, all values contained in DATA statements are put into an internal "file" for each program block. Values are assigned to variables from this data file by means of the READ statement.

Whenever a value is read from the file, a data pointer is updated to point to the next value in the file. The RESTORE statement is used to initialize this data pointer to the beginning of any specified DATA statement in the current program block.

When the RESTORE keyword is not followed by a line reference, the data pointer is moved to the first data value in the lowest-numbered DATA statement in the current program block.

When the RESTORE keyword is followed by a line reference, the data pointer is moved to the first value in the specified DATA statement. The line referred to must be in the current program block, or an error results.

EXAMPLES

```
10000 Restore 10050
```

This statement moves the data pointer to the first value contained in DATA statement 10050. This value will be the next one to be read by a READ statement in the current program block. Line 10050 must be in the current program block or an error occurs.

The SELECT Statement

| | |
|-------------------------|-------------------------------|
| Syntax Form | [line-no.] SELECT strexp |
| Descriptive Form | [line-no.] SELECT stream-spec |

PURPOSE

The SELECT statement defines the default stream spec for RBYTE, WBYTE, and POLL statements.

EXPLANATION

The stream spec "GPIB0:" becomes the SELECTed stream spec when the first RBYTE or WBYTE statement not including a logical unit or stream spec is executed, unless a SELECT statement has been executed previously.

The 4041 powers up with the stream spec "GPIB:" selected by default.

Any RBYTE or WBYTE statements not designating a stream spec for their operation (i.e., not containing a "#" clause) use the SELECTed stream spec.

Use of the SPE Parameter with GPIB Drivers

The GPIB driver's SPE parameter designates the amount of time that the 4041 will wait for a response from a GPIB device when the 4041 is executing a serial poll.

The POLL statement uses the SELECT stream spec to determine the value of the SPE parameter. Therefore, if an SPE value other than the default (10 milliseconds) is to be used, the SPE value must be specified by a SELECT statement.

EXAMPLES

```
3120 Select "gpib1:"
```

This statement selects the optional GPIB port as the default stream spec for RBYTE and WBYTE operations.

```
3200 Select "tape(phy=yes)"
```

This statement selects the DC-100 tape as the default stream spec for RBYTE and WBYTE operations.

The WBYTE Statement

Syntax and Descriptive Form (non-record-write format)

```
[(line-no.) WBYTE [#numexp:] WBYTE-element[,WBYTE-element] ...
                [#strexp:]
```

where WBYTE-element = {numexp (all drivers)
 {strexp (all drivers)
 {GPIB-function (GPIB drivers only)

Syntax Form (record-write format)

```
[(line-no.) WBYTE [#numexp:] numexp,strexp
                [#strexp:]
```

Descriptive Form (record-write format):

```
[(line-no.) WBYTE [#numexp:] physical-record-to-write,data
                [#strexp:]           to-write-to-physical-record
```

PURPOSE

In non-record-write format, the WBYTE statement transfers "literal" 8-bit bytes from the 4041's memory to a specified device.

In record-write format, the WBYTE statement transfers data from the 4041's memory into a specified physical record on an output device (such as the DC-100 tape).

EXPLANATION

Non-Record-Write Format

Any number of string or numeric data items may be transferred using a single WBYTE statement.

When a numeric data item is transferred via WBYTE, the numeric expression specifying the data item is rounded to an integer. This integer must be a number between -255 and +255, or an error is returned.

The 4041 then outputs a byte whose binary representation is equal to the absolute value of the integer (e.g., if the integer is 65, the 4041 outputs an ASCII "A"). If one of the GPIB drivers is being used for output, the 4041 asserts the EOI line as it transmits the bytes if the value of the integer is negative.

When a string data item is transferred via WBYTE, a numeric value immediately following the string expression in the WBYTE list denotes the state of the EOI line concurrent with the transmission of the last byte of the string. If no numeric value follows the string expression in the WBYTE list, the EOI line remains unasserted.

NOTE

If the WBYTE statement is not using one of the GPIB drivers, a numeric value following a string expression in the WBYTE list is still interpreted as denoting the state of the EOI line concurrent with the last byte of the string. Thus, a numeric value following a string expression in a WBYTE statement is effectively ignored.

The WBYTE statement ignores all EOM, EOU, EOH, and EOA processing.

Record-Write Format

In record-write format, the 4041 writes data onto a specified physical record of an output device.

The numeric expression in record-write format specifies the number of the physical record to be written to.

The string expression in record-write format specifies the data to be written onto the physical record.

NOTE

Record-write format is intended for use with the TAPE driver. If a WBYTE statement conforming to record-write format is executed on other than the TAPE driver, the numeric expression denoting the record number is ignored, and the 4041 outputs the string expression to the device called for by the WBYTE statement.

SPECIFYING DATA PATHS FOR WBYTE

The 4041 uses the currently SELECTed stream spec as the default data path for WBYTE. The 4041 powers up with a selected stream spec of "GPIB0".

If all RBYTE and WBYTE commands are to be executed using a stream spec other than "GPIB0:", a SELECT statement selecting another stream spec should be executed.

If the user wants the standard GPIB interface port to remain as the selected stream spec, but wishes to execute one or more RBYTE or WBYTE commands using a different stream spec, a "#" clause should be included in every RBYTE or WBYTE statement so executed.

The ASK\$("SELECT") function may be used to determine the currently SELECTed stream spec. (For more information, see the description of this function in Section 5, *Environmental Control*.)

COMM

When WBYTE is used to output numeric data values via a COMM port, the WBYTE statement outputs each value as an 8-bit byte. If the device attached to the COMM port receives ASCII code, each value sent in this way will cause the ASCII character equivalent of the value sent to be printed on the device.

When WBYTE is used to transfer string data, the WBYTE statement outputs a stream of values as a series of 8-bit bytes.

Example

(To be run with the standard RS-232-C interface port as system console.)

```

200 Num1=65
210 Num2=66
220 Wbyte #"comm":num1,num2
run
AB*
(output)

```

Note how the cursor remains to the right of the output. This is because the WBYTE statement does not output an end-of-message character (WBYTE ignores all EOM, EOU, EOH, and EOA processing).

F RTP

When WBYTE is used to output numeric values to the front panel, each numeric value is rounded to the nearest whole number, and the ASCII character equivalent of that number is displayed.

WBYTE can also be used to output a string to the front panel.

Successive elements of the WBYTE list over-write each other on the front panel. Therefore, it is best to WBYTE only one data element per WBYTE statement to the front panel.

WBYTE**GPIB**

Use of the WBYTE statement with the GPIB driver is described in Section 9, *Instrument Control With GPIB*, under "Low Level Data Transfers."

PRIN

WBYTEs to the thermal printer are executed in the same way as WBYTEs to the front panel, except that successive WBYTE elements appearing on successive lines instead of over-writing each other.

TAPE

WBYTEs to the DC-100 tape write a string onto a physical record of the tape. To send data to the tape using WBYTE, the tape's PHYSICAL parameter must be set to a value of "YES". In addition, ALL TAPE FILES MUST BE CLOSED AT THE TIME OF THE WBYTE OPERATION.

WBYTE commands to the DC-100 tape take the form

```
[line-no.] WBYTE [#lunum:] numexp,strexp
                [#stream
                -spec:]
```

The numeric expression must evaluate to an integer greater than 1 and less than the number of physical records on the tape. (To find the number of physical records on the tape: execute a DIR command; add up the total number of bytes on the tape; divide by 255; add 4.) This integer is the physical record of the tape that will be written on.

If the string expression evaluates to a string with length less than 256, then the record is written with (binary) zero fill for bytes after the last byte of the string. If the string expression evaluates to a string with length greater than or equal to 256, only the first 256 bytes are written.

WARNING

The following example writes over physical record 10 of a DC-100 tape. Don't run the example unless you have no need for the data that may already be stored there.

Example

```
400 Num=10
410 Dim string$ to 256
420 String$="ABCDEFGHIJKLMNPOQRSTUVWXYZ"
.
500 Close all
510 Open #1:"tape(phy=1):"
520 Wbyte #1:num,string$
```

Line 520 writes the string String\$ onto physical record 10 of the current DC-100 tape. Since String\$ is less than 256 characters long, the record is filled with binary zeros (ASCII null characters) after the string is written out.

DISK

WBYTEs to the disk write a string onto a sector of the disk. To send data to the disk using WBYTE, the disk's PHYSICAL parameter must be set to a value of "YES".

WBYTE commands to the disk take the form

```
[line-no.]WBYTE[#lunum:]numexp,strexp
                [#stream
                -spec:]
```

The numeric expression must evaluate to an integer greater than 1 and less than the number of physical sectors on the disk. This integer is the physical sector of the disk that will be written on. To find the number of physical sectors on the disk: execute a DIR command; note last start sector/rec #, then add used and un-used length of that file plus the amount free in last block. Then, divide by 512 and add the last start sector/rec #.

Example:

```
2720 (last Start Sector/Rec #)
51200 (Used Length)
1933312 (Un-used Length)
7268352 (Amount Free)
```

$$9252864 / 512 = 18072 + 2720 = 20792 \text{ (physical sectors)}$$

The string variable should be dimensioned to a length of 512 characters. If the string variable is dimensioned to less than 512 characters, characters after the current dimensioned size of the string are lost.

Example:

```
400 Num=10
410 Dim string$ to 512
420 String$="ABCDEFGHIJKLMNPO
QRSTUVWXYZ"
.
500 Open #1:"disk(phy=1):"
510 Sbyte #1:num,string$
```

Line 510 writes the string String\$ onto a physical sector 10 of the current disk. Since String\$ is less than 512 characters long, the sector is filled with binary zeros (ASCII null characters) after the string is written out.

PROCEED MODE WBYTE

The WBYTE statement may be used when the 4041 is in proceed mode. In proceed-mode, a WBYTE statement may only output a single string expression per statement.

Section 9

INSTRUMENT CONTROL WITH GPIB

INTRODUCTION

This section is intended as a guide to using the 4041 with the General Purpose Interface Bus (GPIB). It assumes that the reader is familiar with such GPIB terminology as "controller", "talker", "listener", "primary address", etc. If you are not familiar with these terms, read Appendix E, *Introduction to GPIB Concepts*, before reading the remainder of this section.

Section 9 discusses the applications, with regard to GPIB instrument control of several statements that are discussed elsewhere in this manual. The statements and the sections in which they are discussed are: SET DRIVER (Section 5, *Environmental Control*) and OPEN, INPUT, PRINT, RBYTE, and WBYTE (Section 8, *Input/ Output*).

SETTING THE GPIB'S PHYSICAL DRIVER PARAMETERS

Before any commands or data are transferred over the GPIB, a SET DRIVER statement should be executed. For a complete description of SET DRIVER, see Section 5, *Environmental Control*.

The SET DRIVER statement is executed to guarantee that the 4041 is operating in the correct system controller mode (i.e., to specify explicitly whether or not the 4041 is the system controller), to specify the 4041's address on the bus, and to set other physical parameters (see the following).

The SET DRIVER Statement (GPIB)

| | |
|--------------------------|-----------------------------------|
| Syntax Form: | [line-no.] SET DRIVER streqp |
| Descriptive Form: | [line-no.] SET DRIVER stream-spec |

PURPOSE

To establish GPIB driver parameter settings that affect all logical data transfers through a GPIB driver.

EXPLANATION

The SET DRIVER statement only affects physical driver parameters. Attempting to set logical GPIB parameters with the SET DRIVER statement has no effect.

The physical driver parameters that can be set for the GPIB driver are:

- SC Determines whether or not the 4041 is the system controller (default: YES).
- MA The interface bus address (default: 30).
- PNS Value to send when polled with nothing to say (default: 0).
- IST Instrument status when a parallel poll is executed (default: FALSE).
- DEL Interface chip T1 delay. Legal values are NORMAL or FAST (default: NORMAL).

If a parameter is not specified in the stream spec included in the SET DRIVER statement, its current setting is not disturbed.

Defaults for SET DRIVER

If no SET DRIVER is explicitly executed within the program, the 4041 executes the following SET DRIVER statement as soon as a program statement calls for GPIB activity:

```
Set driver "gpibx(sc=yes,ma=30,pns=0,ist=0,del=normal):"
```

where "x" equals either 0 or 1, depending on whether the standard or the optional GPIB interface is being called for.

The SC Parameter

When the SC parameter is set to YES in a SET DRIVER statement, the 4041 immediately:

1. Pulses the IFC line for approximately 300 microseconds then waits for approximately 700 microseconds.
2. Asserts the REN line.
3. Becomes controller-in-charge (CIC = 5).
4. Clears all pending interrupts.

When the SC parameter is set to NO in a SET DRIVER statement, the 4041 de-asserts the REN line, and "loses" both system controller and controller-in-charge status.

NOTE

There is no formal mechanism for "passing" system controller status from one controller to another. However, when the 4041 is not system controller, any other controller is free to become the system controller.

Note also that the TCT function, used with the WBYTE command, can be used to pass controller-in-charge status to another controller on the bus.

Because the 4041 loses controller-in-charge status when it executes a SET DRIVER statement with SC = NO, INPUT or PRINT statements using logical units involving the GPIB interface port specified in the SET DRIVER statement will not work.

Example.

```
100 Set driver "gpib(sc=yes):"
110 Open #1:"gpib(pri=1,tim=1):"
120 Set driver "gpib(sc=no):"
130 Print #1:"hi" ! THIS MAY NOT WORK!
```

When the 4041 gives up system controller status in line 120, it also gives up controller-in-charge status. Therefore, it is no longer controlling the ATN line, and cannot put data onto the bus (as required by line 130) until it recognizes its talk address being sent by the new controller-in-charge. If the 4041 does not receive its talk address within one second (the time specified by the TIM parameter in line 110), error 811 is generated.

The MA Parameter

The MA parameter specifies the 4041's address on the bus. Legal values are 0 to 30 or 255(default:30). A value of 255 requests that the address remain unchanged.

The PNS Parameter

The PNS parameter specifies the value of the status byte, which is the value the 4041 returns when serially polled by another device acting as controller-in-charge. The default value for PNS is 0.

The PNS value is a bit-encoded value. Bit 7 (counting from the right, starting with 1), also known as the "rsv" bit, indicates that the 4041 is asserting SRQ when set. Attempting to assign a PNS value with bit 7 already set causes an error. Legal values for the PNS value are therefore 0-63 and 128-191 or 255. A value of 255 requests that the PNS parameter remain unchanged.

When the 4041 is asserting SRQ, the rsv bit is set automatically. Thus, if the PNS parameter value is 1 and the 4041 asserts SRQ, the 4041 will return a status byte value of 65 when polled.

The IST Parameter

The IST parameter specifies the value returned by the 4041 when parallel polled by another device acting as controller-in-charge. Legal values are TRUe or FALse. The default is TRUe.

The DEL Parameter

The DEL parameter specifies the GPIB interface chip's T1-delay setting. It is primarily useful when transferring data over the optional GPIB interface port using Direct Memory Access with very fast acceptors on the bus. Legal values are NORmal or FASt. The default value is NORmal.

Example.

```
100 Set driver "gpib0(ma=28,pns=1):"
```

This command sets the 4041's primary address on the standard GPIB interface port to 28, and sets the "polled with nothing to say" value to 1.

SETTING UP LOGICAL UNITS

Most data transfers to or from a single instrument on the GPIB should be done by means of logical units, i.e., by assigning a logical unit number to a stream spec that describes a data path to a device, then specifying PRINT or INPUT to/from that logical unit.

A logical unit can also be associated with a stream spec defining a data path to the bus itself (no specific device). In this case, output to the logical unit is received by any device that is listen-addressed at the time.

The OPEN statement associates a logical unit number with a stream spec.

The ASK\$("LU") function is used to verify/inspect the stream spec that has been associated with a logical unit.

The OPEN Statement (GPIB)

Syntax Form: [line-no.] OPEN #numexp:strex[,strvar]

Descriptive Form: [line-no.] OPEN #logical-unit:stream-spec[,open-result]

PURPOSE

To associate a logical unit number with a stream spec defining a data path either to an instrument on the bus, or to the bus itself.

EXPLANATION

The OPEN statement is used with the GPIB driver to define a data path to a given instrument on the bus or to the bus itself, and to define specific logical parameters associated with that data path.

The OPEN statement only affects logical parameters. Attempting to set physical parameters with the OPEN statement has no effect.

The logical driver parameters that can be set for the GPIB drivers are:

PRI Primary address of an instrument on the bus
SEC Secondary address of an instrument on the bus
EOA End-of-argument character for default PRINT
EOH End-of-header character for default PRINT
EOM End-of-message driver processing
EOQ End-of-query character for PROMPT clause
EOU End-of-unit character for default PRINT
TIM Data transfer timeout
SPE Serial poll response timeout
TRA Data transfer mode

See the descriptions of the INPUT statement in Section 8, *Input/Output*, and later in this section, for more information on the EOA, EOH, EOM, EOQ, and EOU parameters.

The TIM parameter is described in the discussions of the INPUT and PRINT statements found later in this section.

The SPE parameter is described in the discussion of the POLL statement, later in this section.

The TRA parameter's legal values are NORmal, FAST, and DMA. DMA can only be used with the optional GPIB interface port. When TRA = FAST, the 4041 effectively operates in "non = proceed" mode, even if a SET PROCEED 1 statement has been executed. In addition, the 4041 does not check for timeout during data transfers when TRA = FAST.

See Appendix D, *Stream Specifications*, for complete information on other parameters.

The OPEN Result String

If an OPEN result is called for in the OPEN statement, its length is always set to 0 and its contents set to null. (This syntax is included so that the user can open either a DC-100 tape file or any other logical unit with the same OPEN statement.)

The ASK\$("LU") Function (GPIB)

Syntax Form: ASK\$("LU",numexp)

Descriptive Form: ASK\$("LU",logical-unit)

PURPOSE

To inspect/verify the current settings of a logical unit on the GPIB driver. ASK\$("LU") can also be used to capture the current settings for later program use.

EXPLANATION

When a GPIB stream spec is open as logical unit number LuRef, invoking the ASK\$("LU",LuRef) function returns a string with the following format:

```
GPIBa(MA = b,SC = c,CIC = d,TL = e,ENA = f,
PEN = g,PRI = h,SEC = i,EOA = j,EOH = k,
EOM = l,EOQ = m, EOU = n,TIM = o,SPE = p,
TRA = q,PNS = r,IST = s,DEL = t,TC = u,
SRQ = v):
```

where

- a = port number
- b = interface address on the bus
- c = system controller setting
- d = controller in charge state
- e = addressed state of the bus
- f = enabled interface interrupts
- g = pending interface interrupts
- h = logical unit primary address on the bus
- i = logical unit secondary address on the bus
- j = default PRINT end-of-argument character
- k = default PRINT end-of-header character
- l = end-of-message specifier

- m = default INPUT PROMPT end-of-query character
- n = default PRINT end-of-message-unit character
- o = data transfer timeout
- p = serial poll timeout
- q = logical unit data transfer mode
- r = talker/listener "polled with nothing to say" value
- s = talker/listener IST sense when parallel poll executed
- t = chip T1 delay setting
- u = SRQ message parameter setting
- v = TC parameter setting (synchronous or asynchronous)

The string returned can be saved in a string variable and used in a subsequent OPEN, COPY, LIST, etc.

Example.

```
200 Open #100:"gpib(pri=2,sec=10,tim=.01):"
210 Print #100:"set?"
220 Copy ask$("lu",100) to ask$("console")
```

Line 220 prints the stream spec associated with logical unit 100 on the system console device.

HIGH LEVEL DATA TRANSFERS

Most GPIB data transfers can be performed using the INPUT and PRINT statements, as described in Section 8,

Input/Output, and in the ensuing discussion.

The INPUT Statement (GPIB)

Syntax Form: [line-no.] INPUT clause-list:var[,var...]

where:

clause-list =

{#numexp} [ALTER strexp] [BUFFER strvar][DELN strexp][DELS strexp][USING {numexp}][INDEX {numexp}]
{#strex} [PROMPT strexp] {strex} {strex}

Descriptive Form: [line-no.] INPUT input clauses:input-list

PURPOSE

To transfer data from the bus into program variables.

EXPLANATION

If the 4041 is the controller-in-charge and PRI < > 31, the following bus traffic precedes the input:

| | |
|-----|---|
| ATN | SPD (if serial poll mode active) |
| ATN | UNListen (if one or more listeners are currently addressed) |
| ATN | My-Listen-Address (the MA value plus 32) |
| ATN | Other-Talk-Address (generated from the PRI and SEC logical unit parameters) |

If the 4041 is the controller-in-charge and PRI < > 31, the following bus traffic follows the input:

| | |
|-----|----------|
| ATN | UNTalk |
| ATN | UNListen |

If the 4041 is a talker/listener, it waits patiently to be listen-addressed before transferring any data over the bus. This must happen within TIM seconds, or an error is generated.

Each data byte must be received by the interface within TIM seconds of the previous data byte, or an error is generated.

The EOM parameter setting specifies the character that should terminate a message, in addition to the EOI line's becoming asserted. (The EOI line's becoming asserted ALWAYS terminates a message.) If EOM = <0>, then only the EOI line can terminate a message. If EOM = <255>, then receipt of a line-feed character terminates a message, and if a carriage-return precedes the linefeed, the carriage-return is discarded. Any other EOM parameter setting specifies the character that should terminate a message.

Default Logical Units

If an INPUT statement is executed that specifies a logical unit number in the closed interval 0 to 30 that has not been previously opened, then an implicit open is executed using the stream spec:

```
"Gpib0(pri="&str$(logical-unit)&",tim=4):"
```

This logical unit then remains open until a CLOSE statement that closes the logical unit is executed, or until an END statement is executed.

Use of PROMPT Clause/EOQ Character

When a PROMPT clause is used in an INPUT statement with a GPIB driver, the following bus traffic precedes the transmission of the PROMPT message over the GPIB data lines:

- ATN UNListen (if one or more listeners are currently addressed)
- ATN My-Talk-Address (the MA value plus 64)
- ATN Other-Listen-Address (the listen address of the device to which the 4041 is sending the prompt)

The PROMPT message is then transmitted, followed by the End-of-Query (EOQ) character. (Default: no output).

After the PROMPT message is transmitted, bus traffic preceding the input is as described earlier.

Example.s

```
1040 Dim setting$ to 300
1050 Input #15 prompt "SET?":setting$
```

Line 1040 dimensions a string variable Setting\$ to a length of 300 bytes.

Line 1050 sends the device at primary address 15 the message "SET?". If the device adheres to the Tektronix Codes and Formats standard for GPIB, this asks the device to transmit its settings when it becomes talk addressed.

After transmitting the message "SET?", the 4041 sends its own listen address and the talk address of the device at primary address 15, then inputs a sequence of bytes and stores them in string variable Setting\$.

DCL Received During Input When 4041 is a Talker/Listener

If the 4041 is in talker/listener mode (not acting as controller-in-charge) and the DCL or SDC command is received during execution of an INPUT statement involving the GPIB driver, the 4041 aborts the input and generates a DCL interrupt. The value of any numeric variable whose input was aborted is unchanged; the value of any string variable whose input was aborted is set to null.

If the INPUT statement that was interrupted includes an ALTER clause, the contents of the ALTER clause are returned as though no data had been input.

The PRINT Statement (GPIB)

| | |
|--------------------------|---|
| Syntax Form: | [line-no.] PRINT clause-list: {numexp} [,numexp]... {strex} [,strex]... |
| where: | clause-list = # {numexp}[BUFFER strvar][USING {numexp}][INDEX {strex}] {strex} {strex} {numexp} |
| Descriptive Form: | [line-no.] PRINT {print-clauses;}print-list |

PURPOSE

To transfer data from program variables to instruments on the bus.

EXPLANATION

If the 4041 is the controller-in-charge and PRI < > 31, the following bus traffic precedes the print operation:

- ATN SPD (if serial poll mode active)
- ATN UNListen (if one or more listeners are currently addressed)
- ATN My-Talk-Address (the MA value plus 64)
- ATN Other-Listen-Address (the PRI value plus 32, and the SEC value plus 96)

If the 4041 is the controller-in-charge and PRI < > 31, the following bus traffic follows the print operation:

- ATN UNTalk
- ATN UNListen

If the 4041 is a talker/listner, then it waits to be talk-addressed before sending the data. This must occur within TIM seconds, or a timeout error occurs.

Each data byte must be transmitted within TIM seconds of the previous data byte, or a timeout error occurs.

The EOM parameter setting directs the message termination activity of the 4041. The value of the EOM parameter specifies what characters, if any, are to be appended to messages to signify end-of-message.

If EOM = < 0 >, then no characters are appended to the message bytes. The EOI line is asserted concurrently with the last byte of the message to be transmitted.

If EOM = < 255 >, the carriage-return and line-feed characters are appended to the message bytes, and the EOI line is asserted concurrently with transmission of the line-feed.

Any other value of the EOM parameter specifies the character to be appended to the message bytes to signify end-of-message. The EOI line is always asserted concurrently with the transmission of this character.

If a semicolon is used to terminate a PRINT list, the 4041 does not transmit the EOM character after transmitting its message, nor does the 4041 assert the EOI line with the last byte of the message.

Default Logical Units

If a PRINT statement is executed that specifies a logical unit number in the closed interval 0 to 30 that has not been previously opened, then an implicit open is executed using the stream spec

```
"GPIB0(PRI = '&str$(logical-unit)&',TIM = 4):"
```

This logical unit then remains open until a CLOSE statement closing the logical unit is executed, or until an END statement is executed.

Example.

In Example 9-1, lines 1910 through 1930 assign values to variables Freq, Ampl, and Func\$.

Line 1940 unlistens all devices on the bus, addresses the 4041 to talk, addresses the device at primary address 10 to listen, then sends the following over the data lines:

- the string "freq"
- the EOH character (default:space)
- the value of numeric variable Freq
- the EOU character (default:semicolon)
- the string "ampl"
- the EOH character
- the value of numeric variable Ampl
- the EOU character
- the string "func"
- the EOH character
- the value of string variable Func\$
- the EOM character(s), (default: <cr> <lf>), with EOI asserted as the last character is transmitted.

The result is to set the FG5010 to transmit a 1-MHz square wave with an amplitude of 0.5 volt.

For more information about message units and the EOH, EOU, and EOM characters, see the description of the PRINT statement in Section 8, *Input/Output*.

```
1900 ! Assume device 10 is an FG5010 function generator
1910 Freq=1E6
1920 Ampl=0.5
1930 Func$="square"
1940 Print #10:"freq";freq,"ampl";ampl,"func";func$
```

Example 9-1.

LOW-LEVEL DATA TRANSFERS AND INSTRUMENT CONTROL

Most Tektronix GPIB instruments can be controlled by exchanging messages representing commands or data, using the PRINT and INPUT statements.

Some instruments, however, might not be controllable (or not conveniently, anyway) using PRINT and INPUT; instruments that don't understand ASCII are examples.

Such instruments can be controlled using the "low-level" commands RBYTE and WBYTE, in concert with the GPIB functions for WBYTE.

In addition, using the WBYTE statement with the PPC and PPE functions is the only way to configure instruments on the bus for parallel polling.

The RBYTE Statement (GPIB)

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] RBYTE [#numexp:] {strvar [,numvar]} [#strexp:] {numvar [,numvar...]} |
| Descriptive Form: | [line-no.] RBYTE [#logical-unit:] {strvar[,state-of-EOI]} [#stream-spec:] {numeric-variable-list} |

PURPOSE

To transfer 8-bit bytes from the bus into program variables.

EXPLANATION

If the 4041 is controller-in-charge, it must be listen-addressed to receive data from the bus or an error occurs. The 4041 becomes listen-addressed when the ATN(MLA) function is executed (by means of the WBYTE statement).

If the 4041 is a talker/listener, it waits to be listen-addressed before receiving any data from the bus. If it is not addressed within TIM seconds, a timeout error occurs.

Each numeric variable in the variable list causes one byte to be read from the bus, unless the numeric variable immediately follows a string variable. The value input also indicates the EOI state: a positive value indicates no EOI, while a negative value indicates EOI.

Each string variable in the variable list causes the dimensioned length number of bytes to be read from the bus, or all the bytes up to and including one with EOI asserted, if this happens before the dimensioned length is reached. The state of EOI is placed in the numeric variable immediately following the string variable, if one exists.

Numeric arrays cause one byte to be read from the bus for each array element. The values placed in the array are computed using the same rules specified for simple numeric variables.

String arrays cause bus reads for each string array element, using the same rules specified for simple string variables. If a numeric variable immediately follows the string array, then the state of EOI for the final element read is placed in the numeric variable.

If string variables are specified in the list without having been explicitly dimensioned, then they are given a default dimension of 1.

Example.

```

200 Dim string$ to 80
210 Integer eoistate
220 Wbyte atn (unl, mla, 65)
230 Rbyte string$, eoistate
240 Print string$
250 If not (eoistate) then goto 230
260 Wbyte atn (unt, unl)

```

Line 200 dimensions a string variable String\$ of length 80 characters. Line 210 declares an integer variable Eoistate.

Line 220 asserts the ATN line, unlistens all devices on the bus, then sends the 4041's listen address, followed by the talk address of the device at primary address 1.

Line 230 then reads a sequence of bytes from the bus, and stores them in string variable String\$. Bytes will be read until the EOI line is asserted or until 80 bytes have been read, whichever comes first. The state of the EOI line (1 if asserted, 0 if not) when the last byte is transmitted is stored in numeric variable Eoistate.

Line 240 prints the contents of string variable String\$.

Line 250 tests the contents of numeric variable Eoistate, to see if the EOI line was asserted when the last byte was transmitted. If it was not, control returns to line 230.

Line 260 untalks and unlistens all devices on the bus.

```

250 Integer number
260 Wbyte atn (unl, mla, 65)
270 Rbyte number
280 Print number
290 If number >=0 then goto 270
300 Wbyte atn (unt, unl)

```

Line 250 declares an integer variable Number.

Line 260 asserts the ATN line, unlistens all devices on the bus, then sends the 4041's listen address and the talk address of the device at primary address 1.

Line 270 reads a value from the GPIB data lines and stores it in numeric variable Number. Line 280 prints this value.

Line 290 tests the value received to see if it is less than 0. (Bytes sent with the EOI line asserted are stored into numeric variables as negative numbers.) If not, control returns to line 270.

Line 300 untalks and unlistens all devices on the bus.

The WBYTE Statement (GPIB)

| | |
|--------------------------|---|
| Syntax Form: | [line-no.] WBYTE [# {numexp};] expression-list {strex} |
| Descriptive Form: | [line-no.] WBYTE [# {logical-unit};] sequence-of-expressions- {stream-spec} and-GPIB-functions |

PURPOSE

To execute GPIB functions and to transfer 8-bit bytes from program variables to the bus.

EXPLANATION

If the 4041 is controller-in-charge, it must be talk-addressed to send data to the bus, or an error occurs. The 4041 becomes talk-addressed when the ATN(MTA) function is executed.

If the 4041 is in talker/listener mode, it waits to be talk-addressed before sending any data on the bus. If it is not addressed within TIM seconds, a timeout error occurs.

The remainder of this description of the WBYTE statement describes the GPIB functions that can be used with WBYTE. For more information on the use of WBYTE to transfer data values, see the description of the WBYTE statement in Section 8, *Input/Output*.

The ATN Function

| |
|---|
| Syntax: ATN(Numexp[,NumArg]...) |
|---|

The ATN function causes the values in its argument list to be sent over the GPIB data lines with ATN asserted.

The 4041 must be the controller-in-charge to execute this function or an error is generated.

Each numeric argument must evaluate to an integer in the closed interval 0 to 255. Each argument is sent to the bus with ATN true. ATN remains true after the function has completed executing.

Example.

```
1000 Wbyte atn(unt,un1,mta,33),"SET?",eoi
```

The 4041 asserts the ATN line, sends the UNTalk and UNListen messages followed by its talk address and the listen address of the device at primary address 1 (1 + 32), then unasserts ATN and sends the string "SET?". The EOI line is asserted with the last byte of the message.

The DCL Function

Syntax:
DCL

This function executes the equivalent function ATN(20), which sends the universal command Device Clear to the bus.

The 4041 must be the controller-in-charge to execute the DCL function, or an error is generated.

Example.

```
1500 Wbyte dcl
```

The EOI Function

Syntax:
EOI

The EOI function sends the last byte of the preceding data element with the EOI line asserted.

This function must follow some output data in the WBYTE statement, or an error is generated.

Example.

```
1600 Wbyte "FREQ 1000",eoi
```

Sends the message "FREQ 1000" over the bus, and asserts the EOI line as the last byte is transmitted.

The GET Function

Syntax:
GET[(numexp[,numexp]...)]

Bus Traffic:
[ATN UNL]
[ATN Listen-Address-1]
[{ATN Listen-Address-2} ...]
ATN 8

The GET function sends the listen-addresses of devices specified in its argument list, followed by the Group Execute Trigger (GET) command, all with ATN asserted.

Used without arguments, the GET function sends a value of 8 over the GPIB data lines with ATN asserted.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Each listen address consists of a mandatory primary listen address part, and an optional secondary address part. Each part, if there, must be a numeric expression that evaluates to an integer in the closed interval legal to its appropriate part.

Valid primary listen addresses are in the closed interval 0 to 30, or 32 to 62. If the value is in the former interval, it is added to 32 before being sent to the bus.

Valid secondary addresses are in the closed interval 96 to 128. If the value is 128, then nothing is sent to the bus.

Example.

```
1700 Wbyte get(1,2,3,97)
```

Sends the GET command to the devices at primary address 1 and 2, and the device at primary address 3, secondary address 1 (96 + 1).

The GTL Function

Syntax:

```
GTL[(numexp[,numexp...])]
```

Bus Traffic:

```
[ ATN UNL ]  
[ ATN Listen-Address-1 ]  
[ [ATN Listen-Address-2] ... ]  
ATN 1
```

The GTL function sends the GoToLocal (GTL) command to the devices in its argument list.

Used without arguments, the GTL function sends a value of 1 over the GPIB data lines with ATN asserted.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

See the discussion under the GET function for legal listen addresses.

Example.

```
1790 Select "gpib1:"  
1800 Wbyte gtl(2,4,6)
```

Sends the GTL command to the devices at primary address 2, 4, and 6 on the optional GPIB interface port.

The IFC Function

Syntax:

```
IFC(numexp)
```

The IFC function pulses the IFC line for the amount of time (in seconds) specified by its argument.

Any argument less than 1E-4 pulses the line for 1E-4 seconds (100uS).

Executing this function unaddresses the 4041 interface as well as any interfaces connected to the bus.

The 4041 must be the system controller to execute this function, or an error is generated.

Example.

```
1900 Wbyte ifc(.010)
```

Pulses the IFC line for 10 milliseconds.

The LLO Function

Syntax:

```
LLO
```

Bus Traffic:

```
ATN 17
```

The LLO function sends the universal command LocalLock-Out (LLO).

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Example.

```
2000 Wbyte llo
```

The MLA Function

Syntax:

```
MLA
```

This function returns an integer value representing the 4041's listen address on the bus. It is computed by taking the MA value and adding 32. This function can appear anywhere a numeric value can appear.

Example.

```
2100 Wbyte atn(mla,66)
```

This statement asserts the ATN line and sends the 4041's listen address, followed by the talk address of the device at primary address 2 (2 + 64) over the data lines.

The MTA Function

Syntax:
MTA

This function returns an integer value representing the 4041's talk address on the bus. It is computed by taking the MA value and adding 64. This function can appear anywhere a numeric value can appear.

Example.

```
2200 Wbyte atn(mta,34)
```

This statement asserts the ATN line and sends the 4041's talk address, followed by the listen address of the device at primary address 2 (2 + 32) over the data lines.

The PPC Function

Syntax:
PPC(listen-address,data-line,sense
[,listen-address,data-line,sense]...)

Bus Traffic:

```
ATN UNL
ATN Listen-Address-1
ATN 5
ATN PPE or PPD
ATN UNL
{ATN Listen-Address-2}
{ATN 5}
{ATN PPE or PPD}
{ATN UNL} ...
```

The PPC function configures devices for parallel polling.

Arguments to the PPC function are given in triples. The first argument of the triple determines which device is being configured; the second argument determines which data line it

is being configured to respond on; and the third argument determines the sense of its IST bit on which it is to assert its assigned data line.

The address argument is interpreted as for the GET function (see the discussion of the GET function, earlier in this section).

To configure a device with a secondary address for parallel poll, both a primary and a secondary address must be sent before the data-line argument (i.e., send four values, not three).

The data-line argument determines whether a PPE or a PPD is output. If the data-line expression evaluates to an integer in the closed interval 1 to 8, then a PPE is generated. If the data-line expression evaluates to the integer 0, then a PPD is generated.

The SENSE numeric expression must evaluate to an integer in the closed interval 0 to 1. If SENSE = 0, the device will assert its assigned data line when parallel polled if its IST bit is set to FALSE. If SENSE = 1, the device will assert its assigned data line when parallel polled if its IST bit is set to TRUE.

The actual PPE bus value generated will be:
 $96 + \text{SENSE} * 8 + \text{DATALINE} - 1$.

The actual PPD bus value generated will be 112.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Example.

```
2200 Wbyte ppc(5,6,0,24,7,1)
```

This statement configures the device at primary address 5 to assert data line 6 on an IST value of false, and the device at primary address 24 to assert data line 7 on an IST value of true.

The PPU Function

Syntax:
PPU

Bus Traffic:
ATN 21

The PPU function clears a parallel poll configuration.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Example.

```
2300 Wbyte ppu
```

The REN Function

Syntax:
REN(numexp)

The REN function asserts or un-asserts the REN line, depending on the value of its argument.

If the absolute value of the numeric expression is greater than or equal to 0.5, then the REMOTE ENABLE line is sent true, else the REMOTE ENABLE line is sent false.

The REMOTE ENABLE line is sent true by default when the 4041 first enters the SC = YES state.

The 4041 must be the system controller to execute this function, or an error is generated.

Example.

```
2200 Wbyte ren(ready)
```

This statement asserts the REN line if the variable Ready evaluates to a number whose absolute value is greater than or equal to 0.5.

The SDC Function

Syntax:
SDC[(numexp[,numexp...])]

Bus Traffic:
[ATN UNL]
[ATN Listen-Address-1]
[[ATN Listen-Address-2} ...]
ATN 4

The SDC function sends the Selected Device Clear (SDC) command to the devices in the argument list.

Used without arguments, the SDC function sends a value of 4 over the GPIB data lines with ATN asserted.

Addresses in the argument list are interpreted as for the GET function (see the discussion of the GET function, earlier in this section).

The 4041 must be controller-in-charge to execute this function, or an error occurs.

Example.

```
2400 Wbyte sdc(10,14,98,20)
```

This statement sends the SDC message to the devices at primary address 10, the device primary address 14, secondary address 2 and the device at primary address 20.

The SPD Function

Syntax:
SPD

Bus Traffic:
ATN 25

The SPD function sends the universal command Serial Poll Disable (SPD).

The SPD function disables talkers from reporting status bytes when talk-addressed.

The 4041 must be controller-in-charge to execute this function, or an error occurs.

See the SPE function for an example of the use of SPD.

The SPE Function

Syntax:
 SPE

Bus Traffic:
 ATN 24

The SPE function sends the universal command Serial Poll Enable (SPE).

The SPE function enables devices to send their status bytes over the data lines when talk-addressed.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Example.

```
1000 Wbyte spe
1010 Wbyte atn(unl,m1a,65)
1020 Rbyte dev1stat
1030 Wbyte atn(66)
1040 Rbyte dev2stat
1050 Wbyte atn(67)
1060 Rbyte dev3stat
1070 Wbyte spd
```

Line 1000 sends the SPE command to all devices on the bus.

Line 1010 un-listens all devices, then sends the 4041's listen address and the talk address of device 1.

Line 1020 reads device 1's status byte into numeric variable Dev1stat.

Line 1030 talk-addresses device 2, and line 1040 reads device 2's status byte into numeric variable Dev2stat.

Line 1050 talk-addresses device 3, and line 1060 reads device 3's status byte into numeric variable Dev3stat.

Line 1070 sends to SPD command to all devices on the bus.

The SRQ Function

Syntax:
 SRQ(numexp)

The SRQ function asserts the SRQ line, signifying that the 4041 is requesting service from the controller-in-charge. The argument accompanying the SRQ function is the status byte that the 4041 responds with when serial polled.

The 4041 must be in talker/listener mode (not controller-in-charge) to execute this function, or an error is generated.

The numeric expression must evaluate to an integer in the closed interval 64 to 127 or 192 to 255, or an error is generated.

The first time this function is executed, the integer is placed in the interface chip, which causes the SRQ line to be asserted. Program execution then proceeds.

If this function is executed again before the current controller-in-charge polls the 4041, program execution suspends until the poll is executed. Then the new value is placed in the chip.

Example.

```
2500 Wbyte srq(200)
```

This statement asserts the SRQ line. The 4041 will respond with a status byte of 200 when polled by the controller-in-charge.

The TCT Function

Syntax:
TCT[(numexp)]

Bus Traffic:
ATN Talk-Address
ATN 9

The TCT function sends the Take Control (TCT) command to the device specified as an argument.

Used without arguments, the TCT function sends a value of 9 over the GPIB data lines with ATN asserted.

This function must be the last one performed by the 4041 requiring control of the ATN line, or an error is generated.

The 4041 must be controller-in-charge to execute this function, or an error is generated.

Example.

```
2600 wbyte tct(1)
```

This statement passes control to the device at primary address 1.

The UNL Function

Syntax:
UNL

The UNL function returns the value 63. This function is normally used as an argument in an ATN function to "un-listen" all current listeners.

Example.

```
2700 Wbyte atn(unl)
```

This statement un-listens all devices on the bus currently addressed to listen.

The UNT Function

Syntax:
UNT

The UNT function returns the value 95. This function is normally used as an argument in an ATN function to "un-talk" the current talker.

Example.

```
2900 Wbyte atn(unt)
```

This statement un-talks whichever device is presently addressed to talk.

SERIAL POLLS

When the 4041 is controller-in-charge, it can be made to poll devices requesting service by means of the POLL statement or the SPD/SPE WBYTE functions (see the discussion of these functions earlier in this section).

The usual mechanism for notifying the 4041 that a device requires service is by a device's asserting the SRQ line after the SRQ interrupt condition has been enabled. When the

4041 senses that the SRQ line is asserted, it transfers to a handler for the SRQ interrupt condition.

For more information about the SRQ interrupt condition, the ENABLE statement, and the ON statement (used to designate handlers for interrupt conditions), see Section 12, *Interrupt Handling*.

The POLL Statement

Syntax Form:
 FORM 1: [line-no.] POLL nvar, nvar[, nvar]; nexp[, nexp]
 FORM 2: [line-no.] POLL nvar, nvar[, nvar]; nexp[, nexp]; ...
 FORM 3: [line-no.] POLL nvar, nvar[, nvar]

Descriptive Form:
 FORM 1: [line-no.] POLL status, prildp[, secldp]; pri[, sec]
 FORM 2: [line-no.] POLL status, prildp[, secldp]; pri[, sec]; ...
 FORM 3: [line-no.] POLL status, prildp[, secldp]

where status = variable in which to store status byte
 prildp = variable in which to store primary address of last device polled
 secldp = variable in which to store secondary address of last device polled
 pri = primary address of a device to be polled
 sec = secondary address of a device to be polled

PURPOSE

To serial poll a specific instrument on the currently selected bus (form 1), or to serial poll a group of instruments on the currently selected bus (form 2), or to serial poll all possible addresses on the currently selected bus (form 3).

EXPLANATION

The 4041 must be the controller-in-charge on the currently SELECTed interface port, or an error is generated.

If form 3 of the POLL statement is used, the SRQ line must be asserted, or an error occurs.

The front-end bus traffic for this statement is:

UNListen
 SPE
 My-Listen-Address

Subsequent bus traffic depends on the statement form specified.

The interface waits for instrument responses using the SPE time value to detect the no-response condition.

When the statement completes operation, the terminating bus traffic is:

UNTalk
 UNListen
 SPD

Form 1

```
POLL status,primary-address[,secondary-address];  
primary-address-to-poll[,secondary-address-to-poll]
```

This form of the poll statement polls one device on the bus, whether or not SRQ is asserted. Thus, this form of the POLL statement can be used at any time to read the status byte of a given device.

Possible results of a POLL statement in this form are:

1. The 4041 reads the status byte of the device and returns that value and the device's address in the designated variables; or
2. The 4041 generates a "nobody home" error, if the device with the specified address does not respond to the poll within the amount of time given by the SPE parameter in the currently SELECTed stream spec. (The value of the SPE parameter defaults to .01 second.) This typically happens only if no device with the address specified is on the bus at the time of the poll.

If the command specifies a variable to record the device's secondary address and it has none, a value of "32" will be placed in that variable.

Example.

```
800 Poll status,priadd;4
```

This statement polls device 4 on the currently SELECTed GPIB interface port. The device's status byte is stored in numeric variable Status, and its primary address (which will always be 4, in this case) is stored in numeric variable Priadd.

Form 2

```
POLL status,primary-address[,secondary-address];  
primary-address-to-poll[,secondary-address-to-poll]  
[;primary-address-to-poll[,secondary-address-to-poll]...]
```

This form of the POLL statement differs from Form 1 in that more than one device address is included in the POLL list.

The SRQ line must be asserted in order to execute this form of the POLL statement.

This form of the POLL statement polls each device in the list until:

1. The device asserting SRQ is found, in which case its status byte and address are returned in the designated variables; or
2. The 4041 goes through the entire list without finding the device asserting SRQ (generates an error); or
3. One of the devices in the list does not respond to the poll within the amount of time designated by the SPE parameter ("nobody home" error); or
4. The SRQ line becomes un-asserted (e.g., the device asserting SRQ unasserts it while the poll is being conducted). This condition causes an error.

If a variable is specified to receive the secondary address of the device asserting SRQ and the device has none, a value of "32" is stored in that variable.

Example.

The POLL statement in Example 9-2 polls the following devices: the device with primary address equal to the value of numeric variable Dev1; the device with primary address Dev2; the device with primary address Dev3 and secondary address Sec1; and the device with primary address Dev3 and secondary address Sec2. When the device asserting SRQ is found, its status byte is stored in numeric variable Status, its primary address in numeric variable Priadd, and its secondary address in numeric variable Secadd.

```
900 Poll status,priadd,secadd;dev1;dev2;dev3,sec1;dev3,sec2
```

Example 9-2.

Form 3

POLL status, primary-address[,secondary-address]

Form 3 of the POLL statement could be called the “auto-polling” form. When executing this form of the POLL statement, no device addresses are specified to be polled. The 4041 polls each primary address from 0 to 30, then polls every possible combination of primary and secondary addresses (0-30 primary, 0-30 secondary), stopping when it finds a device asserting SRQ.

The SRQ line must be asserted in order to execute this form of the POLL statement.

Because the 4041 polls all possible device addresses in this form of POLL, no “nobody home” errors are generated.

If the SRQ line becomes un-asserted while the poll is being conducted (e.g. if the device asserting SRQ unasserts it while the poll is being conducted), an error is generated.

If the user specifies a variable to receive the secondary address of the device asserting SRQ and the device has none, a value of “32” is stored in that variable.

Example.

```
1000 Poll status,priadd,secadd
```

This statement polls all combinations of primary and secondary addresses on the currently SELECTed GPIB interface port. When the device asserting SRQ is found, the device’s status byte is stored in numeric variable Status, the device’s primary address is stored in numeric variable Priadd, and the device’s secondary address is stored in numeric variable Secadd.

PARALLEL POLLS

CONFIGURING/UNCONFIGURING PARALLEL POLLS

Parallel polls are configured or unconfigured by using the PPC, PPE, and PPU WBYTE functions. See the descriptions of these functions under the discussion of the WBYTE statement, earlier in this section, for more information.

EXECUTING PARALLEL POLLS

Parallel polls are conducted when the ATN(EOI) function is executed.

NOTE

The ATN(EOI) function need not be invoked within a WBYTE statement to conduct a parallel poll.

The ATN(EOI) Function

Syntax and
Descriptive Forms: ATN(EOI)

PURPOSE

To execute a parallel poll on the currently selected interface and return the result as a numeric value.

EXPLANATION

The 4041 must be controller-in-charge on the currently SELECTed GPIB interface port when this function is executed, or an error is generated.

The 4041 asserts the ATN and EOI lines simultaneously, then reads the data lines and returns the resulting value.

The value returned represents the responses for all configured bus devices, i.e., the value returned BAND the data-line-number gives the response of the device configured to respond on that data line.

This function may be used anywhere a numeric function is allowed.

Example.

```
100 Response=atn(eoi)
```

This statement conducts a parallel poll and stores the result in numeric variable response.

Section 10

RS-232-C DATA COMMUNICATIONS

INTRODUCTION

RS-232-C is a recommended standard, adopted by the Electronic Industries Association (EIA), describing a common interface for serial data transmission.

The RS-232-C standard gives guidelines to ensure that devices are compatible at three levels:

- Interface — both devices using the interface must agree on the transmission lines used and on their meanings. Devices must be able to transmit a digital signal or to reassemble that digital signal correctly at the interface.
- Hardware — each device must be able to translate the signal received at the interface into a usable form. The hardware must be able to control the interface.
- Logic — each device must know what to do with the information it receives. The devices must agree on the meaning and interpretation of the data transmitted across the interface. The devices must be made logically compatible.

In practice, RS-232-C interfaces are implemented in a variety of ways. Most manufacturers provide a variety of switches that can be set to provide compatibility with other equipment.

Configuring the RS-232-C interface involves setting these switches on one or both of the devices on either end of the interface.

The COMM (or "COMMO") driver controls the standard RS-232-C interface port on the 4041. The COMM1 driver controls the operational RS-232-C interface port on 4041 units equipped with Option 1 or Option 3.

RS-232-C interface ports on the 4041 are configured using "soft switches", parameters within stream specifications for the COMM0 and COMM1 drivers. Available switches, their meanings, possible values, and default values are given in Appendix D.

This section is not intended to be a tutorial on RS-232-C interfacing. The *4041 Operators Manual* describes the electrical aspects of the interface, and some of the concepts behind configuring the interface. For more information about the basics of RS-232-C data communications, consult the RS-232-C standard itself, or some publication such as "Essentials of Data Communications," available through Tektronix (Publication #AX-3915-1).

PARITY

The PARity parameter can have one of five possible settings: None, Odd, Even, High, or Low. The default is None.

When parity is set to none, no parity bits are added to outgoing characters, and no parity-checking is done on incoming characters.

When parity is set to odd or even, a bit with value 1 or 0 is added to outgoing characters to make the number of bits with value 1 in the character odd or even, as specified. Incoming characters are checked to ensure that the number of bits with value 1 is odd or even, as specified.

When parity is set to high, a bit with value 1 is added to each outgoing character.

When parity is set to low, a bit with value 0 is added to each outgoing character.

When the BITs parameter is set to 5, PARity must be set to None. When the BITs parameter is set to 9, PARity must be odd, even, high, or low.

STOP BITS

The STOp parameter specifies the number of stop bits to be transmitted between characters for synchronization. STOp can have a value of 1 or 2; the default is 2.

When the BITs parameter is set to 5 and the STOp parameter to 2, 1.5 stop bits are actually transmitted (i.e., a time equal to one-and-a-half times that required to transmit one bit is used for synchronization).

TYPEAHEAD BUFFER

The TYPeahead parameter specifies the size of the typeahead buffer, used during transfers of large amounts of data. The default (and minimum) size is 100.

The user need not be concerned about varying the typeahead buffer size unless a large amount of data is being transferred at high baud rates. In this case, the typeahead buffer size needed will vary with the baud rate and amount of data to be transmitted.

FLAGGING

Because different devices operate at different speeds, devices transferring data via the RS-232-C communications interface must have some way of signalling one another to indicate readiness to receive input. The name given to this signalling procedure is "flagging".

Upon power-up, the 4041 sets the FLAgging on the standard RS-232-C interface port (and on the optional port, if Option 1 or Option 03 is available) to OUTput. This means that, when the 4041 is transmitting data via a COMM driver, it will stop output when it receives an ASCII DC3 character (CTRL-S from a user terminal), and re-start output when it receives an ASCII DC1 character (CTRL-Q from a user terminal).

Other values for the FLAgging parameter include NOne, INput, BIDirectional, MODem, AMOdem, and EMOdem. The meanings of these values are as follows:

- INput: when receiving data via a COMM driver, the 4041 stops input when its typeahead buffer is almost full by sending an ASCII DC3 character. When the typeahead buffer is empty, the 4041 restarts input by sending an ASCII DC1 character.
- MODem: on input, stop transmitting characters to the 4041 when typeahead buffer is almost full by setting CTS line OFF. When typeahead buffer is empty, 4041 requests input by turning CTS line ON.
On output, the 4041 stops transmitting characters when it senses the DTR line OFF, and resumes transmission when it senses the DTR line ON.
- AMOdem: on input, stop transmitting characters to the 4041 when typeahead buffer is almost full by setting CTS line OFF. When typeahead buffer is empty, 4041 requests input by turning CTS line ON.
On output, the 4041 stops transmitting characters when it senses the RTS line OFF, and resumes transmission when it senses the RTS line ON.
- EMOdem: on input, stop transmitting characters to the 4041 when typeahead buffer is almost full by setting both the CTS and DSR lines OFF. When typeahead buffer is empty, 4041 requests input by turning both the CTS and DSR lines ON.
On output, the 4041 stops transmitting characters when it senses either the RTS or the DTR line OFF, and resumes transmission when it senses both the RTS and DTR lines ON again.

THE EDI PARAMETER

The EDI parameter specifies what kind of terminal you are using to communicate with the 4041. The value of the EDI parameter controls the way the 4041 handles line-editing control keys (Rubout, CTRL-E, -H, -K, -O, -W, -Y, and -Z).

If an output-only device such as a line printer is connected to an RS-232-C interface port, the value of the EDI parameter makes no difference.

The legal values of the EDI parameter are:

- STOrage: used with storage-tube terminals.
- RASter: used with "generic" raster-scan terminals. This is the default value for EDI.
- 402: used with Tektronix 4020-Series terminals.
- ANSi: used with raster-scan terminals that support ANSI Standard X3.64-1979 Kill to End of Line (KEOL), Delete Character (DCH), and Insert Character (INS) functions.
- 850: used with CT8500 terminals.

402, ANS, and 850 all default to do something "sensible" (i.e., friendly and not noticeable by or requiring any work on the part of the user) when you use these terminals for line editing. For instance, when you delete a character from a line using one of these terminals, the character disappears from the screen as well.

STOrage and RASter, however, treat characters such as RUBOUT and INSERT differently. More information about the way the 4041 interacts with a terminal with these parameter values appears in the discussion of "Line Editing", later in this section.

CONTROL CHARACTERS

Control characters are either executed (sent out as the actual ASCII code for that particular character) or are shown in up-arrow format (sent out as a two-character sequence of up-arrow (^) followed by the printable equivalent of the control character, which is the ASCII equivalent of the control character plus 64).

The following rules define when the COMM driver executes control characters and when it sends them out in up-arrow format.

1. CR (carriage return) and TAB (horizontal tab) are always executed.
2. WBYTE statements always execute control characters.
3. When FORMAT = ITEM is specified, control characters are always executed.
4. The one or two characters, as specified, that are output as EOM, are always executed.
5. When CR = CRLF or CR = LFCR, the linefeed is always executed.
6. SAVE statements always execute control characters.

The following rules exclude anything mentioned in Rules 1 through 6 (i.e., Rules 1 through 6 take precedence):

7. LIST and SLIST output control characters in up-arrow format.
8. Control characters echoed as the result of user input are in up-arrow format.
9. Control characters displayed as the result of an INPUT ALTER clause are in up-arrow format.
10. Control characters displayed as the result of a PRINT statement, INPUT PROMPT clause, or COPY statement are executed if CON = YES, and shown in up-arrow format if CON = NO.
11. If a LF (linefeed) is output in up-arrow format, based on the preceding specifications, then the LF parameter is ignored. If LF is to be executed, then it is sent as specified by the LF parameter (as LF, CRLF or LFCR), and the one or two characters are executed as specified.
12. The default for the CON parameter is YES (execute control characters).

CONTROL CHARACTER FUNCTIONS

Figure 10-1 shows the control-character functions for the keyboard, i.e., the functions invoked by pressing various keys at the same time as the Control (CTRL) key.

Table 10-1 groups these keys by functional group, lists the functions and the keys that invoke them, briefly describes the effects of each control key, and states whether the COMM driver must be the system console device in order for the control character to function.

Line Editing

For purposes of this discussion, the "line editing" keys include:

- RUBOUT (usually labeled as such on most terminals; the same functionality is also provided by CTRL-E and CTRL-H).
- CTRL-K (Kill to End of Line).
- CTRL-O (Insert Characters).
- CTRL-W (Move to Beginning of Line).
- CTRL-Z (Move to End of Line).

When the EDI parameter is set to 402, ANS, or 850, these keys work just as you would expect them to; RUBOUT deletes characters from the screen, CTRL-O inserts characters at the position of the cursor, etc.

NOTE

When EDI = 402, the terminal command character must be single back quote (ASCII 96) and the command lockout light must be off. Otherwise, commands that the 4041 sends to the terminal to format the display will appear on the terminal display.

When the EDI parameter is set to STO or RAS, however, the keys perform their intended function on the input buffer, but the characters echoed to the screen are different in some cases.

EDI = STO. When EDI = STO, the line editing keys function differently depending on whether the cursor is at the end of a line or somewhere before the end.

If the cursor is at the end of a line, RUBOUT returns a backslash (" \ "), followed by the characters being deleted. Typing a different character than RUBOUT returns a second backslash, followed by the characters to be added to the line.

| | | | | | | | | | |
|---------------------------|----------------------------|------------------------|-----------------------|-------|----------------------------|--------------------------|---------------------------|--------|----------------------------|
| Q | W | E | R | T | Y | U | I | O | P |
| CONTINUE OUTPUT | MOVE TO BEG. OF LINE | RUBOUT | RECALL LINE | STEP | DISPLAY INPUT BUFFER | ERASE CURRENT LINE | | INSERT | RECALL PREVIOUS LINE |
| A | S | D | F | G | H | J | K | L | |
| AUTO NUMBER | STOP OUTPUT | FUNCTION KEYS 11-20 | FUNCTION KEYS 1-10 | | RUBOUT | | KILL TO END OF LINE | | |
| Z | X | C | V | B | N | M | | | |
| MOVE TO END OF LINE | DELETE LINE | ABORT | AUTO LOAD | BREAK | RECALL NEXT LINE | | | | |

ALSO RECOGNIZE:
 CTRL — \: MOVE CURSOR LEFT
 CTRL —]: MOVE CURSOR RIGHT
 CTRL — ^: "META"-PREFIX

Figure 10-1. Control Character Functions.

Table 10-1
CONTROL CHARACTER FUNCTIONS

| Category | Function | Key(s) ^a | Description | Restriction |
|-------------------|----------------------|---|---|-------------|
| Program Execution | BREAK | BREAK CTRL-B | Causes program execution to be interrupted. Execution resumes when CONTinue is typed. | CONSOLE |
| | ABORT | CTRL-C | Abort. | CONSOLE |
| | STEP | CRTL-T | Executes next program line and stops. | CONSOLE |
| | AUTOLOAD | CTRL-V | Loads and runs file named "AUTOLD" from DC-100 tape or disk. | CONSOLE |
| Editing | RUBOUT | RUBOUT CTRL-E CTRL-H Backspace | Causes character to left of cursor to be erased. | |
| | KILL TO END OF LINE | CTRL-K | Delete from cursor position to end of input line. | |
| | RECALL NEXT LINE | CTRL-N | Recall program line following the one last recalled with CTRL-R, -P, or -N. | CONSOLE |
| | INSERT | CTRL-O | Insert characters at cursor position. | |
| | RECALL PREVIOUS LINE | CTRL-P | Recall program line previous to the one last recalled with CTRL-R, -P, or -N. | CONSOLE |
| | LINE RECALL | CTRL-R | Recall program line whose number was typed prior to CTRL-R. | CONSOLE |
| | ERASE INPUT | CTRL-U | Erase current input buffer. | |
| | START OF LINE | CTRL-W | Position cursor at start of current input line. | |
| | DELETE LINE | CTRL-X | Delete program line whose number was typed prior to CTRL-X. | CONSOLE |
| | DISPLAY INPUT LINE | CTRL-Y | Redisplay current input line. | |
| | END OF LINE | CTRL-Z | Position cursor at end of current input line. | |
| | CURSOR < | CTRL-\ CTRL-] | Move cursor one space to the left. | |
| CURSOR > | | Move cursor one space to the right. | | |
| FUNCTION KEYS | KEY FUNCTIONS 1-10 | CTRL-F | Function key, 1-10 range. CTRL-F is followed by a number in 1-9, 0 range. NOTE: CTRL-F followed by "0" calls function 10. | CONSOLE |
| | KEY FUNCTIONS 11-20 | CTRL-D | Function key 11-20 range. CTRL-D is followed by a number in 1-9, 0 range. NOTE: CTRL-D followed by "0" calls function 20. | CONSOLE |

^a To avoid immediate execution of a special control character, precede the control character by an ESCape character. For example, to enter CTRL-C into a string without executing an ABORT, type < esc > followed by < CTRL-C >.

Table 10-1 (cont)
CONTROL CHARACTER FUNCTIONS

| Category | Function | Key(s) ^a | Description | Restriction |
|------------|---------------|---------------------|---|-------------|
| FLAGGING | HALT OUTPUT | CTRL-S | Stops output from 4041. This feature is enabled when FLA = OUT or FLA = BID. | |
| | RESUME OUTPUT | CTRL-Q | Resumes output from 4041 after CTRL-S. This feature is enabled when FLA = OUT or FLA = BID. | |
| AUTONUMBER | AUTONUMBER | CTRL-A | Automatically number lines starting at 100 with increments of 10. Typing a subsequent CTRL-A causes exit from auto-number mode. | CONSOLE |

^a To avoid immediate execution of a special control character, precede the control character by an ESCape character. For example, to enter CTRL-C into a string without executing an ABORT, type <esc> followed by <CTRL-C>.

If the cursor is somewhere before the end of a line, RUBOUT prints an "at" sign ("@") over the character being deleted. The "at" sign does not go into the input buffer, but simply denotes that a character has been deleted at that position in the string.

CTRL-O "opens up" the line being entered to allow characters to be inserted. Characters to the right of the cursor are shifted ten spaces to the right; the spaces do not go into the input buffer.

EDI = RAS. When the EDI parameter is set to RAS, the line editing keys work similarly to the case where EDI = STO, except that RUBOUT deletes characters from the screen when the cursor is at the end of a line.

When the EDI parameter is set to STO or RAS, you will probably find CTRL-Y (Display Input Buffer) to be a useful key. CTRL-Y always returns the current contents of the input buffer. This allows the user to view the line being typed without any "at" signs, backslashes, or other characters that may have accumulated during the editing process.

Control Character Notes

CTRL-B is ignored when the 4041 is in idle mode, or when a break is already pending.

CTRL-C is ignored when ABORT is disabled, unless the 4041 is PAUSED.

CTRL-D and CTRL-F, and the numeric digit(s) following, are ignored when a function key is already in the function key queue (i.e., if KEYS are not enabled, the first function key pressed is placed in the function key queue). Subsequent function keys are ignored by the system and ring the bell on the terminal.

When the COMM is the system console, CTRL-B, -C, -D, and -F cannot be input via an RBYTE statement unless preceded by an ESCAPE character.

When the COMM is the system console and the 4041 is PAUSED, CTRL-A, -N, -P, -R, -T, -V and -X are executed. Otherwise, they are treated as data. These characters can be input as data via an RBYTE statement at all times.

CTRL-E, -H, -K, -U, -W, -Y, -Z, -\ , -], along with Escape and Rubout, are always treated as special keys when the value of the FORMAT parameter equals ASCII, and are always treated as data when the value of the FORmat parameter equals ITEM. These characters can always be input as data via an RBYTE statement.

CTRL-S and -Q are treated as special keys when FLA = OUT or FLA = BID, and are otherwise treated as data. These characters cannot be input as data via an RBYTE statement when FLA = OUT or FLA = BID.

The ESCAPE character can always be input when the FORmat parameter is set to ITEM. When the FORmat parameter is set to ASCi, the ESCAPE character can only be input by preceding it with itself (i.e., typing <esc> - <esc>).

When in doubt about whether or not to precede a control key by an ESCAPE character, always do so. If the ESCAPE character is not needed, it will be ignored.

The “Meta” Character: CTRL-^

When the COMM driver is the system console:

- Typing CTRL-^ followed by a “C” produces the same result as pressing the CONTINUE key on the P/D keyboard.
- Typing CTRL-^ followed by an “L” produces the same result as pressing the LIST key on the P/D keyboard.
- Typing CTRL-^ followed by an “R” produces the same result as pressing the RUN key on the P/D keyboard.
- Typing CTRL-^ followed by any other character (except B, C, D, or E when the COMM is the system console, or Q or S when FLA = OUT or FLA = BID) will cause both the CTRL-^ and that character to be ignored. Typing CTRL-^ followed by <esc> followed by any character will cause all three to be ignored.

This capability is especially useful when operating with a 4041 without Option 30 (Program Development ROMs). Effectively, it allows the COMM driver to be used as the system console device with such units, with no loss of capability during execution.

(Note that the Program Development ROMs are still necessary to develop programs on the 4041. Once saved in ITEM format, however, programs can be run with or without the P/D ROMs.)

ITEM FORMAT

When the FORmat parameter is set to ITEM:

- The CR, LF, EOM, and CON parameters are ignored. All characters are transmitted as-is in ITEM format.
- The INPUT and PRINT statements cannot be used to transfer data via the COMM driver; only the RBYTE and WBYTE statements can be used. If the functionality of an INPUT or PRINT statement is desired while FORmat = ITEM, use a combination of INPUT and GETMEM or PUTMEM and PRINT statements.

Section 11

SUBPROGRAMS & USER-DEFINED FUNCTIONS

INTRODUCTION

This section discusses statements in 4041 BASIC relating to subprogram segments that: (1) begin with a SUB statement and are entered via the CALL statement; or (2) begin with a FUNCTION statement.

Statements relating to a third kind of subprogram segment called "handlers" are discussed in Section 12, *Interrupt Handling*. Handlers can begin with a SUB statement, but are not entered via a CALL statement. Instead, they are entered when the condition they are meant to handle is sensed after the handler has been created and the condition enabled. (See Section 12 for more information.)

DEFINITIONS OF SUBPROGRAMS AND USER-DEFINED FUNCTIONS

There are two kinds of subprograms in 4041 BASIC: those defined by the SUB statement and those defined by the FUNCTION statement.

Subprograms defined by the SUB statement are referred to in this manual as "Subprograms" (capital S), while subprograms defined by the FUNCTION statement are referred to as "user-defined functions", or simply "functions".

Since Subprograms and user-defined functions are very similar, in this manual they are often discussed together using the term "subprograms". When the "s" in subprogram is capitalized, the discussion either refers only to SUB-statement-defined subprograms or to a particular Subprogram being used as an example. When the "s" in subprogram is not capitalized, the discussion applies to either Subprograms or user-defined functions.

DIFFERENCE BETWEEN SUBPROGRAMS, USER-DEFINED FUNCTIONS, AND GOSUB SUBROUTINES

The main purpose of a subprogram is to transfer control to a specified segment of the program, perform some action, and (usually) return control to the point in the program from which control was transferred out.

Although Subprograms and user-defined functions are very similar, they have two major differences: (1) the ways they are called into use; and (2) the ways they affect the execution of the program when they are used.

A Subprogram is brought into use by means of the CALL statement. For example:

```
1590      Call swap(x1,x2,x3)
      .
      .
9990      End
10000 Sub swap(a,b,c)
      .
      .
10080      Return
10090      End
```

Line 1590 calls Subprogram SWAP, giving as arguments X1, X2, and X3. Control transfers to the first line of Subprogram SWAP (line 10000), where the arguments X1, X2, and X3 map into parameters A, B, and C, respectively. (We will have more about parameters shortly.)

Execution continues through line 10080, whereupon control returns to the line following the one in which the subprogram was called. (In this case, that's the line after 1590.)

INTRODUCTION

Line 10090 marks the end of the Subprogram.

Notice that the Subprogram assigns no value to its own name. The CALL statement simply calls the Subprogram, control transfers to that Subprogram, the Subprogram is executed, and control returns to the line following the call.

A user-defined function, on the other hand, returns a value in its function name. For example,

```

1590     Sw=formula(x1,x2,x3)
      .
      .
9990     End
10000 Function formula(a,b,c)
10010     Formula=3*a^2+4*b+c
10020     Return
10030     End

```

Line 1590 invokes the function FORMULA. (Note: You “call” a Subprogram, but “invoke” a function.)

Control then transfers to the first line of function FORMULA (line 10000.)

Somewhere in the body of the function, a value is assigned to a variable that is the same as the function name. In this case, line 10010 assigns a value to a variable called FORMULA.

When the 4041 encounters a RETURN statement, control returns to the line that invoked the function. Upon return, the value of FORMULA is assigned to variable SW and the statement finishes executing. Program execution continues with the next line.

SUMMARY: The major differences between a Subprogram and a function are:

1. You call a Subprogram with a CALL statement, but you “invoke” a function within another statement.
2. Control returns to the line following the one that called a Subprogram, but returns to the line that invoked a function.
3. A function returns a value to its function name; a Subprogram does not.

PARAMETERS

A subprogram can be thought of as a “program within a program.” Subprograms accept input from a main program or calling subprogram, and perform an action or return a value.

Subprograms receive input by means of arguments and parameters. SUB and FUNCTION statements list the parameters of the Subprogram or function they define. Statements that call/invoke Subprograms/ functions list arguments that “map into” these parameters. For example:

```

      480     Call blark(x1,x2,x3)
      .
      .
1490     End
1500 Sub   blark(a,b,c)
1510     D=2*a^2+8*b-54*c
1590     Return
1600     End

```

The CALL statement in line 480 “maps” the values of X1, X2, and X3 into parameters A, B, and C, respectively, of Subprogram BLARK. X1, X2, and X3 are called “arguments” of Subprogram BLARK.

“Value” and “Reference” Parameters

There are two kinds of parameters used in 4041 BASIC subprograms: (1) “value” parameters; and (2) “reference” parameters.

Value parameters and reference parameters do different things with the arguments that are passed to them when a Subprogram is called or a function is invoked.

“Value” parameters copy the value of an argument and use the copy during execution of the subprogram.

“Reference” parameters copy the address of the variable passed as an argument. The subprogram then reads from or writes to that address when the parameter is referred to in the subprogram.

The major difference in function between value and reference parameters is that the value of a variable used as a value parameter is NEVER changed by a subprogram, but the value of a reference parameter MAY be.

Reference parameters are distinguished from value parameters in SUB or FUNCTION statements by the VAR keyword. Value parameters are listed before the VAR keyword, while reference parameters are listed after.

Examples:

```
2000 Function smurf(a,b var c,d)
```

In this example, A and B are value parameters; C and D, coming after the VAR keyword, are reference parameters.

```
2500 Sub price(var x1,x2,x3)
```

Parameters X1, X2, and X3 of Subprogram PRICE are reference parameters. Changing the values of these variables also changes the values of the corresponding arguments in the statement that called the subprogram.

Uses of Value and Reference Parameters

A good general rule to help programmers choose between value and reference parameters is to use value parameters to pass values into a subprogram and reference parameters to return them.

Other factors that must be considered include whether or not the data to be passed into the subprogram must be preserved. If the data must be preserved (because they will be used in later calculations), the use of value parameters is dictated. If the data need not be preserved but will be changed by the subprogram instead, the use of reference parameters is called for.

Another consideration is storage requirements for large arrays of data. Because value parameters create a "copy" of the data passed into them, they require more memory than reference parameters. In effect, passing an array into a value parameter sets up two arrays of equal size. If memory space is a potential problem, it is best to use a reference parameter to pass the data, taking care that no array contents are inadvertently changed within the subprogram.

LOCAL AND GLOBAL VARIABLES

Just as subprograms use two kinds of parameters, so do they use two kinds of variables: "local" variables and "global" variables.

A global variable is a variable that has a "universal" value, i.e., its value is the same no matter where it is referred to in the program.

A local variable is a variable that has a specific value within a given subprogram. References to variables with the same name outside the subprogram (whether the reference is to global variables or to the local variables of another subprogram) may have a different value.

In 4041 BASIC, all subprogram parameters are local variables. In addition, local variables can be defined by means of the LOCAL keyword within a SUB or FUNCTION statement.

In 4041 BASIC, all variables local to a given Subprogram or user-defined function must be named in the subprogram's SUB or FUNCTION statement. Thus, all subprogram parameters are automatically local variables, as well as all variables named after the keyword LOCAL within the SUB or FUNCTION statement.

All other variables in the subprogram are global. Changing the value of a global variable, in either the main program or a subprogram, changes the value of that variable throughout the rest of the program.

Example:

```
100 I=1
110 J=2
120 K=3
130 Call example
140 Print "this is the main program"
150 Print "i=";i,"j=";j,"k=";k
160 End
200 Sub example local i,j
210 I=3
220 J=4
230 K=5
240 Print "this is subprogram 'example'"
250 Print "i=";i,"j=";j,"k=";k
260 Return
270 End
2500 Sub price(var x1,x2,x3)
*run
this is subprogram 'example'
i=3.0 j=4.0 k=5.0
this is the main program
i=1.0 j=2.0 k=5.0
```


INTRODUCTION

Line 200 defines I and J to be local variables for Subprogram "EXAMPLE". Values assigned to I and J within the Subprogram do not affect the values of global variables with the same name. Thus, lines 210 and 220 only change the values of I and J within Subprogram Example, but not in the main program.

However, since K is not a local variable, it is assumed to be a global variable. Line 230 therefore changes its global value. Thus, when control returns to the main program and line 150 is executed, K has a value of 5.0 instead of the value previously assigned to it in the main program.

Local Line Labels

It is a good programming practice to declare all line labels used within a subprogram to be local to that subprogram. This allows the same line label to be used in different subprograms.

Example:

```

200 Sub nerp local finis,condition
.
.
250   If (condition) then goto finis
.
.
280 Finis:   return
290   End
300 Sub foo local finis,condition
.
.
350   If (condition) then goto finis
.
.
390 Finis:   return
400   End

```

The same line label (finis) is used in Subprograms Nerp and Foo to indicate the exit point for the subprogram.

Local Variables as Protection Against Undesired Accessing

The primary reason for using local variables is to ensure that variables within a subprogram and variables outside a subprogram do not affect each other by being inadvertently given the same name.

A good example is the use of local variables as indices in FOR... NEXT loops. It is common programming practice to use the letters I, J, or K as indices in FOR...NEXT loops.

If these variables are used in both the main program and a subprogram, however, the values of the variables in one program segment may clobber the values of variables in the other, with disastrous effect. For example, suppose we tried to run the following block of code:

```

130   For j=1 to 5
140     Call task
150     Print "j=";j
160     Next j
490   End
500 Sub task
550   J=1
570   Return
580   End
*run
j=1.0
j=1.0
j=1.0
j=1.0
j=1.0
.
.

```

This program never exits the loop in lines 130 to 150. This is because J is not defined as a local variable in subprogram Task. Thus, J always has a value of 1 when control returns from the subprogram, and the loop in lines 130 to 150 never completes execution.

To remedy this problem, line 500 should read

```
500 Sub task local j
```

which makes J a local variable in Subprogram Task. Then, the code runs as follows:

```

130   For j=1 to 5
140     Call task
150     Print "j=";j
160     Next j
490   End
500 Sub task local j
550   J=1
570   Return
580   End
*run
j=1.0
j=2.0
j=3.0
j=4.0
j=5.0

```

INHERITING ENVIRONMENTS

A subprogram's "environment" consists of the values and resources that the subprogram has access to during its execution.

These include the ANGLE, AUTOLOAD, and UPCASE system environmental parameters (described in Section 5, *Environmental Control*).

They also include handlers for different kinds of interrupts that may occur (e.g., function-key interrupts, GPIB interrupts, etc.).

Subprograms "inherit" these environmental elements from the program segments that call them. Thus, if the ANGLE parameter in the main program is set to "1" (degrees), and the main program calls Subprogram Sub1, Sub1 "inherits" that value for its ANGLE parameter.

Similarly, if Sub1 left the value of the ANGLE parameter unchanged and called Subprogram Sub2, Sub2 would inherit a value of "1" for its ANGLE parameter.

Likewise, Sub1 inherits all interrupt handlers from the calling program segment. Thus, if the main program is set to transfer to Subprogram HANDLE when the SRQ line is asserted on the standard GPIB interface port, the same transfer will occur if SRQ is asserted during execution of Sub1.

Should a subprogram change the value of an environmental parameter during its execution, that parameter is returned to its original value when the subprogram stops executing.

ENTERING, EDITING, DELETING, APPENDING, AND RENUMBERING

The "segmented" structure of programs in the 4041 restricts the programmer's ability to freely insert, edit, delete, append, and renumber lines in subprograms.

These restrictions arise because the 4041 must always be able to determine which program segment a statement belongs to.

Because indiscriminate inserting, deleting, editing, appending, or renumbering of subprogram lines could make it unclear which program segment a given line is a part of, the following restrictions apply:

Inserting

1. All statements except SUB, FUNCTION, and END statements may be inserted into the body of any program segment.
2. SUB or FUNCTION statements may be inserted only after END statements (i.e., they must be the next-higher numbered lines in sequence after an END statement), with one exception.

EXCEPTION: The first statement of a program may be a SUB or FUNCTION statement. This allows a programmer to create a file consisting of a collection of subprograms, which may be appended or inserted as necessary into another program.

3. END statements may only be inserted after the last statement in the body of a program segment that does not already have an END statement. (No program segment can have two END statements.)

Editing

1. Statements within the body of a program segment may be edited normally.
2. If a SUB or FUNCTION statement is edited, its replacement must be a SUB or FUNCTION statement.

NOTE

Replacing one SUB or FUNCTION statement with another takes more translation time than replacing any other kind of statement, because the 4041 must erase references to the old subprogram, replace them with references to the "new" subprogram, and partially re-translate each statement in the "new" subprogram to check it for legality. The resulting delay could be noticeable for large subprograms.

INTRODUCTION**Deleting**

1. Statements within the body of a program segment may be deleted normally.
2. SUB and FUNCTION statements may not be deleted. To delete a SUB or FUNCTION statement, a "DELETE LINE subname" statement must be executed, deleting the entire subprogram.
3. END statements may be deleted, but such deletions may result in errors when the program is executed if the statement is not replaced.

Appending

1. Results of append operations must obey all rules given for inserting lines within program segments.
2. Appends may be done into the body of a program segment or to the end of an incomplete subprogram. A new program segment may also be appended after the last program segment in the current program, or between program segments.

In order to ensure clarity and avoid complexities resulting from deleting and appending parts of subprograms, it is considered good programming practice to delete or append entire subprograms only.

Renumbering

1. When renumbering results in a sequence change (i.e., a change in the order in which statements will be executed), the renumbering operation must obey rules for insertions into and deletions from program segments.
2. Use the RENUMBER statement to combine two subprograms A and B, as follows:
 - a. renumber the END statement of subprogram A and all of subprogram B, to make room within A for subprogram B's statements;
 - b. renumber the body of program B to put it after the last statement before the END statement in subprogram A;
 - c. delete subprogram B.
3. Use the RENUMBER statement to (effectively) insert subprogram B within subprogram A, as follows:
 - a. create the first line of subprogram B (i.e., a SUB or FUNCTION statement) outside subprogram A;
 - b. renumber the lines from subprogram A that are to go into subprogram B;
 - c. be sure to include an END statement in both subprogram A and subprogram B.

FUNCTION**The FUNCTION Statement**

| | |
|--------------------------|---|
| Syntax Form: | line-no. FUNCTION subname [parm-list][LOCAL loc-var-list] |
| where: | parm-list = (exp[,exp]... [VAR var[,var]...) loc-var-list = LOCAL var[,var]... |
| Descriptive Form: | line-no. FUNCTION subname [parameter-list] LOCAL local-variable-list |

PURPOSE

The FUNCTION statement marks the beginning of a program segment, and defines the parameters and local variables for a user-defined function.

EXPLANATION

A user-defined function is a subprogram that returns a value to its function name. It is invoked by another statement in the program, and upon completion returns control to the statement that invoked it.

In every other way, a user-defined function is identical to a Subprogram.

Naming User-Defined Functions

User-defined function names follow the same rules as other variable names: eight-character maximum; first character must be a letter; succeeding characters may be letters, numerals, or imbedded underscores; the function name may not be the same as a keyword.

One difference between user-defined function names and Subprogram names is that the last character of a user-defined function name may be a dollar sign (\$), indicating a function that returns a string value.

Returning a Value to the Function Name

A user-defined function must always return a value to its function name. This means that somewhere in the body of the function, an assignment must be made to the function name. The last value assigned to the function name is the value returned by the function.

Example:

```

250      X=2
260      Y=3
270      Z=cubesum(x,y)
.
.
900      End
1000 Function cubesum(a,b)
1010      Cubesum=x^3+y^3
1020      Return
1030      End

```

Line 270 invokes the function CubeSum, which returns the sum of the cubes of the two arguments used to invoke the function. Z is assigned a value of 35 when the statement completes execution.

User-Defined String Functions

User-defined functions can return a string value. The names of user-defined functions returning a string value must conform to the rules for forming string variable names (must end with "\$").

Example:

```

290      Str1$="b"
300      Str2$=next$(str1$)
310      Print "str1$=";str1$
320      Print "str2$=";str2$
.
.
1000     End
2000 Function next$(a$)
2010     Next$=chr$(asc(a$)+1)
2020     Return
2030     End
*run
str1$=b
str2$=c

```

Function Next\$ returns the next character in the ASCII code after the first character in its argument. Thus, Str2\$ in line 300 is assigned a value of "c".

Calling/Invoking Other Subprograms/User-Defined Functions

A user-defined function may call any Subprogram or invoke any other user-defined function, subject to one restriction: the subprogram being called/invoked must not be in the "active call sequence".

The "active call sequence" is the sequence of subprograms currently being executed.

For example, if the main program invokes Function A, which calls Subprogram B, which invokes Function C, the "active call sequence" consists of Function A, Subprogram B, and Function C. Subprogram C cannot call/invoke any of these subprograms. (NOTE: Subprograms and user-defined functions can NEVER call/invoke themselves.)

INTEGER, DIM, and LONG Statements

In order to return INTEGER or LONG values or arrays in a function name, an INTEGER, DIM, or LONG statement must be executed within the body of the function.

Example:

```

2000 Function intrtrn(x,y,z)
2010     Integer intrtrn
      .
2050     Intrtrn=x+y+z
      .
2090     Return
2100     End

```

Line 2010 makes IntRtrn an integer variable. Line 2050 therefore returns an integer value to the line that invoked the function.

Function names and local variables should be typed within the body of the function if they are to take on other than short-floating-point scalar or string scalar values. Parameters, however, "inherit" their types from the program segment that invoked the function.

"Inheriting" Parameter Types

User-defined function parameters "inherit" the types of the arguments passed to them when the function was invoked.

Thus, if an integer variable A is used as an argument to map into a parameter X, X becomes an integer variable for that execution of the function.

If, later, a long-floating-point variable L is used as an argument to map into X, X becomes a long-floating-point variable for that execution of the function.

Failure to Include a RETURN Statement in a Function

If a function does not contain a RETURN statement, program execution stops when the END statement defining the end of the function is encountered.

Passing Arrays and Array Elements as Parameters

Array elements may be passed as value parameters only.

Arrays may be passed as either value or reference parameters.

To change one or more elements within an array, the calling program segment should pass the entire array as a reference parameter and the position(s) of the element(s) within the array as value parameter(s).

Passing Subprograms as Parameters

A Subprogram name may be passed as a reference parameter to another Subprogram. A call to that parameter name will then call the Subprogram whose name was passed. The Subprogram whose name is passed may not include any parameters.

The SUB Statement

| | |
|--------------------------|---|
| Syntax Form: | line-no. SUB subname [parm-list][LOCAL loc-var-list] |
| where: | parm-list = (var[,var]... [VAR var[,var]...) loc-var-list = LOCAL var[,var]... |
| Descriptive Form: | line-no. SUB subname [parameter-list] LOCAL local-variable-list |

PURPOSE

The SUB statement marks the beginning of a subprogram segment, and specifies the parameters and local variables for a Subprogram.

EXPLANATION

The SUB and FUNCTION statements are identical in form (apart from their main keywords), and are very similar in function.

A SUB statement tells the 4041 that any program lines that follow until the next END statement are part of a Subprogram.

A FUNCTION statement, similarly, tells the 4041 that any program lines that follow until the next END statement are part of a user-defined function.

Subprograms are called and have parameters passed to them by a CALL statement, and return control to the line following the CALL statement that called them.

User-defined functions are invoked and have parameters passed to them within another statements, and return control to the line in which they were invoked.

Subprograms do not return a value in their subprogram name.

User-defined functions return a value in their function name.

Both Subprograms and user-defined functions return control to the program segments that called/invoked them when a RETURN statement is executed.

Naming Subprograms

The rules for forming Subprogram names are the same as the rules for forming numeric variable names: (1) eight-character maximum; (2) first character must be a letter; (3) successive characters must be letters, numbers, or imbedded underscores.

The dollar sign (\$) is not a legal character for Subprogram names.

The main program segment has a default name of MAIN. Program segment names can be used to refer to enter program segments, such as in the statement RENUMBER MAIN.

Handler Subprograms

Handler Subprograms are Subprograms written to handle any of several kinds of interrupt conditions that may occur, such as the pressing of a user-definable function key or the assertion of the SRQ line on the GPIB.

Handlers are a special subset of Subprograms, and are discussed in detail in Section 12, *Interrupt Handling*.

Calling/Invoking Other Subprograms/User-Defined Functions

A Subprogram may call any other Subprogram or invoke a user-defined function, subject to one restriction: the subprogram being called/invoked must not be in the active call sequence.

For example, if the main program calls Subprogram A, which calls Subprogram B, which calls Subprogram C, the “active call sequence” consists of Subprograms A, B, and C. Subprogram C, therefore, cannot call any of these Subprograms. (NOTE: A Subprogram or user-defined function can NEVER call/invoke itself.)

Subprogram and user-defined function names are globally visible. It is possible to have a local variable with the same name as a subprogram or user-defined function, but a subprogram containing such a variable cannot call or invoke the subprogram of that name.

“Inheriting” Parameter Types

Subprogram parameters “inherit” the types of the arguments passed to them from a calling program segment.

Thus, if an integer variable A is used as an argument to map into a parameter X, X becomes an integer variable for that execution of the Subprogram.

If, later, a long-floating-point variable L is used as an argument to map into X, X becomes a long-floating-point variable for that execution of the Subprogram.

Failure to Include a RETURN Statement in a Subprogram

If a Subprogram does not contain a RETURN statement, program execution stops as soon as the END statement defining the end of the Subprogram is encountered.

Section 12

INTERRUPT HANDLING

INTRODUCTION

This section discusses the statements the 4041 uses to handle interrupts.

The section starts with a discussion of the different kinds of interrupts the 4041 recognizes. It then discusses each different type of interrupt individually.

The statements used to handle interrupts are then presented, in alphabetical order.

INTERRUPTS

“Interrupt” is the term used to describe the branching of program control to a user-defined section of program (called a “handler”) when the 4041 senses a condition’s becoming true.

The handler handles the condition that caused the interrupt (in a manner determined by the programmer) and then takes one of several courses:

1. Returns control to the point in the program at which execution was interrupted.
2. Branches to a different point in the active call sequence, and resumes execution from there.
3. Calls the system handler for a condition (e.g., prints an error message and halts execution).

The 4041 recognizes five different kinds of interrupt conditions: the ABORT condition, the ERROR condition, the GPIB FUNCTION condition, the IODONE condition, and the USER-DEFINABLE FUNCTION KEY condition.

The ABORT Condition

This condition is sensed as true when the ABORT key is pressed on the front panel or P/D keyboard, or when CTRL-C is pressed on a computer terminal connected to the 4041 through an RS-232-C interface port.

The ERROR Condition

The ERROR condition is sensed as true when an error occurs during program execution.

GPIB Conditions

The 4041 can be made to branch to a user-defined handler when the 4041 senses any of several conditions coming true on the GPIB. Examples include the 4041 sensing a device asserting the SRQ, EOI, or IFC management lines, or the 4041 hearing its talk or listen address.

The IODONE Condition

The IODONE condition becomes true when an I/O operation is completed in proceed-mode.

The KEYS Condition

This condition becomes true when the user presses a user-definable function key on the system console device. (If a computer terminal connected to the 4041 through an RS-232-C interface port is the system console device, the user-definable functions are invoked by pressing CTRL-F or CTRL-D, followed by the function number.)

The SRQ Condition (OPT2 Driver)

This condition becomes true when a device connected to the Option 2 (TTL) interface port requests service from the 4041. The condition is called an “SRQ” condition because it is handled similarly to the SRQ interrupt on the GPIB driver; note, however, that THIS INTERRUPT DOES NOT AFFECT THE STATUS OF THE SRQ LINE ON THE GPIB.

DEFINING, ENABLING, AND LINKING

Three tests must be met before the 4041 will transfer control to a handler upon sensing a condition's coming true:

1. A handler for the condition must be defined.
2. The condition be enabled.
3. The condition and the its handler must be linked.

Defining Handlers

Two kinds of handlers can be defined to handle a condition: GOSUB handlers and CALL-type handlers.

The act of entering a handler into a program defines the handler.

The 4041's system ABORT and ERROR handlers are automatically defined on power-up. The user may define alternate ABORT and ERROR handlers, if desired.

Enabling Conditions

The 4041 is enabled to sense a given condition by means of the ENABLE statement.

The ABORT and ERROR conditions are automatically enabled on power-up. The user may disable the ABORT condition, if desired.

The IODONE condition is automatically enabled when the 4041 enters proceed-mode.

Any condition except the ERROR and IODONE conditions may be disabled using the DISABLE statement.

Linking Conditions and Handlers

Conditions and handlers are linked by means of the ON statement.

The ON statement defines the section of program to which control is passed when a condition is sensed.

The 4041's system ABORT and ERROR handlers are automatically defined, enabled, and linked on power-up.

The OFF statement negates the effect of a matching ON statement previously executed in the same program segment.

HANDLERS

Two types of handlers can be defined to handle a condition: GOSUB-type handlers and CALL-type handlers.

GOSUB-type handlers belong to the same program segments in which they are linked to conditions. ON statements for GOSUB-type handlers take the form

line-no. ON condition THEN GOSUB line

CALL-type handlers are distinct program segments in themselves; in fact, they are special cases of subprograms. ON statements for CALL-type handlers take the form

line-no. ON condition THEN CALL subprogram

GOSUB- and CALL-type handlers are formed the same as regular GOSUB subroutines and subprograms, with two exceptions:

1. The statement used to exit either handler must be an ADVANCE, RANCH, MONITOR, RESUME, or RETRY statement, and NOT a RETURN statement.
2. In the case of a CALL-type handler, the handler must not require that any parameters be passed to it (although the handler's SUB statement may define local variables for the handler's use).

Example.

In Example 12-1, lines 200 through 220 define CALL-type handlers for user-definable function keys 1 through 3. Lines 230 and 240 define a GOSUB-type handler for user-definable function keys 4 and 5. (Note the use of the RESUME statement in line 910, instead of the RETURN that usually ends a GOSUB subroutine.)

Subprogram K1H1 defines several new GOSUB-type handlers for user-definable function keys 1 through 5, then re-enables the keys. (User-definable function keys were automatically disabled when program control passed to a key handler.)

Each GOSUB-type handler performs some action, then returns to line 1080, which returns control to line 270. The handlers "unstack", so that function keys 1, 2, and 3 are once again linked to CALL-type handlers K1H1, K2H1, and K3H1, respectively, and function keys 4 and 5 are linked to the GOSUB-type handler starting at line 900.

```

200 On key(1) then call k1h1
210 On key(2) then call k2h1
220 On key(3) then call k3h1
230 On key(4) then gosub 900
240 On key(5) then gosub 900
250 Enable keys
260 Wait
270 Goto 260
.
.
900 Print "keys 4 and 5 are undefined at this time"
910 Resume
.
.
990 End
1000 Sub k1h1 ! handler 1 for key #1
1010 On key(1) then gosub 1100
1020 On key(2) then gosub 1200
1030 On key(3) then gosub 1300
1040 On key(4) then gosub 1400
1050 On key(5) then gosub 1500
1060 Enable keys
1070 Wait
1080 Resume
1100 ! Handler 1.2 for key #1
.
.
1190 Resume
1200 ! Handler 1.2 for key #2
.
.
1290 Resume
1300 ! Handler 1.2 for key #3
.
.
1390 Resume
1400 ! Handler 1.2 for key #4
.
.
1490 Resume
1500 ! Handler 1.2 for key #5
.
.
1590 Resume
1600 End
2000 Sub k2h1 ! handler 1 for key #2
.
.
2600 End
3000 Sub k3h1 ! handler 1 for key #3
.
.
3600 End

```

Example 12-1.

HANDLER VISIBILITY

All conditions except ABORT use the following “visibility” rules to determine the handler that control will transfer to when the condition occurs. Visibility rules for ABORT are presented in the discussion of the ABORT condition, later in this section.

CALL-type handlers for any condition are “visible from below” in the active CALL sequence.

This means that a CALL-type handler linked and enabled in one program segment is also linked and enabled in any program segment the first segment calls or invokes, unless the handler is superseded by another handler.

GOSUB-type handlers, on the other hand, are only “visible from below” when part of the main program segment, and are otherwise visible only within the program segments that create them. They are “transparent from below” when created in a subprogram segment. (EXCEPTION: ABORT handlers).

When a condition is enabled and sensed, the 4041 searches for handlers in the following sequence:

1. A CALL- or GOSUB-type handler linked in the current program segment.
2. A CALL-type handler linked in a preceding segment in the active call sequence.
3. A CALL- or GOSUB-type handler linked in the main program segment.
4. A system handler (ABORT and ERROR conditions only).

Example.

```
100 On key(1) then call key1hand
110 Enable keys
120 call a
130 End
200 Sub a
210 On key(1) then gosub 250
220 call b
230 Return
240 Print "gosub-type handler for key #1"
250 Resume
260 End
300 Sub b
310 Wait
320 Return
330 End
400 Sub key1hand
410 Print "call-type handler for key #1"
420 Resume
430 End
```

The main program sets up a CALL-type handler for function key #1, then calls Sub A.

Sub A sets up a GOSUB-type handler for function key #1, then calls Sub B.

If function key #1 is pressed while Sub B is executing, control will pass to Key1hand. Sub A’s GOSUB-type handler is “transparent” to Sub B.

Disabling Conditions Within CALL-Type Handlers

When control transfers to a handler, the condition causing the transfer is automatically disabled for the duration of the handler’s execution (except for ERROR handlers; the ERROR condition is never disabled).

The condition is automatically re-enabled when the handler finishes executing.

The user may over-ride the automatic disabling feature by enabling the condition within the handler. Once enabled, the condition may be disabled within the handler as well. Regardless of whether the condition is enabled or disabled within the handler, the condition is always automatically enabled when the handler finishes executing.

EXIT STATEMENTS FROM INTERRUPT HANDLERS

One of the following statements must be used to exit an interrupt handler: ADVANCE, BRANCH, MONITOR, RESUME, or RETRY.

(These statements are not legal for use with all interrupt conditions; see Table 12-1 to find which statements are legal with which conditions.)

ADVANCE

This statement (which can only be used in ERROR handlers) resumes execution with the statement after the one that caused the error.

BRANCH

This statement (which can be used in a handler for any condition) branches back to a given line in the active call sequence to resume execution.

MONITOR

This statement (which can only be used to exit from ABORT and ERROR handlers) returns control to the 4041's system ABORT or ERROR handler.

The system ABORT handler prints a message and aborts execution.

The system ERROR handler prints a message and halts execution after the line that caused the error.

RESUME

This statement (which can be used to exit from GPIB condition, user-definable function key, or iodone handlers) resumes execution from the point at which control was transferred.

RETRY

This statement (which can only be used to exit from ERROR handlers) re-tries the statement that caused the error.

Table 12-1
4041 INTERRUPTS

| Condition | Defined/Linked at Powerup? | Enabled at Powerup? | Enable/Disable | Exits | | | | |
|---------------|----------------------------|---------------------|----------------|---------|--------|---------|--------|-------|
| | | | | Advance | Branch | Monitor | Resume | Retry |
| ABORT | YES | YES | YES | | X | X | | |
| ERROR | YES | YES | NO | X | X | X | | X |
| GPIB FUNCTION | NO | NO | YES | | X | | X | |
| IODONE | NO | NO | NO | | X | | X | |
| KEYS | NO | NO | YES | | X | | X | |
| SRQ (OPT 2) | NO | NO | YES | | X | | X | |

ABORT

STATEMENT FORMS

```

[[line-no.] DISABLE ABORT

[[line-no.] ENABLE ABORT

[[line-no.] OFF ABORT

[[line-no.] ON ABORT THEN {GOSUB numexp}
                        {CALL subprogram}

```

EXPLANATION

The ABORT condition stops execution of the program and transfers control to the currently defined ABORT handler. The ABORT condition occurs when the ABORT key is pressed on the front panel or P/D keyboard, or when CTRL-C is pressed on a user terminal connected to the 4041 through an RS-232-C interface port. THE DEVICE ON WHICH ABORT/CTRL-C IS PRESSED NEED NOT BE THE SYSTEM CONSOLE DEVICE.

Setting Up ABORT Handlers

The 4041 powers up with a "system" ABORT handler defined and linked and the ABORT condition enabled. The user links the ABORT condition with a user-defined handler by means of the keywords ON ABORT THEN followed by a GOSUB or CALL.

Example.

```
2020 On abort then gosub 2500
```

Line 2020 links the ABORT condition with a handler at line 2500. (Line 2500 must be in the current program segment.) If the ABORT condition becomes true during execution of the current segment, control will transfer to line 2500.

Effectiveness of ABORT Handlers

The ABORT condition in the 4041 is implemented to allow the user to set up a user-defined ABORT handler in the main program segment in the "standard" case. Such a handler is globally visible, as long as no other program segment sets up a different ABORT handler.

Unlike handlers for other conditions, however, handlers for the ABORT condition do not "stack" and "unstack". Instead, linkage of a new handler causes the old linkage to "go away", and only the system ABORT handler takes its place. (You can think of it as stacking and unstacking just like any other condition, except ABORT has a stack just one element high.)

A CALL-type ABORT handler is visible from below in the active call sequence. It is effective as long as the subprogram that linked it is in the active call sequence.

A GOSUB-type ABORT handler is not visible from below, but is only effective when the subprogram that linked it is executing (except for GOSUB-type ABORT handlers linked in the main program segment).

Example.

```

100 On abort then call aborthan
.
200 Call a
.
300 ! if ABORT happens here,
310 ! 4041 goes directly to system handler
.
490 End !{main program}
500 Sub a
510 On abort then gosub 580
.
570 return
580 ! gosub-type handler for ABORT
.
600 Monitor
610 End ! {sub a}
700 Sub AbortHan
710 ! CALL-type handler for ABORT
.
790 Monitor
800 End ! {sub aborthan}

```

The main program sets up a CALL-type handler for the ABORT condition, then calls Sub A. Sub A sets up a GOSUB-type handler for the ABORT condition. When control returns to the main program, the ABORT condition is no longer linked to Sub AbortHan, but only to the system ABORT handler.

Disabling/Re-Enabling the ABORT Condition

The ABORT condition can be disabled by means of the DISABLE ABORT statement. If the ABORT condition is disabled, the ENABLE ABORT statement re-enables it.

OFF ABORT

The OFF ABORT statement destroys the linkage between a user-defined ABORT handler and the ABORT condition, without disabling the ABORT condition. In effect, OFF ABORT re-links the ABORT condition with the 4041's system ABORT handler.

Exits from ABORT Handlers

The legal exits from ABORT handlers are the BRANCH statement and the MONITOR statement.

The BRANCH statement causes control to transfer to a specified line upon completion of the ABORT handler. The target line of the BRANCH statement must be a line in the active call sequence.

The MONITOR statement causes the system ABORT handler to be invoked upon completion of the user-defined ABORT handler. The MONITOR statement causes the 4041 to do the same "housekeeping" as the RUN statement.

System ABORT Handler

The system ABORT handler prints the message SYSTEM ABORTED, and (if the ABORT condition occurs during program execution) the line number at which execution was aborted.

ERRORS

STATEMENT FORMS

```
ASK$("ERROR")

[line-no.] OFF ERROR(error-no.[TO error-no.],[,lunum[TO lunum]])

[line-no.] ON ERROR(error-no.[TO error-no.],[,lunum[TO lunum]])THEN {GOSUB numexp}
                                           {CALL subname}

[line-no.] TRAP
```

ERROR NUMBERS

Each error that can occur during program execution, or while the 4041 is executing statements in immediate mode, is numbered. The user defines a handler for an error by means of the ON ERROR statement.

Error numbers range from 1 to 32767. Not all numbers in this range, however, have an error associated with them that can be handled by the user. Errors 1 through 46 and 141 through 146, for example, cannot be handled by a user-definable error handler. Errors in these ranges cause control to transfer to the system error handler for these errors, even if the user has defined a separate handler for them. Refer to Appendix A for a complete listing of all error numbers and their meanings.

The user can define one handler to handle a range of error numbers by using the

```
line-no. ON ERROR (numexp [TO numexp])
      THEN {GOSUB line}
           {CALL subprogram}
```

form of the ON ERROR statement. When the TO keyword is not used, the first numeric expression gives the number of the error the user wishes to link to a handler. When the TO keyword is used, the two numeric expressions give the first and last error numbers, inclusive, of a range of errors that the user is linking to a handler.

Logical-Unit-Related Errors

A "logical-unit-related" error is defined to be any error that occurs in a statement specifying a logical unit number. Possible statements for logical-unit-related errors are OPEN, INPUT, PRINT, RBYTE, and WBYTE.

If I/O is being performed on several different devices, the user may wish to trap only errors coming from certain logical units and may wish to trap only certain of those. The user may do this with the following form of the ON ERROR statement:

```
line-no. ON ERROR (numexp [TO numexp]
                  [,numexp[TO numexp]])
      THEN {GOSUB line}
           {CALL subprogram}
```

The first pair of numeric expressions in this form of the ON ERROR statement operate as defined earlier.

The third numeric expression, when used without an accompanying TO keyword, specifies the logical unit over which the error must occur in order to transfer control to the specified handler.

When four numeric expressions are used in the ON ERROR statement, the third and fourth numeric expressions specify the range of logical unit numbers over which the error must occur in order to transfer control to the specified handler.

Logical unit numbers range from 0 to 32767.

If no logical units are specified in the ON ERROR statement, control transfers to the specified handler on occurrence of the error, regardless of logical unit.

SETTING UP ERROR HANDLERS

When an error occurs, the 4041 “matches” the error with the last handler linked with that error. Thus, in setting up error handlers, it is important to set up the most “general” handlers first, and the more “particular” or “restricted” handlers last.

Suppose a program segment set up error handlers in this order:

```
100 On error (300 to 400) then call h1
110 On error (381) then call h2
120 On error (381,12 to 24) then call h3
```

After this segment of the program is executed, errors 300 through 380 and 382 through 400 will cause control to transfer to Subprogram H1. Error 381, occurring on any logical unit other than logical units 12 through 24 (including no logical unit), will cause control to transfer to Subprogram H2. Error 381, occurring on logical units 12 through 24, will cause control to transfer to Subprogram H3.

A different order of ON statements, however, would have produced entirely different (and possibly undesirable) results. Suppose the same statements had been executed in this order:

```
100 On error (381) then call h2
110 On error (381,12 to 24) then call h3
120 On error (300 to 400) then call h1
```

In this case, line 120 “negates” the effects of lines 100 and 110. Any error from 300 to 400, inclusive, occurring on any logical unit (including no logical unit), transfers control to Subprogram H1.

LEGAL EXITS FOR ERROR HANDLERS

Legal exit statements for ERROR handlers are ADVANCE, BRANCH, MONITOR, and RETRY.

“STATEMENT” ERROR NUMBERS

There are two kinds of error numbers: “statement” error numbers and “specific” error numbers.

“Specific” error numbers, as the name implies, refer to a specific kind of error (e.g., division by zero, out-of-range operation, etc.).

“Statement” error numbers refer to any error that occurs while a particular kind of statement is being executed (e.g., 220 is the statement error number for the CLOSE statement).

When an error occurs, the 4041 looks to find the most recent ERROR handler set up for either the specific or the statement error number. If it finds such a handler, control transfers to it. If no such handler is found, control transfers to the 4041’s system error handler for the specific error.

Only the specific error number is returned by the ASK\$(“ERROR”) function. (Exception: proceed-mode I/O errors; see the discussion that follows.)

Example

In Example 12-2, line 100 sets up a handler for error 320, the statement error for FOR statements. When line 120 attempts to set the value of I out of the integer range, errors 80 (the actual error number) and 320 (the statement error number) are generated. Since a handler has been set up for error 320, control transfers to that handler. The ASK\$(“ERROR”) function returns the actual error number in its string.

```
100 On error(320) then call e320han
110 Integer i
120 For i=4.0E+4 to 4.1E+4
130 Print i
140 Next i
150 End
200 Sub e320han
210 Print "sensed an error in a FOR statement"
220 Print ask$("error")
230 Branch 150
240 End
*run
sensed an error in a FOR statement
80,120,-1,1
*
```

Example 12-2.

USE OF THE ASK\$("ERROR") FUNCTION

The ASK\$("ERROR") function returns a string containing four numeric arguments: the specific error number of the error being handled; the number of the line in which the error occurred; the logical unit related to the error (if not LU-related, returns -1); and the repetition count of the error (i.e., the number of consecutive times the same error has occurred without another error intervening).

Results of the ASK\$("ERROR") function can be stored in a string variable and analyzed using string functions.

When a proceed-mode I/O error is being handled, however, the ASK\$("ERROR") function returns a string with eight numeric arguments (see the discussion that follows).

ASK\$("ERROR") returns a string of four zeros separated by commas if no error handler is active.

PROCEED-MODE I/O ERRORS

When an error occurs during proceed-mode I/O, the error is serviced at the completion of the currently executing statement. The proceed-mode I/O is aborted, and control transfers to a handler for error number 999 (proceed-mode I/O error).

The ASK\$("ERROR") function returns a string containing eight arguments (instead of the normal four) when invoked while a proceed-mode I/O error is being handled.

The first four arguments give the proceed-mode error code (999), followed by three zeros separated by commas, followed by the actual error number, the line number of the proceed-mode I/O statement, the logical unit on which the error occurred, and the repetition count of the error (number of consecutive times the error has occurred without a different error intervening).

If the user wishes to continue with the proceed-mode I/O, the I/O operation must be resumed or restarted from within the handler for the proceed-mode error.

If the program is resumed (by means of the CONTINUE statement, etc.), execution resumes from the line at which execution was interrupted, NOT from the line containing the proceed-mode I/O statement. The proceed-mode I/O operation itself cannot be resumed, only restarted (i.e., by executing the proceed-mode I/O statement again).

The RETRY statement should not be used to exit from a handler for proceed-mode I/O errors. Use the ADVANCE, BRANCH, or MONITOR exits instead.

Example

Example 12-3 shows the string returned by ASK\$("ERROR") for a proceed-mode I/O error.

Because there is no device at primary address 1, line 130 causes error 812, "no listener on the bus", to be generated. The first four digits of the message returned by ASK\$("ERROR") indicate that the error occurred during proceed-mode I/O. The last four digits indicate that error 812 was sensed while the 4041 was executing line 130; the repetition count for this error is 1.

```
100      Set proceed 1
110      On error(1 to 32767) then call errhan
120      Open #1:"gpib(pri=1):" !nobody at this address
130      Print #1:"message for GPIB device"
140      Print "message"
150      End
200 Sub errhan
210      Print "error encountered"
220      Print ask$("error")
230      Advance
240      End
*run
error encountered
999,0,0,0, 812,130,1,1
message
*
```

Example 12-3.

USER-DEFINED ERRORS: THE TRAP STATEMENT

Error 1000 is reserved for a "user-defined" error. This error is intended for use when it can be determined that an error condition not defined by the error codes has occurred (e.g., the user determines that a certain combination of instrument settings on the GPIB indicates an illegal or undesirable condition).

The TRAP statement sets error 1000. Control passes to a handler for error 1000 whenever the TRAP statement is executed.

Example

In Example 12-4, line 100 sets up an error handler for a user-defined error. If the user inputs too low or too high a value in line 130, line 140 catches the error and transfers to the handler, which prints an appropriate message, prints the values of variables Low and High, then branches back to the INPUT statement.

EFFECT OF OFF ERROR STATEMENT

The OFF ERROR statement cancels the effect of any ON ERROR statement created in the current program segment whose arguments EXACTLY MATCH the arguments of the OFF ERROR statement.

It is important that the OFF ERROR statement exactly match the ON ERROR statement it is intended to cancel. For example, if the statement ON ERROR (1 TO 5) was executed in a program segment, the statement OFF ERROR (1) executed in the same segment would have no effect. Both the error number(s) and the logical unit number(s), if any are given, must exactly match those given in the corresponding ON ERROR statement for the OFF ERROR statement to be effective.

After the OFF ERROR statement is executed, occurrence of the error passes control to the most recently linked "globally-visible" handler: either a CALL-type handler linked in the active call sequence, a GOSUB-type handler linked in the main program segment, or the 4041's system handler for that error.

```

100   On error(1000) then call u_def
110   Low=5
120   High=100
130   Input prompt "value:";value
140   If value<low or value>high then trap
500   End
1000 Sub u def ! handler for user-defined error
1010  If value<low then print "too low" else print "too high"
1020  Print "low=";low
1030  Print "high=";high
1040  Print "please try again"
1050  Branch 130
1060  End
*run
value:1
too low
low=5.0
high=100.0
please try again
value:155
too high
low=5.0
high=100.0
please try again
value:50
*
```

Example 12-4.

GPIB CONDITIONS

STATEMENT FORMS

```
[line-no.] DISABLE GPIB-condition  
  
[line-no.] ENABLE GPIB-condition  
  
[line-no.] OFF GPIB-condition  
  
[line-no.] ON GPIB-condition THEN {GOSUB numexp}  
                                {CALL subname}  
  
where  
    GPIB-condition = {DCL(logical-unit)}  
                    {EOI(logical-unit)}  
                    {IFC(logical-unit)}  
                    {MLA(logical-unit)}  
                    {MTA(logical-unit)}  
                    {SRQ(logical-unit)}  
                    {TCT(logical-unit)}
```

WHEN GPIB CONDITIONS ARE RECOGNIZED

Once a GPIB condition is enabled, each of the GPIB conditions is recognized when the following requirements are met:

- The DCL condition is recognized when the 4041 is in talker/listener mode (NOT controller-in-charge) and receives the SDC or DCL commands over the bus (values of 4 or 20, received with ATN asserted).
- The EOI condition is recognized when the 4041 is operating as controller-in-charge, but is not participating in a data transfer, and senses the EOI line being asserted by some device on the bus.
- The IFC condition is recognized when the 4041 is operating as controller-in-charge but not system controller, and senses the IFC line being asserted by the system controller.
- The MLA condition is recognized when the 4041 is in talker/listener mode (not controller-in-charge) and receives its listen address over the bus.
- The MTA condition is recognized when the 4041 is in talker/listener mode (not controller-in-charge) and receives its talk address over the bus.
- The SRQ condition is recognized when the 4041 is operating as controller-in-charge and senses the SRQ line being asserted by some device on the bus.
- The TCT condition is recognized when the 4041 is in talker/listener mode (not controller-in-charge) and receives the TCT command over the bus (value of 9 with ATN asserted).

CONDITION ENABLED, NO HANDLER

If a GPIB condition is sensed when the condition is enabled but not handler is defined for it, an error is generated.

LEGAL EXITS FOR GPIB CONDITIONS

The legal exit statements for GPIB condition handlers are BRANCH and RESUME.

USE OF SELECTED STREAM SPEC

If no logical unit is specified, the 4041 uses the port given by the currently selected stream spec as the default. Attempting to use a non-GPIB stream spec as a default (e.g., if a stream spec for a device other than a GPIB interface port is the SELECTed stream spec), or to enable/disable a GPIB condition on a non-GPIB logical unit, results in an error.

EXAMPLE

```
100 Open #50:"gpiB:"  
110 Open #100:"gpiB1:"  
120 On dcl(50) then call dcl_h  
130 On srq(100) then call srq_h
```

Line 120 sets up a handler to transfer control to subprogram dcl_h if the DCL interrupt condition is sensed on the standard GPIB interface port. (Logical unit 50 is opened to that port.)

Line 130 sets up a handler to transfer control to subprogram srq_h if the SRQ interrupt condition is sensed on the optional GPIB interface port. (Logical unit 100 is opened to the optional GPIB interface port.)

Logical units 50 and 100 could have been opened to any device on their respective ports; the 4041 only uses the port designation for purposes of enabling/disabling GPIB conditions.

Refer to Section 9, *Instrument Control With GPIB*, for more information about GPIB programming.

IODONE

STATEMENT FORMS

```
[line-no.] OFF IODONE(lunum)

[line-no.] ON IODONE(lunum) THEN {GOSUB numexp}
                                {CALL subname}
```

AUTOMATIC ENABLING AND DISABLING

The IODONE condition is automatically enabled when a SET PROCEED 1 statement is executed.

The IODONE condition is automatically disabled when a SET PROCEED 0 statement is executed.

USE OF LOGICAL UNIT NUMBERS WITH ON AND OFF

The ON IODONE statement, when used with a logical unit number, transfers control to a user-defined handler when the IODONE condition becomes true on the specified logical unit, i.e., when proceed-mode I/O on that logical unit is completed.

When the ON IODONE statement is used without a logical unit number, control transfers to a user-defined handler when the IODONE condition becomes true on the system console device.

The OFF IODONE statement is used similarly; when used with a logical unit number, it negates the effect of an ON IODONE statement specifying the same logical unit. When used without a logical unit number, it negates the effect of an ON IODONE statement directed to the system console device. In both cases, the ON IODONE statement must have been previously executed in the same program segment for the OFF IODONE statement to be effective.

LEGAL EXITS FOR IODONE HANDLERS

Legal exit statements for IODONE handlers are BRANCH and RESUME.

EXAMPLE

```
200 On iodone(20) then call lu20_han
```

This statement transfers control to subprogram lu20_han when proceed-mode I/O is completed on logical unit 20.

SRQ INTERRUPTS (OPT2 DRIVER)

RELATED STATEMENTS

```
[line-no.] SET DRIVER opt2-stream-spec
[line-no.] OPEN #lunum:opt2-driver-spec
[line-no.] ON SRQ(lunum)THEN {GOSUB numexp}
                        {CALL subname}
[line-no.] OFF SRQ(lunum)
[line-no.] ENABLE SRQ(lunum)
[line-no.] DISABLE SRQ(lunum)
where
    lunum = logical unit number
    opt2-stream-spec = "OPT2(IREG = <numexp>,IVAL = <numexp>):"
    opt2-driver-spec = "OPT2:"
```

OPT2 PARAMETERS

The OPT2 driver (available on 4041 units equipped with Option 2 (TTL interface port) has two physical parameters and one ASK\$ parameter, as follows:

Physical Parameters for OPT2 Driver:

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|--|
| IREG | 0 - 127 | 0 | Number of register to be written into to turn off "SRQ" interrupt. |
| IVAL | 0 - 255 | 0 | Value to send to IREG register to turn off "SRQ" interrupt. |

ASK\$ Parameters for OPT2 Driver:

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|---|
| ICN | any integer | 0 | Number of interrupts sensed since "SRQ" interrupt was last disabled on OPT2 driver. |

The values of the physical parameters determine the way the 4041 handles SRQ interrupts from a logical unit associated with the OPT2 driver.

“ON SRQ” INTERRUPT CAPABILITY WITH OPT2 DRIVER

Devices connected to the 4041 via the TTL (Option 2) interface port may interrupt 4041 execution by means of the ENABLE SRQ and ON SRQ statements. Interrupts from devices connected to the TTL interface port may be turned off by means of the DISABLE SRQ and OFF SRQ statements.

Note that the devices connected to the Option 2 interface port DO NOT use the GPIB's SRQ line. The “ON SRQ”, “OFF SRQ”, “ENABLE SRQ”, and “DISABLE SRQ” statements are used because the interrupt is handled similarly to a GPIB SRQ interrupt.

SETTING UP INTERRUPT HANDLING FOR OPT2 DRIVER

The procedure for setting up interrupt handlers for the OPT2 driver is as follows:

1. Set the OPT2 driver parameters with a SET DRIVER statement.

Example:

```
200 Set driver "opt2(ireg=10,ival=128):"
```

2. Open a logical unit to the Option 2 interface port:

Example:

```
210 Open #1:"opt2:"
```

3. Define an SRQ handler for that logical unit:

Example:

```
220 On srq(1) then call srqhnd
```

4. Enable the SRQ interrupt on that logical unit:

Example:

```
230 Enable srq(1)
```

The IRQ line on the Option 2 interface port will now function like the SRQ line on the GPIB. When the IRQ line becomes asserted, the 4041 sends the value specified by OPT2 parameter IVAL to the register specified by OPT2 parameter IREG in the user's device, then transfers control to the user-specified handler for the interrupt.

The device connected to the Option 2 interface port must be configured to un-assert the IRQ line when it receives the IVAL value in the IREG register. If it does not do so, the 4041 will “hang” at the program line being executed until the interrupt is removed.

While the 4041 is handling an SRQ interrupt from the Option 2 interface port, subsequent interrupts from that port are disabled. They are automatically re-enabled when the 4041 resumes execution upon exiting the handler. If the user wishes, the SRQ interrupt may be explicitly re-enabled during execution of the handler by means of the ENABLE SRQ statement.

The ICN parameter keeps count of the number of SRQ interrupts received from the Option 2 interface port while a previous interrupt is being handled. The value of the ICN parameter is contained in the string returned by the ASK\$(“LU”) function for the logical unit associated with the Option 2 interface port. The ICN parameter value is cleared whenever the SRQ interrupt capability for the Option 2 interface port is enabled, either implicitly (upon completion of a handler), or explicitly (by means of the SET DRIVER statement).

The user should note that the 4041's SRQ interrupt routines for the OPT2 driver do not program the user's hardware, but only affect the way the 4041 treats SRQ interrupts from that driver. Hardware-dependent functions such as the transmission of values to enable or disable interrupts within the user's hardware should be handled by appropriate “OPT2OUT” romcalls.

LEGAL EXITS FOR OPT2 SRQ INTERRUPT HANDLERS

The legal exit statements for SRQ interrupt handlers on the OPT2 driver are BRANCH and RESUME.

USER-DEFINABLE FUNCTION KEYS

RELATED STATEMENTS

```
[line-no.] DISABLE KEYS

[line-no.] ENABLE KEYS

[line-no.] OFF KEY(key-number)

line-no. ON KEY(key-number)THEN {GOSUB numexp}
                                {CALL subprogram}
```

ENABLING AND DISABLING KEYS

The ENABLE KEYS and DISABLE KEYS statements enable/disable all the user-definable function keys. It is not possible to enable or disable some of the keys only. However, the user need not define a handler for every key.

ENTERING USER-DEFINABLE FUNCTION KEYS FROM A USER TERMINAL

When the system console device is a computer terminal attached to an RS-232-C interface port, users may enter user-definable function keys by pressing CTRL-F or CTRL-D, followed by one of the digits 0 through 9.

Pressing CTRL-F followed by a digit invokes functions 1 through 9 and function 10 (pressing <CTRL-F> <0>-invokes function 10).

Pressing CTRL-D followed by a digit invokes functions 11 through 19 and function 20 (pressing <CTRL-D> <0>-invokes function 20).

SYSTEM CONSOLE REQUIREMENT

The 4041 only accepts user-definable function keys from the system console device. Pressing a user-definable function key on the front panel or P/D keyboard when the COMM is system console, or pressing user-definable keys on a user terminal when the front panel is system console, rings the bell on the "offending" device.

CONDITION ENABLED, NO HANDLER

With the user-definable functions enabled, pressing a key for which no handler has been defined has no effect.

QUEUEING USER-DEFINABLE KEYS

If the user-definable keys are disabled, the 4041 queues one key and invokes the handler for that key as soon as the user-definable keys are enabled.

Since the KEYS condition is automatically disabled while handling a user-definable function key, this queueing feature allows the user to queue up one key while another key is being handled. After the first key is handled, the handler for the second key is invoked when the KEYS condition is re-enabled.

The ASK("KEY") function returns the number of the key awaiting service, if any. A value of "0" indicates that no key is awaiting service.

"THROWING AWAY" KEYS

Pressing a user-definable function key when the KEYS condition is disabled or when the function key queue already contains one key results in the key's being ignored (thrown away). In addition, the bell rings on the 4041 if the key was pressed on the front panel or P/D keyboard, or the bell on the user's terminal rings if the key was pressed from that source.

WHEN USER-DEFINABLE FUNCTIONS KEYS ARE HANDLED

User-definable function keys are handled after the statement during which the interrupt occurs is executed, except that pressing a user-definable function key during an INPUT or WAIT statement terminates that statement.

LEGAL EXITS

Legal exit statements from a user-definable function key handler are BRANCH and RESUME.

EXAMPLE

```
100 On key(1) then call key1han
110 On key(2) then call key2han
120 Enable keys
.
.
490 End
500 Sub key1han ! handler for function key 1
.
.
580 Resume
590 End
600 Sub key2han ! handler for function key 2
.
.
680 Resume
690 End
```

Lines 100 and 110 set up handlers for user-definable function keys 1 and 2, respectively.

The ADVANCE Statement

**Syntax and
Descriptive Forms:** [line-no.] ADVANCE

PURPOSE

The ADVANCE statement tells the 4041 to resume execution after statement executed is the statement following the one that caused the error.

EXPLANATION

The ADVANCE statement can only be used to exit from an ERROR handler.

BRANCH**The BRANCH Statement**

Syntax Form: [line-no.] BRANCH numexp

Descriptive Form: [line-no.] BRANCH target-line-in-active-call-sequence

PURPOSE

The BRANCH statement tells the 4041 to resume execution at a specific line after handling a condition. The line designated by the BRANCH command must be in the active call sequence.

EXPLANATION

The BRANCH statement may be used to return from any condition handler.

BRANCH unconditionally transfers control to a specified line in the active call sequence.

The "active call sequence" is the sequence of program segments currently being executed.

Any "pending" statements (GOSUBs, FOR..NEXT loops) are cleared when the BRANCH statement is executed.

In addition, any conditions that were automatically disabled because the 4041 started executing a handler for that condition are re-enabled. Effectively, the 4041 executes a RESUME for each condition handler and a RETURN for each CALL or function invocation back to the target program segment.

EXAMPLE

```

100 ! this is the main program segment
110 ! on key(1) then call key1_h
120 ! on srq then call srq_h
130 enable keys,srq
140 wait
150 goto 140
160 end
200 sub key_h
.
. ! somewhere in here, we get an SRQ
.
280 resume
290 end
300 sub srq_h
310 poll status,priadd,secadd
320 print "I have an srq from: ";priadd,secadd
330 print "with status byte: ";status
340 branch 140
350 end

```

Suppose this program were to execute, and function key 1 were pressed. Control would then transfer to Subprogram Key__h, and the KEYS condition would be disabled.

Suppose further that during the handling of function key 1, the 4041 sensed an instrument asserting the SRQ line on the GPIB. Control would then transfer to Subprogram Srq__h, which polls the devices on the bus, prints a message, and branches back to the main program segment.

Because Subprogram Key__h is in the active call sequence, and because the KEYS condition was automatically disabled when the 4041 started to execute a handler for a function key, the KEYS condition is re-enabled (along with the SRQ condition) when control re-enters the main program segment.

The DISABLE Statement

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] DISABLE {ABORT; [,...] {KEYS; {DCL[(numexp)]} {EOI[(numexp)]} {IFC[(numexp)]} {MLA[(numexp)]} {MTA[(numexp)]} {SRQ[(numexp)]} {TCT[(numexp)]} |
| Descriptive Form: | [line-no.] DISABLE {ABORT; [,...] {KEYS; {DCL[(logical-unit)]} {EOI[(logical-unit)]} {IFC[(logical-unit)]} {MLA[(logical-unit)]} {MTA[(logical-unit)]} {SRQ[(logical-unit)]} {TCT[(logical-unit)]} |

PURPOSE

The DISABLE statement disables a condition.

EXPLANATION

After a condition has been disabled, its occurrence no longer causes control to transfer to a handler, even if a handler for the condition is defined and linked.

The ABORT, KEYS, IODONE, and GPIB conditions are automatically disabled when control transfers to a handler for one of these conditions. The conditions are automatically re-enabled when control exits the handler.

The ERROR condition is always enabled, but if an error being handled re-occurs within the handler, control passes to the system handler for that error.

The ABORT condition powers up enabled, but may be disabled if the user so desires.

The ERROR condition may never be disabled.

The IODONE condition is automatically enabled when the 4041 enters proceed mode, and disabled when it exits.

DISABLE KEYS disables all user-definable function keys. All keys are always either enabled or disabled; it is not possible to have some enabled and some disabled. However, handlers need not be defined for all possible user-definable function keys.

The port associated with the logical unit number in a DISABLE "GPIB-CONDITION (logical-unit)" statement is the one on which the function is disabled. The standard GPIB interface port (port 0) is the default.

EXAMPLE

```
1500 Disable srq(50)
```

This statement disables the SRQ interrupt on the GPIB interface port associated with logical unit 50.

```
2000 Disable keys
```

This statement disables all user-definable function keys.

The ENABLE Statement

| | |
|--------------------------|---|
| Syntax Form: | [line-no.] ENABLE {ABORT} [,...] {KEYS} {DCL[(numexp)]} {EOI[(numexp)]} {IFC[(numexp)]} {MLA[(numexp)]} {MTA[(numexp)]} {SRQ[(numexp)]} {TCT[(numexp)]} |
| Descriptive Form: | [line-no.] ENABLE {ABORT} [,...] {KEYS} {DCL[(logical-unit)]} {EOI[(logical-unit)]} {IFC[(logical-unit)]} {MLA[(logical-unit)]} {MTA[(logical-unit)]} {SRQ[(logical-unit)]} {TCT[(logical-unit)]} |

PURPOSE

The ENABLE statement enables a condition.

EXPLANATION

After a condition has been enabled, its occurrence causes control to transfer to a handler for that condition, if a handler has been defined and linked with an ON statement when the condition occurs.

The ABORT condition powers up enabled. The ENABLE ABORT command is used to re-enable the ABORT condition after it has been disabled with the DISABLE ABORT command.

The ERROR condition powers up enabled and can never be disabled.

The IODONE condition is automatically enabled when the 4041 enters proceed mode, and disabled when it exits.

The ENABLE KEYS command causes control to transfer to a handler for a user-definable function key when the key is pressed, if the handler has been defined and linked with an ON statement when the condition occurs.

ENABLE KEYS enables all the user-definable function keys. It is not possible to have some user-definable function keys enabled and some disabled. However, handlers need not be defined or linked for all possible user-definable function keys.

The port associated with the logical unit given in an "ENABLE GPIB-FUNCTION (logical-unit)" statement tells the 4041 which GPIB interface port the condition is enabled on. The standard port (Port 0) is the default.

ENABLE DCL enables the DCL interrupt condition on the GPIB. When DCL is enabled, control passes to a DCL handler when the 4041 is in talker/listener mode (NOT controller-in-charge) and receives the SDC or DCL commands over the bus (values of 4 or 20, received with ATN asserted).

ENABLE EOI enables the EOI interrupt condition on the GPIB. When EOI is enabled, control passes to an EOI handler when the 4041 is operating as controller-in-charge, but is not participating in a data transfer. This interrupt is typically used to signal the end of remote data transfers, e.g., the 4041 tells one device to talk and another device to listen, then waits for the EOI line to be asserted (indicating that the talking device has delivered its message) before issuing any new bus messages.

ENABLE IFC enables the IFC interrupt condition on the GPIB. When IFC is enabled, control passes to an IFC handler when the 4041 is controller-in-charge but is not the system controller. The 4041 loses controller-in-charge status automatically when the system controller asserts IFC (whether or not the IFC interrupt is enabled). Therefore, when the 4041 is operating as controller-in-charge but not system controller, the IFC interrupt should always be enabled and a handler defined for it, to prevent the 4041 from attempting to continue execution as if it were still controller-in-charge after the system controller asserts IFC.

ENABLE MLA enables the MLA interrupt condition on the GPIB. When MLA is enabled, control transfers to an MLA handler when the 4041 is in talker/listener mode (not controller-in-charge) and receives its listen address over the bus.

ENABLE MTA enables the MTA interrupt condition on the GPIB. When MTA is enabled, control transfers to an MTA handler when the 4041 is in talker/listener mode (not controller-in-charge) and receives its talk address over the bus.

ENABLE SRQ enables the SRQ interrupt condition on the GPIB. When SRQ is enabled, control transfers to an SRQ handler when the 4041 is controller-in-charge and senses the SRQ line being asserted by some device on the bus.

ENABLE TCT enables the TCT interrupt condition on the GPIB. When TCT is enabled, control transfers to a TCT handler when the 4041 is in talker/listener mode (not controller-in-charge) and receives the TCT command over the bus (value of 9 with ATN asserted).

EXAMPLES

```
2500 Enable keys
```

This statement enables the user-definable function keys.

```
2600 Enable eoi(25)
```

This statement enables the EOI interrupt on the GPIB interface port associated with logical unit 25.

The MONITOR Statement

**Syntax and
Descriptive Forms:** [line-no.] MONITOR

PURPOSE

The MONITOR statement transfers control to the 4041's system ABORT or ERROR handler.

EXPLANATION

The MONITOR statement can only be used to exit from ABORT or ERROR handlers.

The 4041's system ABORT handler prints the message "SYSTEM ABORTED" and aborts program execution. The program must then be re-started with a RUN or DEBUG command, or by pressing the PROCEED key on the front panel.

The 4041's system ERROR handler prints a message giving the number of the error and the line in which the error occurred, then halts execution at that line. Execution can be resumed at that line by entering the CONTINUE statement from the P/D keyboard or user terminal (for 4041 units equipped with Option 30, Program Development ROMs), or by pressing the PROCEED key on the front panel.

The ON Statement

Syntax Form: [line-no.] ON {ABORT}
 {ERROR(numexp[TO numexp][, numexp[TO numexp]])}
 {IODONE(numexp)}
 {KEY(numexp)}
 {DCL(numexp)} THEN {GOSUB numexp}
 {EOI(numexp)} {CALL subname}
 {IFC(numexp)}
 {MLA(numexp)}
 {MTA(numexp)}
 {SRQ(numexp)}
 {TCT(numexp)}

Descriptive Form: [line-no.] ON {ABORT}
 {ERROR(error-no.[TO error-no.]
 [, logical-unit[TO logical-unit]])}
 {IODONE(logical-unit)}
 {KEY(key-no.)}
 {DCL(logical-unit)} THEN {GOSUB line}
 {EOI(logical-unit)} {CALL subprogram}
 {IFC(logical-unit)}
 {MLA(logical-unit)}
 {MTA(logical-unit)}
 {SRQ(logical-unit)}
 {TCT(logical-unit)}

PURPOSE

The ON statement “links” a condition and a user-defined handler. It does not enable the condition.

EXPLANATION

In order to transfer control to a user-defined handler upon occurrence of a condition, three things must apply: (1) the handler must be defined (i.e., the program lines making up the handler must actually be in the program); (2) the condition must be enabled; and (3) the condition and its handler must be “linked”.

The ON statement links a condition and a user-defined handler, allowing control to be transferred to the handler when the condition is sensed.

EXAMPLES

```

150 Set proceed 1
160 Open #1:"outfil(open=new,size=20000)"
170 On iodone(1) then call oneio
180 Enable iodone
190 Print #1:bigaray1,bigaray2,bigaray3
200 Wait 1000
.
900 End
1000 Sub oneio
.
1490 Resume
1500 End

```

Line 170 sets up a handler for completion of proceed-mode I/O on logical unit 1. Line 180 enables the IODONE condition.

Line 190 starts logical unit 1 printing three arrays. Line 200 then causes the 4041 to wait for the completion of the I/O operation on logical unit 1 (assuming it will take less than 1000 seconds to complete).

If I/O completes on logical unit 1 within the 1000-second waiting period, control transfers to subprogram OneIO. Upon return from the subprogram, execution resumes with the line after line 200.

```

1990 Open #1:"gpib1:"
2000 On srq(1) then call optpoll
2010 Enable srq

```

Line 2000 sets up a handler for SRQ interrupts on the optional GPIB interface port. After line 2010 is executed, devices asserting SRQ on the bus connected to the optional GPIB interface cause program control to transfer to subprogram "OptPoll".

The RESUME Statement

**Syntax and
Descriptive Forms:** [line-no.] RESUME

PURPOSE

The RESUME statement is used to resume execution from the point at which it was interrupted after exiting a GPIB CONDITION, IODONE, or USER-DEFINABLE FUNCTION KEY handler.

EXPLANATION

The RESUME statement may only be used to exit from GPIB CONDITION, IODONE, or USER-DEFINABLE FUNCTION KEY handlers.

The RETRY Statement

Syntax and Descriptive Forms: [line-no.] RETRY

PURPOSE

The RETRY statement re-tries the statement that caused an error.

EXPLANATION

The RETRY statement can only be used to exit an ERROR handler.

EXAMPLE

```

100   On error(80) then call handle
110   Integer number
120   Input prompt "number:" :number
130   Print "number=" ; number
140   End
200 Sub handle ! handler for error 80
210   Print "value out of integer range"
220   Retry
230   End
*run
number:36000
value out of integer range
number:-40000
value out of integer range
number:31000
number=31000

```

When the user attempts to input an out-of-range value in line 120, control passes to subprogram Handle, which warns the user that the value is out of range, then re-tries line 120.

The TRAP Statement

**Syntax and
Descriptive Forms:** [line-no.] TRAP

PURPOSE

The TRAP statement “sets” (i.e., causes) a user-defined error (error 1000). Control can then be passed to a handler for that error.

EXAMPLE

In Example 12-5, when the user attempts to input an out-of-range value in line 130, line 140 “traps” it, prints an appropriate message and transfers control back to the INPUT statement.

EXPLANATION

The TRAP statement only causes control to transfer if the user has defined a handler for error 1000 and linked it with the ON ERROR(1000) THEN... statement.

The handler for error 1000 can be exited via any legal exit for ERROR handlers (ADVANCE, BRANCH, MONITOR, or RETRY).

```
100 On error(1000) then call u_def
110 Low=5
120 High=100
130 Input prompt "value:":value
140 If value<low or value>high then trap
.
500 end
1000 Sub u-def ! handler for a user-defined function
1010 Print "that value is out of range"
1020 Print "low=";low
1030 Print "high=";high
1040 Print "please try again"
1050 Branch 130
1060 End
```

Example 12-5.

The WAIT Statement

Syntax Form: [line-no.] WAIT [numexp]

Descriptive Form: [line-no.] WAIT [number-of-seconds-before-next-statement- is-executed]

PURPOSE

The WAIT statement causes the 4041 to wait for a specified number of seconds before executing the next statement.

EXPLANATION

The WAIT statement idles the system for the number of seconds specified by the numeric expression after the WAIT keyword.

The WAIT statement is usually used as a convenient stopping point for a program to pause and await an interrupt. When an interrupt condition is sensed, the program branches to a handler for that interrupt.

If no interrupt occurs within the specified time, execution continues with the next statement.

A negative number given as an argument for the WAIT statement causes no waiting at all.

The maximum number that can be used as an argument for the WAIT statement is $2.1478E + 7$ seconds (approximately 248 days).

The default value for the WAIT statement is the maximum value.

The maximum resolution of the WAIT statement is 10 milliseconds.

If the system becomes PAUSEd during execution of a WAIT statement, execution resumes with the WAIT statement unless an immediate-mode statement transfers control elsewhere during the pause.

EXAMPLE

```
980 On srq then call polling
990 Enable srq
1000 Wait
```

This statement causes the 4041 to pause indefinitely before continuing with program execution. If a device on the bus sends SRQ (Service Request) in that time, control transfers to Subprogram "Polling".

Section 13

PROGRAM MANAGEMENT

INTRODUCTION

This chapter discusses statements in 4041 BASIC that transfer programs or portions of programs between the 4041's memory and a peripheral device.

The APPEND statement loads a program or program segment from a driver and adds it to the program already in memory.

The LOAD statement loads a program from a driver.

The SAVE statement sends a program or program segment to a driver.

APPEND**The APPEND Statement**

Syntax Form: [line-no.] APPEND strexp[,numexp[,numexp]]

Descriptive Form: [line-no.] APPEND stream-spec[,target-line[,max-increment]]

PURPOSE

The APPEND statement loads a program (or program segment) from a DC-100 tape file and adds it to the program already in memory.

EXPLANATION

The file designated by the stream spec is inserted into the current program at or immediately after the designated target line. Lines being inserted are numbered consecutively using the specified maximum increment or a calculated increment.

If no target line is specified in the APPEND statement, the target line defaults to the last line of the program currently in memory plus the specified increment (default:10).

If the target line for incoming statements does not already exist in the current program, the first incoming line is given a number equal to the number specified in the APPEND statement.

If the target line already exists in the current program, the first incoming line is given a number equal to the number specified in the APPEND statement plus the increment.

Appending Into an Existing Program

The 4041 uses the following procedure to append a program or program segment into the body of an existing program:

1. If the program does not contain a line with the same number as the target line the user specified, the first incoming line is given the number specified by the user for the target line.

If the program already contains a line with the same number as the target line, the first incoming line is given the number specified by the user plus one.

2. Successive incoming lines are inserted into the program, using an increment of one.
3. If there is room enough in the original program to fit all incoming lines, the "new" lines are renumbered after being inserted, according to the following procedure:

- a. If the increment specified by the user (default: 10) can be used without interfering with previous program lines, the new program lines are renumbered using that increment.
- b. If the increment specified by the user cannot be used without interfering with previous program lines, the new program lines are renumbered using an increment calculated by one of the following formulas:

- i. if the target line existed in the program before the APPEND operation:

$$\text{Increment} = \frac{(\text{Next line} - (\text{target line} + 1))}{(\text{number of incoming lines} + 1)}$$

- ii. if the target line did not exist in the main program before the APPEND operation:

$$\text{Increment} = \frac{(\text{Next line} - \text{target line})}{(\text{number of incoming lines} + 1)}$$

where "next line" is the number of the line that follows the target line number in the existing program.

4. If there is not sufficient room in the existing program to fit all incoming lines, an error is generated. All lines that fit between the target line and the next line with an increment of one remain in the program. The user should renumber later program lines to make more room for incoming lines, delete the lines that just came in, and try the APPEND again.

EXAMPLES:

Suppose the program currently in memory is as follows:

```
100  Rem existing program line #1
150  Rem existing program line #2
```

Entering the command

```
append "newfil",110
```

has the following effects:

1. if Newfil is four lines long, incoming lines from Newfil are brought in and re-numbered to 110, 120, 130, and 140;
2. if Newfil is less than four lines long, incoming lines from Newfil are brought in and renumbered, starting with line 110 and using an increment of 10;
3. if Newfil is more than four but less than forty lines long, all lines from Newfil are brought in and renumbered, starting with line 110 and using an increment calculated by the 4041;
4. if Newfil is more than forty lines long, forty lines from Newfil are brought in, starting with line 110 and numbered with an increment of one. An error is also generated.

```
3010  Append "dc510p"
```

This command adds a file "DC510P" from the DC-100 tape drive to the program currently in memory. The first incoming line is given a line number equal to the last line of the current program plus 10.

LOAD**The LOAD Statement**

Syntax Form: [line-no.] LOAD stexp

Descriptive Form: [line-no.] LOAD stream-spec

PURPOSE

The LOAD statement clears memory and loads a program from a driver.

EXPLANATION

If the file being loaded into memory is an ITEM file, the program is loaded directly into memory. If the file is an ASCII file, the 4041 must be equipped with Option 30 (Program Development ROMs) to translate the program line-by-line as it is being loaded.

EXAMPLES:

```
3100 Load "errors"
```

The program "Errors" is loaded from the DC-100 tape or disk.

The SAVE Statement

Syntax Form: [[line-no.] SAVE stexp [,num-exp [TO num-exp]]

Descriptive Form: [[line-no.] SAVE stream-spec [,first-line-to-be-saved [TO last- line-to-be-saved]]

PURPOSE

The SAVE statement transfers a copy of the current program to a driver. All or a portion of a program may be saved.

EXPLANATION

If a file is to be saved on the DC-100 tape or disk, the SAVE command must include a file specification. The **FORMAT** attribute of the file specification determines the format of the saved program (default is ASCII).

If no beginning or ending line references are given in the SAVE command, the entire program in memory is saved. If only a beginning line reference is given, the portion of the program from the specified line to the end is saved. If both beginning and ending line references are given, that portion of the program between the two lines (inclusive) is saved.

The SAVE command writes the program lines to the output device in the shortest form possible, shortening all keywords to four characters and eliminating all unnecessary spaces.

EXAMPLES:

```
5550       Save "prog5(ope=new,cli=yes)"
```

The program currently in memory is saved in ASCII format as file "Prog5" on the DC-100 tape or disk.

```
5560       Save "sub9(ope=new,cli=yes)"
```

Lines 4000 through 4999, inclusive, of the program currently in memory are saved in ITEM format as file "Sub9" on the DC-100 tape or disk.

Section 14

DC-100 TAPE

INTRODUCTION

This section discusses commands in 4041 BASIC that affect storage of programs or data on the DC-100 tape drive. A maximum of 48 files may be stored on the tape.

ASCII VS. ITEM FILES

Both programs and data may be written on DC-100 tapes in ASCII or ITEM format. The format is specified by the FORMAT parameter in the output stream spec.

4041 units not equipped with Option 30 (Program Development ROMs) can only load ITEM-format program files from tape. They can, however, read ASCII-format data from tape.

LOGICAL VS. PHYSICAL TAPE I/O

"Logical" tape I/O describes a method of accessing the DC-100 tape using a file name to specify the area of the tape the user wishes to read from or write to. INPUT and PRINT statements use logical tape I/O.

"Physical" tape I/O describes a method of accessing the DC-100 tape by dividing the tape into a series of physical records, then specifying a record to be read from or written to the tape. RBYTE and WBYTE statements use physical tape I/O.

TAPE STREAM SPECIFICATIONS

The driver name "TAPE" need not appear in stream specifications designating a file on the DC-100 tape. In addition, the VERify, DIRectory-update, and CLIp parameters may be grouped with the "file-side" parameters in the stream spec.

Example: The following two stream specs are equivalent:

- (1) "tape(cli=yes):file1(ope=new,cli=yes,siz=10000)"
- (2) "file1(cli=yes,ope=new,siz=10000)"

A complete list of TAPE parameters is included in Appendix D.

The DELETE FILE Statement

Syntax Form: [line-no.] DELETE FILE strexp[,strexp]...

Descriptive Form: [line-no.] DELETE FILE stream-spec[,stream-spec]...

PURPOSE

The DELETE FILE form of the DELETE statement deletes files from a file-structured device.

EXPLANATION

Attempting to delete a non-existent file has no effect.

When a file is deleted from the DC-100 tape, the records it occupied are merged with any surrounding free records into one contiguous free area. New files will occupy portions of these free areas.

If no driver is specified in the stream spec, the 4041 uses the driver "TAPE:" by default.

EXAMPLE

```
3090      Delete file "Tinker","Evers","Chance"
```

Three files gone, just like that!

The DIR Statement

Syntax Form: [line-no.] DIR [strex] [TO strex]
Descriptive Form: [line-no.] DIR [source-stream-spec][TO target-stream-spec]

PURPOSE

The DIR statement prints a directory of the DC-100 tape to a specified device.

EXPLANATION

The "short-form" directory consists of the volume ID, followed by a list of each file on the device, along with the file's type (ASCII or ITEM) and length in bytes. Each tape can contain up to 48 files. Free areas on the tape and the length in bytes of each free area are also shown.

The long-form directory gives complete information about each file on the device. The long form includes all of the short-form information, along with the file's starting record, length in records, and date and time created. The long form is obtained by entering a stream spec of "(LON = YES)" after the DIR keyword.

The date and time of creation for a file are only accurate if a SET TIME statement has been executed before the file was written. Otherwise, the date and time recorded represent the time since power-up, with date and time automatically set at power-up to "01-JAN-81 00:00:00".

The keyword "TO" followed by a stream spec after the DIR keyword sends the directory information to the device designated by the stream spec (default: system console).

EXAMPLE

DIR "(LON = YES)" TO "COMM:"

Sends a "long-form" directory of the current DC-100 tape to the standard RS-232-C interface port (see Example 14-1).

DIR TO "PRIN:"

Sends a "short-form" directory of the current DC-100 tape to the thermal printer on the front panel.

```

sorts      16-SEP-82 09:28:36 SOFT ERRORS = 0
FILE FILE  LENGTH START  NUMBER      LAST
NAME TYPE IN BYTES RECORD OF REC.  MODIFICATION DATE
-----
AUTOLD AS    510     5     2    24-SEP-82 11:29:00
INSERT AS   765     7     3    17-SEP-82 13:36:00
MERGE AS  1785    10     7    27-SEP-82 08:48:00
SHAKER AS  1530    17     6    17-SEP-82 14:12:00
HEAP AS   1020   23     4    17-SEP-82 14:21:00
BUBBLE AS   765    27     3    17-SEP-82 14:38:00
QUICK AS  1785    30     7    17-SEP-82 14:48:00
TSTIN AS   510    37     2    10-DEC-82 11:42:00
TSTOUT AS  510    39     2    10-DEC-82 11:53:00
SSORT AS   765    41     3    10-DEC-82 11:54:00
157335

```

Example 14-1.

DISMOUNT

The DISMOUNT Statement

**Syntax and
Descriptive Form:** [line-no.] DISMOUNT

PURPOSE

The DISMOUNT command is used to recover from certain kinds of mistakes without destroying tape files.

EXPLANATION

When the DISMOUNT command is executed, the 4041 checks the volume ID of the DC-100 tape in the drive.

If the tape in the drive is the same as the one whose directory is in the 4041's memory, the DISMOUNT command acts like a CLOSE ALL.

If the tape in the drive is not the same as the one whose directory is in the 4041's memory, or if no tape is in the drive, the DISMOUNT command simply returns the internal TAPE driver to its power-up state.

The EOF Function

Syntax Form: EOF (numexp)

Descriptive Form: EOF (logical-unit-number)

PURPOSE

The EOF function returns a value of 1 if an end-of-file condition is encountered on a specified logical unit, and returns 0 otherwise.

EXPLANATION

The logical unit given by the EOF function argument must be open.

Attempting to invoke the EOF function for a logical unit that has not been opened results in an error.

EXAMPLES

```
1500    If eof(1) then call eofsub
```

This command transfers control to a subprogram called "EOFSub" if the file specified by logical unit 1 is at end-of-file.

```
250     Eofval=eof(33)
```

The numeric variable EOFVal is assigned a value of 1 if logical unit 33 is at end-of-file, 0 otherwise.

FORMAT**The FORMAT Statement**

Syntax Form: [line-no.] FORMAT stexp

Descriptive Form: [line-no.] FORMAT volume-label

PURPOSE

The FORMAT statement prepares a DC-100 tape or disk for use, writing a volume label in the first record of the tape creating an empty directory or disk and filling the remainder of the tape or disk with empty records.

TAPE EXPLANATION

When a DC-100 tape is formatted, all information previously on the tape is lost. The tape is rewound, a volume label and the date and time of formatting are written onto the first record, an empty directory is created, and the remainder of the tape is filled with empty records.

The date and time of formatting are only accurate if the SET TIME statement has been executed before the FORMAT statement. If not, the date and time of formatting represent the time since power-up. The date and time are automatically set to "01-JAN-81 00:00:00" at power-up.

The volume label can contain up to 10 letters or numerals.

EXAMPLE

```
2500      Format "tapeone"
```

This command formats a tape and labels it "TapeOne".

DISK EXPLANATION

If the DC-100 tape is the system device and a disk requires formatting, the following command will format a disk:

```
Format "DISK(DEV=1,UNIT=0):", "Volume _ Name"
```

The first string is the stream specification for the disk and the second is the volume name associated with that disk.

If the disk is the system device the format command is the same as that for the tape, except the volume name may contain up to 12 characters.

The INPUT Statement (TAPE)

Syntax Form: [line-no.] INPUT [clause-list:]{var} [,var...]

where: clause-list =
 [numexp][ALTER strexp][BUFFER strvar][DELN strexp][DELS strexp][USING nemexp]
 [strex] [PROMPT strexp] [USING strexp]

Descriptive Form: [line-no.] INPUT [input clauses:][input list]

PURPOSE

The INPUT statement transfers data from a peripheral device into variables in the 4041's memory.

EXPLANATION

The file used for input from the DC-100 tape must be opened with an OPEN parameter of OLD. (OLD is the default.)

When used with the TAPE driver, the INPUT statement reads data from a file using the default characters for delimiting successive numeric and string data items (delimiters for numerics are space, tab, comma, semi-colon, colon, and the EOM character; delimiter for strings is EOM).

Reading Data from Logical Unit

If a pound sign (#) clause is followed by a logical unit number in an INPUT statement involving tape files, the 4041 keeps track of the read/write head's position as it "moves through" the file. Thus, successive INPUT statements read successive data elements from the file.

Reading Data from Files Named in Stream Specs

If a pound sign (#) clause is followed by a stream spec designating a tape file for INPUT, that file is automatically opened, read from, and closed during execution of the INPUT statement. The 4041 returns the read/write head to the beginning of the file each time the statement is executed.

See Section 8, *Input/Output*, for more information about the INPUT statement.

EXAMPLE

```
100   Open #1:"datafi"
110   Dim a1(5),a2(10)
120   Input #1:a1,a2,a3
```

A file called "DataFi" is opened (with OPEN parameter of OLD, by default) on the DC-100 tape. Line 110 dimensions two floating point arrays. Line 120 reads 5 values from DataFi to fill array A1 and the next 10 values to fill array A2. The next value read is stored in numeric variable A3. The next INPUT statement calling for a value from logical unit 1 will read the 17th element in the file.

OPEN**The OPEN Statement (TAPE)**

Syntax Form: [line-no.] OPEN #numexp:strexp[,strvar]

Descriptive Form: [line-no.] OPEN #lunum:stream-spec[,directory-entry]

PURPOSE

The OPEN statement associates a logical unit number with a stream spec for subsequent I/O operations. The stream spec defines a data path and specifies logical parameters for I/O operations through that logical unit.

EXPLANATION

If a stream spec does not include a driver spec, the driver "TAPE:" is used.

A complete listing of stream spec parameters that can be used with the TAPE driver and with tape files is included in Appendix D.

A string variable may be specified to receive a string containing directory information about the file at the time the file is opened. This directory information includes the file name, type (ASCII or ITEM), and length in bytes.

EXAMPLE

In Example 14-2, line 110 opens a new 10,000-byte-long file called "Datafi" on the DC-100 tape. The file will be "clipped" when closed, i.e., unused space will be removed from the end of the file. Line 110 also stores directory information about Datafi in the string variable Dirinf\$.

Line 120 causes the following message to be printed on the system console:

```
DATAFI AS      10200
```

This indicates that, when opened, file "Datafi" was an ASCII file 10,200 bytes long. (Note that the length specified in the OPEN statement was rounded up to the nearest multiple of 255.)

```
110   Open #1:"datafi(ope=new,siz=10000,cli=yes)",dirinf$
120   Print dirinf$
```

Example 14-2.

PRINT

This program puts the numbers 26 through 50 into "Datafi". The previous contents of the file are lost.

To add data to the end of this file, we assign the file an OPEN parameter of UPDATE, and write more data.

```

100   Open #1:"datafi(ope=upd,siz=5000)"
110   Integer i
120   For i=101 to 110
130     Print #1:i
140   Next i
150   Close all
160   Copy "datafi" to ask$("console")
170   End

```

The numbers 26 through 50 and 101 through 110 will appear on the system console device.

Delimiters

The 4041 writes an EOM character onto the tape after writing the last message unit designated by a PRINT statement, unless the last message unit in the PRINT statement is followed by a semicolon. The semicolon suppresses the writing of the EOM character after the last message unit.

The default EOM character for the tape is CARRIAGE-RETURN (ASCII 13).

Example

```

100   Open #1:"junk(ope=new)"
110   For i=1 to 3
120     Print #1:i
130   Next i

```

This sequence of statement prints the following information into file "Junk":

1< cr> 2< cr> 3< cr>

If line 140 were

```
120   Print #1:i;
```

the following information would be printed into the file:

123

(Semicolon at the end of the PRINT statement suppresses EOM.)

The 4041 writes EOU characters between message units and between elements of an array.

The default EOU character for the tape is SPACE (ASCII 32). The default EOH and EOA character for the tape is SPACE (ASCII 32).

Example:

Suppose lines 110 — 130 in the previous example were replaced by

```
110   Print #1:1,2,3
```

The following information would be written into file "Junk":

1< sp> 2< sp> 3< cr>

Example:

```

100   Open #1:"junk(ope=new,eoh=<65>,eoa=<66>)"
110   Integer num(4)
120   For i=1 to 4
130     Num(i)=i
140   Next i
150   Print #1:num(1);num(2);num(3);num(4)

```

Line 100 opens a new file called "Junk" and sets the EOH character to "A" and the EOA character to "B".

Line 150 prints the following information into file "Junk":

1A2B3B4< cr>

(First semicolon of the message unit causes the EOH character to be printed; second and succeeding semicolons cause the EOA character to be printed. Since the PRINT list does not end with a semicolon, the EOM character is printed the last element in the list.)

When writing arrays onto a tape file, the 4041 puts an EOA character between successive elements of the array, and an EOM character after the last element. In such cases, it is often convenient to open the logical unit such that the EOA and EOM characters are the same.

Example:

```
100   Open #1:"junk(ope=rep,cli=yes,boa=<13>)"
110   Integer array(100),i
120   For i=1 to 100
130     Array(i)=i
140   Next i
150   Print #1:array
```

Line 150 writes the following into file 'junk' on the DC-100 tape:

```
1<cr> 2<cr> 3<cr> 4<cr> 5<cr> 6<cr> 7<cr>
...100<cr>
```

When the user needs to read the data in again, the logical unit used to read them in should set the EOM character to <cr> (ASCII 13). The data can then be read in groups of any size (subject only to end-of-file limitations).

The RBYTE Statement (TAPE)

| | |
|--------------------------|--|
| Syntax Form: | [line-no.] RBYTE [#numexp:]{numexp,strar} [#strexp:] |
| Descriptive Form: | [line-no.] RBYTE [#lunum:]{physical-record-to-transfer,string- [#stream-variable-in-which-to-store-contents] spec:} |

PURPOSE

The RBYTE statement transfers 8-bit bytes from the GPIB, COMM, or FRTP drivers into the 4041's memory. The RBYTE statement transfers physical records from the DC-100 tape into the 4041's memory.

EXPLANATION

The 4041 uses the currently SELECTed stream spec as the default data path for RBYTE. The 4041 powers up with a default SELECTed stream spec of "GPIB0".

In order to execute an RBYTE command using a driver other than the standard GPIB interface, the RBYTE statement must either specify the logical unit or stream spec to be used, or a SELECT statement specifying a new default stream spec for RBYTE and WBYTE must be executed.

RBYTEs from the DC-100 tape drive read a string from a specified physical tape record.

To read data from the tape using RBYTE, the tape's PHYSICAL parameter must be set to a value of "YES". In addition, ALL TAPE FILES MUST BE CLOSED AT THE TIME OF THE RBYTE OPERATION.

The numeric expression must evaluate to an integer greater than or equal to 1 and less than or equal to the number of physical records on the tape. This integer is the physical record of the tape that will be read.

(To find the number of physical records on the tape: execute a DIR command; add up the total number of bytes on the tape; divide by 255; add 4.)

The string variable should be dimensioned to a length of 256 characters. If the string variable is dimensioned to less than 256 characters, characters after the current dimensioned size of the string are lost.

Reading physical records from the tape ignores all file boundaries.

EXAMPLE

```
100   Open #1:"tape(phy=yes):"  
110   Dim bigstr$ to 256  
120   Rbyte #1:10,bigstr$
```

Line 120 reads physical record 10 from the DC-100 tape and stores its contents in string variable Bigstr\$.

The RENAME Statement

Syntax Form: [line-no.] RENAME stexp TO stexp

Descriptive Form: [line-no.] RENAME old-stream-spec TO new-stream-spec

PURPOSE

The RENAME statement is used to rename a file on a file-structured device. The old file name is simply replaced by the new file name.

The two driver specs in the stream specs must match, or an error results.

If no driver spec is provided, the files are renamed using the driver "TAPE:".

EXPLANATION

The 4041 only allows one file on the tape to be renamed at a time. Attempting to rename a file that does not exist or to rename an existing file to the name of another existing file results in an error.

EXAMPLE

```
3500        Rename "this" to "that"
```

A file named "This" on the DC-100 tape is renamed to "That".

TYPE**The TYPE Function**

Syntax Form: TYPE (numexp)

Descriptive Form: TYPE (lunum)

PURPOSE

The TYPE function returns an integer from 0 through 4 indicating the type of data stored as the next data item in a file.

EXPLANATION

The TYPE function returns the following values, depending on the type of data stored in the next item of a file:

- 0 Empty File or File Not Open
- 1 End-of-File Character
- 2 Numeric Data or Character String Data, ASCII Format
- 3 Numeric Data, ITEM Format
- 4 Character String Data, ITEM Format

Data stored in ITEM format are stored using the internal data representation of the device the data were stored from.

The TYPE function is especially useful when working with files of ITEM data containing both numbers and character strings.

EXAMPLE

See Example 14-3.

```

990      Dattype=type(33)
1000     Goto dattype of 1100,1200,1300,1400,1500,1600,1700
1010     Print "empty file or file not open"
1100     Rem "end-of-file" character routine
1200     Rem ASCII character routine
1300     Rem ITEM short-floating-point data routine
1400     Rem ITEM string data routine
1500     Rem ITEM integer data routine
1600     Rem ITEM long-floating-point data routine
1700     Rem ITEM-format program routine

```

Example 14-3.

Section 15

SCSI Disk Sub-System

INTRODUCTION

This section lists examples and applications programs (software) and discusses commands in 4041 BASIC that affect storage of program or data on any magnetic disk device connected to the 4041 via the SCSI interface.

All of the commands used with the 4041 DC-100 Magnetic Tape device (refer to Section 14 of the 4041 Programmers Reference Manual) are compatible with the SCSI interface disk device driver. The SCSI interface may either be used with floppy disks, winchester hard disks, or both.

The stream specification for the SCSI interface contains all the parameters that the DC-100 Magnetic tape contains, with additional parameters that are required to specific disk information.

ASCII Vs. ITEM FILES

Both programs and data may be written on disk in ASCII or ITEM format. The format is specified by the FORMAT parameter in the output stream specification. 4041 units not equipped with Option 30 (Program Development ROMS) can only load ITEM-format program files from disk. They can however, read/write ASCII-format data from/to disk.

LOGICAL Vs. PHYSICAL DISK I/O

"Logical" disk I/O describes a method for accessing disks using a file name to specify the area of the disk the user wishes to read from or write to. INPUT and PRINT statements use logical disk I/O.

"Physical" disk I/O describes a method for accessing disks by sector number. RBYTE and WBYTE statements use physical disk I/O.

WILD CARDS AND OTHER SPECIAL CHARACTERS

Files are normally specified by filenames; however, "wild cards" may be used to name a file or group of files in an abbreviated fashion. Consider the following wildcard characters:

* ? []

When the DIR or DELeTe commands encounter these characters in a command word, they replace the word with a sorted list of matching filenames. The filenames come from your directory. Consider the following pattern matching (also called expansion):

- Most characters match themselves
- The ? matches any single character
- The * matches any string of characters
- The set of characters in [...] matches any one character in that set.

A pair of characters seperated by a dash [—] includes all the characters in the alphabetic or numeric range of the pair.

The following examples show wildcard characters used in commands:

| | |
|-------------------|---|
| DIR * | Lists all files in the directory. |
| DELETE FILE *.doc | Removes all files ending in ".doc". |
| DIR sec? | Directory of all files whose name begins with "sec" followed by any single character. |
| DIR Chap[1—3] | List the directories Chap1, Chap2, and Chap3. |

CAUTION

*The wild-card characters should be used carefully. For example, DELETE FILE * will delete all files in your directory.*

EXAMPLES

DIRECTORY Example:

| HARD_DISK File Name | File Type | Used Length | Unused Length | Start Rec # | Last Modification Date |
|------------------------|--------------|------------------|------------------|----------------|---------------------------|
| AUTOLD | AS | 147 | 365 | 17 | 01-JAN-81 00:02:40 |
| ASCIIFILE | AS | 51200 1323008 | 9216 | 18 | 01-JAN-81 02:57:00 |
| ITEMFILE | IT | 51200 7268352 | 1933312 | 2720 | 01-JAN-81 02:57:17 |

ASK\$ ("lu",n) Example:

```
DISKO (EDM=<13>, EDU=<32>, EQA=<0>, EDH=<0>): AUTOLD (OPE=OLD, SIZE=5.12000E+2,
ENT=256, FOR=ASC, CLI=NO, PHY=NO, VER=YES, DIR=NO, LON=YES, CTL=TEK, HAR=YES, ADR
=7, DEV=1, UNI=0, SEN=0, CYL=306, HEA=4, SEC=17. PAR=NO)
```

```

100 ! *****
110 ! ** FILE TO FILE COPY ROUTINE **
120 ! ** **
130 ! ** June 15, 1984 **
140 ! ** **
150 ! ** Copyright (c) 1984, Tektronix, Inc. All rights reserved. **
160 ! ** This software is provided on an "as is" basis without **
170 ! ** warranty of any kind. It is not supported. **
180 ! ** **
190 ! ** This software may be reproduced without prior permission, **
200 ! ** in whole or in part. Copies must include the above **
210 ! ** copyright and warranty notice. **
220 ! ** **
230 ! ** REQUIRED EQUIPMENT: **
240 ! ** 4041 OPT 3 **
250 ! ** 4925 (SCSI Dual Floppy Disk Drive) **
260 ! ** **
270 ! ** PURPOSE: **
280 ! ** To copy a file from any device to another device **
290 ! ** **
300 ! ** OPERATING PROCEDURE: **
310 ! ** Connect equipment as per 4041 manual. The system device **
320 ! ** will be set according to the operators inputs in the Sub **
330 ! ** program "sys_dev". The destination device will be set in **
340 ! ** the same way. **
350 ! *****
360 Init
370 Delete var all
380 !
390 ! Initialize program variables
400 !
410 Dim ans$ to 1
420 Init var sysdev$,cpydev$,sfil$,dfil$
430 Integer isfile
440 !
450 ! Error handler for destination file name that already exists
460 !
470 On error(850) then call isfil ! Tape error
480 On error(1204) then call isfil ! Disk error
490 !
500 ! Select source device for file copy
510 !
520 Sou: print "^J^JSelect source device."
530 Input prompt "^JEnter tape or disk (t or d) ":"ans$
540 If ans$(">t" and ans$(">d" then goto sou
550 Sysdev%=ans$
560 Call sys_dev(sysdev%)
570 !
580 ! Set system device to sysdev$
590 !
600 Set sysdev sysdev$
610 !
620 ! Select destination device for file copy
630 !
640 Des: print "^J^JSelect destination device."
650 Input prompt "^JEnter tape or disk (t or d) ":"ans$
660 If ans$(">t" and ans$(">d" then goto des

```

Fig. 15-1. Option 3 (SCSI) file-to-file copy routine.

SCSI Disk Sub-System

```

670     Cpydev$=ans$
680     Call sys_dev(cpydev$)
690     !
700     !   Dim sfil$ and dfil$ according to whether disk or tape
710     !
720     !   If pos(sysdev$,"TAPE",1) then dim sfil$ to 6 else dim sfil$ to 12
730     !   If pos(cpydev$,"TAPE",1) then dim dfil$ to 6 else dim dfil$ to 12
740     !
750     !   Directory prompt
760     !
770     !   Input prompt "^JPrint a directory of "&sysdev$&" (y or n):":ans$
780     !   If ans$="y" then dir sysdev$
790     !
800     !   Enter source file name
810     !
820     !   Input prompt "^JEnter source file name. ":sfil$
830     !
840     !   Directory prompt for destination file name
850     !
860     !   Input prompt "^JPrint a directory of "&cpydev$&" (y or n):":ans$
870     !   If ans$="y" then dir cpydev$
880     !
890     !   Enter destination device file name
900     !
910     !   Input prompt "^JEnter destination file name. ":dfil$
920     !
930     !   Actual copy process
940     !
950     !   Isfile=0 ! Clear error flag incase this is a retry
960     !   Copy sfil$ to cpydev$&dfil$&"(ope=new,cli=yes)"
970     !   If isfile then goto dfil
980     !
990     !   End of copy process
1000    !
1010    Endit:   close all
1020    Print  "^J^G^GCopy completed."
1030    End
1100    Sub sys_dev(var device$) local temp$,sysd,dev$,lu$,dev,lu
1110    Dim dev$ to 1,lu$ to 1
1120    If device$="d" then goto dev
1130    Device$="TAPE:"
1140    Return
1150    Dev:    ! This section for disk device number
1160    Input prompt "^JEnter the disk device # (0 thru 6) ":dev$
1170    If asc(dev$)>=48 and asc(dev$)<=54 then goto lu
1180    Print  "^JInvalid device number.....try again."
1190    Goto dev
1200    Lu:    ! This section for disk logical unit number
1210    Input prompt "^JEnter the disk logical unit # (0 or 1) ":lu$
1220    If asc(lu$)>=48 and asc(lu$)<=49 then goto ret
1230    Print  "^JInvalid logical unit number.....try again."
1240    Goto lu
1250    Ret:   device$="DISK(dev="&dev$&"),uni="&lu$&"):"
1260    Return
1270    End
1270    End
1300    Sub isfil ! Sub program error handler for file already exists
1310    Isfile=1 ! Set flag for error
1320    Advance
1330    End

```

Fig. 15-1. Option 3 (SCSI) file-to-file copy routine (cont.).

```

100 ! *****
110 ! **          TAPE TO DISK COPY ROUTINE          **
120 ! **                                          **
130 ! ** June 15, 1984                               **
140 ! **                                          **
150 ! ** Copyright (c) 1984, Tektronix, Inc. All rights reserved. **
160 ! ** This software is provided on an "as is" basis without      **
170 ! ** warranty of any kind. It is not supported.                **
180 ! **                                          **
190 ! ** This software may be reproduced without prior permission, **
200 ! ** in whole or in part. Copies must include the above       **
210 ! ** copyright and warranty notice.                        **
220 ! **                                          **
230 ! ** REQUIRED EQUIPMENT:                                     **
240 ! ** 4041      OPT 3                                       **
250 ! ** 4925 (SCSI Dual Floppy Disk Drive)                    **
260 ! **                                          **
270 ! ** PURPOSE:                                             **
280 ! ** To duplicate a tape on to a floppy disk                **
290 ! **                                          **
300 ! ** OPERATING PROCEDURE:                                  **
310 ! ** Connect equipment as per 4041 manual. The system device **
320 ! ** will be set to "TAPE:" and the destination device will be **
330 ! ** set according to the operators inputs in the Sub program **
340 ! ** "sys_dev". Tape file names will appear on disk with the **
350 ! ** form "TAPE_filnam" where filnam is the file name as it **
360 ! ** appears on the tape directory.                        **
370 ! *****
380   Init
390   Delete var all
400 !
410   Initialize program variables
420 !
430   Dim s$ to 256,p$ to 2816,vol$ to 6,ans$ to 1
440   Init var s$,p$,sysdev$,cpydev$
450   Integer i,r,n
460 !
470   Select destination device for duplication
480 !
490   Print "^J^JSelect destination device stream specs."
500   Call sys_dev(cpydev$)
510 !
520   Input prompt "^JInsert tape and destination disk <RETURN>:";ans$
530 !
540   Set system device to "TAPE:"
550 !
560   Sysdev$="TAPE:"
570   Set sysdev sysdev$
580 !
590   Prompt for formatting of new disk (not required)
600 !
610   Prompt:      input prompt "^JFormat the disk (y or n) ? ";ans$
620   If ans$="n" then goto rdir
630   Print "^J^JWARNING: Format will destroy all files on disk!!!"
640   Input prompt "^JIs this what you want to do? (y or n) ";ans$
650   If ans$="y" then goto vol else goto prmet
660   Vol:      vol$=ses$(ask$("volume"),1,6)
670   Format cpydev$,vol$

```

Fig. 15-2. Option 3 (SCSI) tape-to-disk copy routine.


```

680  !
690  ! call to sub program to read tape directory
700  !
710 Rdir:    call readir
720  !
730  ! Actual copy process
740  !
750    For i=1 to last step 16
760      Filnam#=trim$(ses$(p$,i,6))
770      Copy filnam$ to cpydev#&"TAPE_"&filnam#&"(ope=new,cli=yes)"
780      Next i
790  !
800  ! End of copy process
810  !
820 Endit:    close all
830    Print "^J^G^GCopy completed."
840    End
900 Sub sys_dev(var device$) local temp$,sysd,dev$,lu$,dev,lu
910    Dim dev$ to 1,lu$ to 1
920 Dev:    ! This section for disk device number
930    Input prompt "^JEnter the disk device # (0 thru 6) :":dev$
940    If asc(dev#)>=48 and asc(dev#)<=54 then goto lu
950    Print "^JInvalid device number.....try again."
960    Goto dev
970 Lu:    ! This section for disk logical unit number
980    Input prompt "^JEnter the disk logical unit # (0 or 1) :":lu$
990    If asc(lu#)>=48 and asc(lu#)<=49 then goto ret
1000    Print "^JInvalid logical unit number.....try again."
1010    Goto lu
1020 Ret:    device#="DISK(dev="&dev#&","uni="&lu#&"):"
1030    Return
1040    End
1100 Sub readir local ret,z$ ! Sub program to read tape directory
1110    Z#=chr$(0)&chr$(0)&chr$(0)&chr$(0)&chr$(0)&chr$(0)
1120    Open #1:"tape(phy=yes):"
1130    E=2
1140    P#=""
1150  !
1160  ! Read tape directory
1170  !
1180    For r=0 to 2
1190      Rbyte #1:r+e,r,s$
1200      P#=#P#&s$
1210    Next r
1220    Close 1
1230  !
1240  ! Check for last file on tape
1250  !
1260    For n=len(P#) to 16 step -16
1270      A#=ses$(P#,n-15,6)
1280      If s#(<)>z# then exit to ret
1290    Next n
1300 Ret:    last=n-15
1310    Return
1320    End

```

Fig. 15-2. Option 3 (SCSI) tape-to-disk copy routine (cont.).

```

100 | *****
110 | **          DISK TO DISK COPY ROUTINE          **
120 | **          FLOPPY TO FLOPPY                  **
130 | **          **                                  **
140 | ** June 15, 1984                               **
150 | **          **                                  **
160 | ** Copyright (c) 1984, Tektronix, Inc. All rights reserved. **
170 | ** This software is provided on an "as is" basis without **
180 | ** warranty of any kind. It is not supported.      **
190 | **          **                                  **
200 | ** This software may be reproduced without prior permission, **
210 | ** in whole or in part. Copies must include the above **
220 | ** copyright and warranty notice.                **
230 | **          **                                  **
240 | ** REQUIRED EQUIPMENT:                          **
250 | ** 4041 OPT 3                                   **
260 | ** 4925 (SCSI Dual Floppy Disk Drive)          **
270 | **          **                                  **
280 | ** PURPOSE:                                     **
290 | ** To duplicate one floppy disk to another floppy disk **
300 | **          **                                  **
310 | ** OPERATING PROCEDURE:                        **
320 | ** Connect equipment as per 4041 manual. The system device **
330 | ** will be set according to the operators inputs in the Sub **
340 | ** program "sys_dev".                          **
350 | *****
360 | Init
370 | Delete var all
380 |
390 |   Initialize program variables
400 |
410 |   Dim a$ to 512,ans$ to 1,vol$ to 12
420 |   Init var a$,donefl$,sysdev$,cpydev$
430 |   Integer i
440 |
450 |   Error handler for end of disk error
460 |
470 |   On error(1250) then call donecpy
480 |
490 |   Select system device (sysdev)
500 |
510 |   Print "^J^JSelect source device stream specs."
520 |   Call sys_dev(sysdev$)
530 |
540 |   Select destination device for duplication
550 |
560 |   Print "^J^JSelect destination device stream specs."
570 |   Call sys_dev(cpydev$)
580 |
590 |   Input prompt "Insert source and destination disks <RETURN>:" :ans$
600 |
610 |   Set system device per operators entries
620 |
630 |   Set sysdev sysdev$
640 |
650 |   Prompt for formatting of new disk (not required)
660 |
670 |   Input prompt "^JFormat the new disk <y or n> ? " :ans$

```

Fig. 15-3. Option 3 (SCSI) disk-to disk copy routine.

SCSI Disk Sub-System

```

680     If ans$="n" then goto rep
690     Vol$=ses$(ask$("volume"),1,12)
700     Format cpydev$,vol$
710     !
720     ! Insert in stream spec "phy=yes" for rbyte and wbyte commands
730     !
740 Rep:     rep$(sysdev$,pos(sysdev$,""),1,0)=",phy=yes"
750     Rep$(cpydev$,pos(cpydev$,""),1,0)=",phy=yes"
760     !
770     ! Open logical units for copy sequence
780     !
790     Open #1:sysdev$
800     Open #2:cpydev$
810     !
820     ! Actual copy process
830     !
840     For i=1 to 700
850         Rbyte #1:i,a$
860         If donefls then exit to endit ! Fls to indicate end of disk
870         Wbyte #2:i,a$
880         Next i
890     !
900     ! End of copy process
910     !
920 Endit:   close all
930     Print "^J^G^GCopy completed."
940     End
1000 Sub donecpy ! Sub program to set end of disk error fls
1010 Donefls=1
1020 Advance
1030 End
1100 Sub sys_dev(var device$) local temp$,sysd,dev$,lu$,dev,lu
1110     Dim dev$ to 1,lu$ to 1
1120 Dev:    ! This section for disk device number
1130     Input promet "^JEnter the disk device # <0 thru 6> ":"dev$
1140     If asc(dev$)>=48 and asc(dev$)<=54 then goto lu
1150     Print "^JInvalid device number.....try again."
1160     Goto dev
1170 Lu:    ! This section for disk logical unit number
1180     Input promet "^JEnter the disk logical unit # <0 or 1> ":"lu$
1190     If asc(lu$)>=48 and asc(lu$)<=49 then goto ret
1200     Print "^JInvalid logical unit number.....try again."
1210     Goto lu
1220 Ret:    device$="DISK(dev="&dev$&","uni="&lu$&"):"
1230     Return
1240     End

```

Fig. 15-3. Option 3 (SCSI) disk-to-disk copy routine (cont.).

STREAM SPECIFICATION SETTINGS FOR SCSI DISK INTERFACE

| Driver Parameters | Meaning |
|---|---|
| ADR = [n] | [n] indicates the switch settings for the SCSI device address of the 4041. The number range is 0-7. When the SCSI system device switch is set on, this number range is 80-87, with the device number being the low order digit and the system device switch setting indicated in the high order digit. |
| CLIp = YES,NO | If YES, the remaining free space is clipped from the end of a file when closed. The default is NO. |
| CTL = xxx | This parameter simply displays the type of disk controller being used. The displayed values are: <ul style="list-style-type: none"> TEK - The disk controller is a 4925 floppy disk controller or a modified 4926 hard disk controller. ADP - The disk controller is an Adaptec disk controller (Model 4000 or 5500). DTC - The disk controller is a Data Technology Corp. disk controller (Model 520A or 520B). |
| CYLinders = [n] | Specifies the number of cylinders on the disk device. ¹ This parameter only has meaning at format time. |
| DEvice = [n] | The SCSI disk device number being accessed. If none is specified, the device number from the rear panel is used. |
| NOTE | |
| When using multiple disks attached to the SCSI interface, the device number defaults to the rear panel system device unit switch, setting and not the specific system device setting. | |
| DIRectory = YES,NO | Has no affect; used for DC-100 magnetic tape compatibility only. For example, if program used magnetic tape but now uses this parameter with the disk; no error occurs. |
| ENTries = [n] | Specifies the number of directory entries to be allocated on the disk at format time. Default is 256 entries. Each disk sector (512 bytes) can contain 16 directory entries. When an entry is specified, that number is rounded up to the next whole sector. For example, if 1603 was specified, the actual number of entries would be 1616. |
| EOA = [n] | End-of-Argument character. Default is <0>. |
| EOH = [n] | End-of-Header character. Default is <0>. |
| EOM = [n] | End-of-Message character. Default is <13>. |
| EOU = [n] | End-of-Message-Unit character. Default is <32>. |
| FORmat = ASCii,ITEm | ASCii-data stored on disk as ASCII characters. Default is ASCii. ITEm-data stored on disk in ITEM format (4041 internal representation). |
| HARd = YES,NO | Specifies disk type being accessed. Setting at OPEN time has no affect. |

SCSI Disk Sub-System

Stream Specification Settings for SCSI Disk Interface (cont.)

| Driver Parameters | Meaning |
|-----------------------------------|---|
| HEAdS={n} | Specifies the number of disk heads. ¹ This parameter only has meaning at format time. |
| LONG=YES,NO | Default is YES and only has meaning at format time and specifies if the disk device is to be physically formatted. If NO, the volume header and empty directory are recorded on the disk. |
| OPEn=NEW, OLD, REPlace, UPDate | NEW - opens a new file for writing; error if file already exists. OLD - opens existing file and positions it at beginning of file; error if file non-existing. REPlace - write only; deletes existing file. Opens and positions new file at beginning of file. UPDate - write only. Opens and positions file at end of file. Error occurs if file is non-existing. |
| PARity= YES,NO | If YES, this parameter specifies that the disk controller (being accessed) has the parity option set. Errors result if this parameter is not set to reflect the disk controllers actual setting. NOTE: Since the default is NO, a disk controller set for parity cannot be used as the default system boot device. |
| PHYSical= YES,NO | If YES, the disk is in physical mode (I/O via Rbyte, Wbyte). Other logical units can open to same disk in logical file mode with no error generated. |
| SECTors={n} | Specifies number of sectors per track (cylinder) on a disk device. ¹ This parameter only has meaning at format time. |
| SENse={n} | Used to report disk sensed error. Has no meaning if specified during OPEN time. Will not cause an error. |
| SIZE={nn} | Maximum size of file (bytes). |
| UNIT={n} | SCSI system device unit number address that overrides rear panel SCSI switch setting. Defaults to SCSI switch setting, if not specified. If parameter is incorrect, an error can occur. |
| VERify= YES,NO | If YES (default), Read after Write is performed and all data bytes are verified. If No, there is no Read after Write performed. |

¹The CYLinder, HEAdS, and SECTor settings are only used at format time to specify a larger or smaller disk device than the default 10 Mbyte full height hard disk or the 320 kbyte floppy disk. This information is sent to a disk controller at format time to specify the capacity of the connected disk drive.

Section 16

UTILITY AND ARRAY HANDLING ROMCALLS

INTRODUCTION

Utility and array handling romcalls are part of an internally mounted ROM pack ("UTL2") that gives the 4041 added capabilities for high-speed binary block transfers, array segmentation, array scaling, and histograms.

Romcalls supplied by UTL2 are briefly described below:

ARRSEG— copies a segment of one array, starting with a specified point and ending with a specified point, into a new array. ARRSEG provides a convenient and fast way to extract a segment of a larger array for processing or display.

GETARR— reads a binary data block from an instrument on the IEEE-488 bus into an integer array. The block must conform to Tektronix Standard Codes and Formats. An input parameter specifies the number of significant bits per array value, and checksums are automatically tested.

GETSCL— reads a binary data block from an instrument on the IEEE-488 bus into an array, much the same as the GETARR romcall does. As with GETARR, the block must conform to Tektronix

Standard Codes and Formats. This romcall differs in that it also applies specified zero-reference and scaling values to the array, so that the result is a scaled floating-point array.

ARRSCL— performs the same array scaling as GETSCL, except the scaling is applied to an existing data array in 4041 memory. No new data are transferred from an instrument.

HISTOG— generates the histogram of a source array. The histogram shows how often each value in a source array occurs. A histogram is a simple way to find the most common values in a source array—such as the top and base of a pulse waveform.

The routines can be called using the RCALL statement in 4041 BASIC or by simply inserting the romcall name with the required arguments into the program.

The ARRSEG Romcall

| | |
|--------------------------|--|
| Syntax Form: | [line no.] ARRSEG numarray, numexpr, numexpr, numarray |
| Descriptive Form: | [line no.] ARRSEG source array, begin pt., end pt., destination array |

PURPOSE

The ARRSEG romcall will zone the source array into a new destination array. This routine is particularly useful for segmenting an array from other than the first point.

EXPLANATION

The second and third arguments specify the segment of the source array that is transferred to the destination array. The destination array must be of the same type as the source array and must be large enough to receive the range of data.

If used in a subprogram, ARRSEG must reference global data, or data passed by reference only (not by value), or local data created within the subprogram.

EXAMPLE

```
100 Dim wfmdata (1024) ,wfmdat1 (512)
110 Open #1:"gpib (pri=11):" ! Setup
    7D20T for lun #1
120 Print #1:"curve?" ! Tell 7D20T to
    send array of data
130 GETSCL 1,wfmdata,0.005,128,8 ! Go
    get scaled data
140 Begin=513
150 Ending=1024
160 ARRSEG wfmdata,begin,ending,wfmdat1
```

Line 160 segments array "wfmdata" from location 513 to 1024 into array "wfmdat1".

NOTE

The value passed for the first point in the destination array must be at least a value of 1, otherwise the destination array will be filled with zeros.

The GETARR Romcall

Syntax Form: [line no.] GETARR numexpr, numarray [,numexpr]

Descriptive Form: [line no.] GETARR lun, integer array [,# of bits]

PURPOSE

The GETARR romcall is used to transfer a binary block of data into the 4041 from an instrument on the IEEE-488 bus.

EXPLANATION

The GETARR romcall brings a binary block data transfer directly into an integer array, greatly increasing the speed with which the 4041 can acquire an array. The instrument specified in the first argument must have been previously told to prepare to send the data, and the integer array must have been previously dimensioned. If the optional number of bits argument is not used, the default number of bits (of resolution) is 16.

EXAMPLE

```
100 Integer wfm (1024)
110 Open #1:"gpib (pri=11):" ! setup
    7D20T for lun #1
120 Print #1:"curve?" ! Tell 7D20T to
    send array of data
130 GETARR 1,wfm,8 ! Bring the data into
    the integer array
```

Line 130 will bring in a binary block from lun #1 and store it in array "wfm" with 8 bits per value.

The GETSCL Romcall

Syntax Form: [line no.] **GETSCL** numexpr, numarray, numexpr, numexpr, numexpr

Descriptive Form: [line no.] **GETSCL** lun, real array, volts/bit value, zero reference value, # of bits

PURPOSE

The GETSCL romcall works much like the GETARR romcall, but also provides scaled data in the floating point (real) target array. This romcall can be used in place of GETARR if the data must be in volts/bit resolution, such as for pulse parameter analysis.

EXPLANATION

The GETSCL romcall will transfer a binary block of data into the 4041 from the specified instrument, then subtract the zero-reference value and multiply by the bits/volt value. The zero reference value is applied first (to find the baseline in bit levels), so this value must be in vertical bit levels.

EXAMPLE

```
100 Dim wfmdata (1024)
110 Open #1:"gpib (pri=11):" ! Setup
    7D20T for lun #1
120 Print #1:"curve?" ! Tell 7D20T to
    send array of data
130 GETSCL 1,wfmdata,0.005,128,8 !Go get
    scaled data
```

Line 130 will bring in from lun #1 a binary block transfer (8 bits per value), subtract 128 from each value, multiply them by .005, and store the results in the array "wfmdata".

The ARRSCL Romcall

| | |
|--------------------------|--|
| Syntax Form: | [line no.] ARRSCL numarray, numexpr, numexpr, numarray |
| Descriptive Form: | [line no.] ARRSCL source integer array, volts/bit value, zero reference value, destination floating point array |

PURPOSE

The ARRSCL romcall takes an integer data array and applies scaling values to the destination floating point array like the GETSCL romcall does. Unlike GETSCL, this routine takes an existing integer array and scales it, instead of acquiring the data over the IEEE-488 bus. It is useful for scaling an integer array brought into memory from mass storage or from an instrument with the GETARR romcall.

EXPLANATION

The source integer array is scaled using the second and third arguments, then the scaled data are stored in the destination floating point array. The destination array must have been previously dimensioned to at least as large as the integer source array.

EXAMPLE

```

100 Integer wfm (1024)
105 Dim wfmdata (1024)
110 Open #1:"gpib (pri=11):" ! Setup
    7D20T for lun #1
120 Print #1:"curve?" ! Tell 7D20T to
    send array of data
130 GETARR 1,wfm,8 ! Bring the data into
    an integer array
140 ARRSCL wfm,0.005,128,wfmdata

```

Line 140 takes the integer array "wfm" and performs the same scaling process of GETSCL romcall. The result is stored in the floating point array "wfmdata". This routine would most likely be used if the original integer data were needed elsewhere or if the integer source array was saved. Using GETARR instead of GETSCL speeds up the transfer of data, so ARRSCL is useful when processing can be done later.

The HISTOG Romcall

Syntax Form (Integer Source Array):^{1, 2}

[line no.] HISTOG integer array, integer array, integer scalar

Syntax Form (Real Source Array):^{1, 3}

[line no.] HISTOG real array, integer array, real scalar, real scalar, integer scalar

Descriptive Form (Integer Source Array):

[line no.] HISTOG source, destination, minimum value

Descriptive Form (Real Source Array):

[line no.] HISTOG source, destination, minimum value, maximum value, number of slots

¹ The calling syntax is different for integer source arrays and real source arrays.

² If the source array is integer, the values must not be negative.

³ If the source array is real, only short floating point numbers are supported (not long floating point).

PURPOSE

The HISTOG romcall is used to find the number of occurrences of a value, or range of values, in a waveform.

EXPLANATION

The algorithm for this romcall computes the number of occurrences of particular values in the source array (real or integer) and returns the number of each value's occurrence in the destination array. The routine will work much faster on integer source arrays than on real source arrays. Regardless of the source array type, the destination array must always be integer.

EXAMPLES

The following sample program shows how to determine waveform base and top values with a histogram. Examples 1a, 1b, and 1c show three different methods of establishing the number of slots to sort. In these three examples, the base value is assumed to be in the lower half of waveform data values, while the top value is assumed to be in the upper half of waveform data values.

In Example 1a, the digitizer's resolution is used to determine the number of slots to sort. The source array for this example is an integer array.

Example 1b again has an integer array as its source, and uses the span of values in this array to determine the number of slots to sort.

Example 1c operates on a scaled waveform, the source array being a short floating point array output from the ARRSCAL romcall. This example uses the digitizer's "volts per bit" to determine the number of slots to sort.

Example 1

```

2000 Integer I_wfm (1024), num_bit, slots,
      I_min, I_max, PL, PL1
2110 Call get_wfm (scope, ch_num, I_wfm,
      Y_mult, Y_zero, num_bit, volt_bit)
      ! Get a waveform from a digitizer
2120 ! scope = digitizer IEEE-488 bus
      address
2130 ! ch_num = which channel of digi-
      tizer to get data from
2140 ! I_wfm = integer array containing
      binary wfm values
2150 ! Y_mult = Y value multiplier - re-
      lationship between volts/div and
      number of discrete levels/div.
2160 ! Y_zero = Y value zero reference -
      value that represents 0 volts on
      screen
2170 ! num_bit = vertical resolution of
      digitizer expressed as the power
      to which to raise 2 to (e.g., 8, 9,
      10)
2180 ! volt_bit = the quantity a change
      of one digitized level represents
2190 !

```

In the preceding sample program segment, line 2100 gets the waveform data from a digitizer.

Example 1a

```

2200 Rem *** integer HISTOG type 1 ***
2210 Slots=2num_bit ! Number of possi-
      ble digitized levels
2220 Integer hist_arr (slots), top, base,
      top_arr (slots/2), base_arr
      (slots/2)
2230 I_min=0
2240 HISTOG I_wfm, hist_arr, I_min, !
      Histogram of data "UTL2"
2250 ARRSEG hist_arr, 1, slots/2, base_arr
      ! "UTL2"
2260 ARRSEG hist_arr, slots/2+1, base_arr
      ! "UTL2"
2270 ! Determine base value from histo-
      gram-modal value
2280 AMAX base_arr, I_max, PL ! 4041R03 or
      Option 10
2290 Base=PL ! Since arrays start at 1
      and digitizers start at 0
2300 ! Determine top value from
      histogram-modal value
2310 AMAX Top=PL1+slots/2-1 ! Since ar-
      rays start at 1 and digitizers
      start at 0
2300 !

```

In example 1a, line 2210 sets up the the number of sorting slots for the histogram based on the digital resolution of the digitizer.

Line 2220 creates arrays for the histogram output and for lower and upper halves of the histogram output.

Line 2240 performs a histogram on the array "I_wfm"; the output array is "hist_arr".

Lines 2250 and 2260 put the lower and upper halves of the histogram data into the arrays "base_arr" and "top_arr", respectively.

Lines 2280 and 2290 determine the waveform base level as the largest data point in the lower half of the histogram output.

Lines 2310 and 2320 determine the waveform top level as the largest data point in the upper half of the histogram output.

Example 1b

```

2340 Rem *** integer HISTOG type 2 ***
2350 AMIN I_wfm, I_min, PL ! 4041R03 or
      Option 10
2360 AMAX I_wfm, I_max, PL1 ! 4041R03 or
      Option 10
2370 Slots=I_max-I_min+1
2380 Integer hist_arr (slots, top_arr (int
      (slots/2)), base_arr (round
      (slots/2))
2390 HISTOG I_wfm, hist_arr, I_min ! His-
      togram of data "UTL2"
2400 ARRSEG hist_arr, 1, round (slots/2),
      base_arr ! "UTL2"
2410 ARRSEG hist_arr, round (slots/2)+1,
      slots, top_arr ! "UTL2"
2420 ! Determine base value from
      histogram-modal value
2430 AMAX base_arr, I_max, PL ! 4041R03 or
      Option 10
2440 Base=PL+I_min-1 ! Arrays start at 1
      and digitizers start at 0
2450 ! Determine top value from
      histogram-modal value
2460 AMAX top_arr, I_max, PL1 ! 4041R03 or
      Option 10
2470 Top=PL1+round (slots/2)+I_min-1 !
      Arrays start at 1 and digitizers
      start at 0
2480 !

```

UTL2

HISTOG

In example 1b, lines 2350 and 2360 find the minimum and maximum data values of the integer source array, respectively.

Line 2370 calculates the number of slots based on the span of values in the source array.

Line 2380 defines the data arrays for histogram output data.

Line 2390 performs the histogram, putting the output into the array "hist_arr".

Lines 2400 and 2410 put the lower and upper halves of the histogram data values into the arrays "base_arr" and "top_arr", respectively.

Lines 2430 and 2440 determine the waveform base level as the largest data point in the array "base_arr".

Lines 2460 and 2470 determine the waveform top level as the largest data point in the array "top_arr".

Example 1c

```
2490 Rem *** real HISTOG ***
2500 Dim wfm_act (1024)
2510 ARRSCl I_wfm,volt_bit,Y_zero,
      wfm_act ! "UTL2"
2520 AMIN wfm_act,R_min,PL ! 4041R03 or
      Option 10
2530 AMAX wfm_act,R_max,PL1 ! 4041R03 or
      Option 10
2540 Slots= (R_max-R_min)/volt_bit-1
2550 Integer hist_arr (slots+1),top_arr
      (int ((slots+1)/2)),base_arr
      (round ((slots+1)/2))
2560 HISTOG wfm_act,hist_arr,R_min,
      R_max,slots ! Histogram of data
      "UTL2"
2570 ARRSEG hist_arr,1,round
      ((slots+1)/2),base_arr ! "UTL2"
2580 ARRSEG hist_arr,round
      ((slots+1)/2)+1,slots+1,top_arr
      ! "UTL2"
2590 ! Determine base value from histo-
      gram-modal value
2600 AMAX base_arr,I_max,PL ! 4041R03 or
      Option 10
2610 Base= (PL-1)*volt_bit+R_min
2620 ! Determine top value from histo-
      gram-modal value
2630 AMAX top_arr,I_max,PL1 ! 4041R03 or
      Option 10
2640 Top= (PL1+round (slots+1)/2)
      *volt_bit+R_min
```

In example 1c, line 2500 defines real array "wfm_act" with the same number of elements as integer array "I_wfm".

Line 2510 produces the scaled array "wfm_act" using the digitizer "volts per bit" and vertical offset ("Y_zero").

Lines 2520 and 2530 get the minimum and maximum values of the waveform, respectively.

Line 2540 calculates the number of slots (bits), using the span of voltage values divided by the "volts per bit".

Line 2550 defines data arrays for the histogram output data.

Line 2560 performs a histogram bounded by the minimum and maximum voltage values of the source array, using a calculated number of slots; the output is placed in the array "hist_arr".

NOTE— the actual number of slots could be different; more slots provide better resolution and immunity to arithmetic round-off errors (4*slots is recommended as optimum).

Lines 2570 and 2580 put the lower half of the histogram data into the array "base_arr" and the upper half of the histogram data into the array "top_arr".

Lines 2600 and 2610 calculate the base value as the maximum value of array "base_arr" multiplied by "volts per bit" plus the minimum waveform voltage value.

Lines 2630 and 2640 calculate the top value as the maximum value of array "top_arr" multiplied by "volts per bit" plus the minimum waveform voltage value.

Appendix A

ERROR MESSAGES

Group I (1-19) Translation Errors. These cannot be handled by the user.

- 1 Syntax error. User visible as "Syntax Error".
- 3 Subprogram syntax error. A SUB/FUNCTION statement was entered that did not follow an END statement, or a statement was entered following an END statement that was not a SUB/FUNCTION statement.
- 4 END statement error. An END statement was entered in a program segment that already has an END statement, or that has lines numbered greater than that of the END statement.
- 5 Label error. A label name was entered that is already in use as either a label or a subprogram/function name.
- 6 Subprogram name error. A subprogram/function name was entered that is already in use as a subprogram/function name.
- 7 Parameter label error. A parameter name was used as a label.
- 8 Symbol table full. The main symbol table, with room for 1024 entries, is full. The program in memory must be saved, and a DELETE ALL done to remove deleted entries from the symbol table.
- 9 Local label error on subprogram edit.
- 10 Statement too complex.
- 11 Too many tokens in statement.
- 12 Edit of active subprogram statement

Group II (20-49). Other errors that cannot be handled by the user.

- 20 Unable to continue. There is no next valid line to continue with, or the program is in a state where continuation is invalid.
- 21 Subprogram not found. The subprogram name specified in a CONNECT statement does not exist.
- 22 Illegal variable trace. An argument to a TRACE VAR was not a variable, or an argument to a TRACE SUB was not a subprogram/function name.
- 23 Illegal syntax for RENUMBER. The specification of the operands for a RENUMBER was not valid.
- 24 Line limit reached in RENUMBER. There is not sufficient space in the program to renumber without moving existing lines which are outside the renumbered segment.
- 25 Break point on non-executable statement. Break-points may not be set on SUB, FUNC, REM, DATA or IMAGE statements.
- 26 Function in immediate mode. User functions may not be invoked in immediate mode.
- 27 Program statement in immediate mode OR attempt to enter immediate-mode statement into program.
- 28 Immediate mode statement during load or append.
- 29 Line cannot be listed. A RENUMBER operation has made the line too long to be listed. The line can still be executed, however.
- 40 Run time stack full. The internal stack that manages CALLs, GOSUBs, FORs, handler and user- function invocations does not have room for any more entries.
- 41 BRANCH not into active program segment. The target of a BRANCH statement must be a line within a program segment in the active call sequence.

ERROR MESSAGES

Group III (50-69) Common (Utility Routine) Errors.

- 50 Number expected, string found. In the course of evaluating an expression, an operand of improper type was encountered.
- 51 Invalid logical unit/error number. An error number or key or logical unit number specified for an ON/OFF or ENABLE/DISABLE statement is not valid.
- 52 Undefined variable. An identifier was referenced that has no current value.
- 53 Zero or minus subscript. An array subscript did not evaluate to a positive value.
- 54 Scalar variable subscript. A non-array variable was referenced with a subscript.
- 55 Wrong number of subscripts. An array variable was referenced with the wrong number of subscripts. Invalid combination of operands. An array name was assigned to a scalar.
- 56 Type mismatch. An attempt was made to assign an entire array to a scalar variable, or a string value to a numeric variable, or a numeric value to a string variable. Also given for invalid array-to-array assignment by a rompack (RCALL).
- 57 Numeric expected, something else found. A statement was expecting a numeric argument, and a string or other value was encountered.
- 58 Array reference without required subscripts. An array variable was referenced in its entirety in a context where only an array element reference is valid.
- 60 Different size arrays specified in implied operation. An array-array operation was attempted on arrays that are dimensioned to different sizes.
- 61 String expected but not found. A statement expected a string argument, but encountered a non-string value.
- 62 Subscript exceeds dimensioned value.
- 63 Value or variable expected. Something else, such as a function reference, or the result of an implied array operation, was found.
- 64 String truncated on assignment or READ operation.

Group IV (70-79) Program Structure Errors.

- 70 Incomplete subprogram. An attempt was made to enter a subprogram that does not have an END statement.
- 71 Invalid line number. An attempt was made to execute a line that does not exist, or is not valid for execution.
- 72 Label error. An attempt was made to assign a value to a label.
- 73 PD roms not available. A requested action required the program development option, which was not available.
- 74 Invalid use of subprogram name. A subprogram name was used where a string or numeric value is required.
- 75 Main program incomplete. A program has subprograms, and there is no END statement in the main program.

Group V (80-99) Mathematical Errors.

- 80 Overflow. This error may also be generated during internal conversions for some statements such as DIM.
- 81 Underflow.
- 82 Divide by zero.
- 83 Logarithm error. Attempt to take log of a number less than or equal to zero.
- 84 Exponential overflow.
- 85 Trigonometric range error. An argument to an inverse trig function is out of the valid range (e.g., ASIN(2)).
- 86 Exponentiation error. X^Y where $X < 0$, Y non-integer or 0^0 .
- 87 Square root of negative number.
- 88 Tangent overflow.
- 89 Integer overflow.

- 90 SUM function error. The argument passed to the SUM function was invalid.
- 91 ASK/ASK\$ function error. The number, order, type or value of the arguments passed to an ASK or ASK\$ function were invalid.

Group VI (100-109) String Function Errors.

- 100 Error in ASC function. A null string was passed as an argument to the function.
- 101 Error in VAL function. No number was found in the string argument passed to a VAL or VALC.
- 102 REP\$ result length error. The call to REP\$ resulted in a string longer than 32767 chars.
- 103 Concatenation error. The concatenation of two strings resulted in a string longer than 32767 characters.

Group VII (110-119) Memory Errors.

- 110 Memory full. It is still possible to perform certain tasks, such as DELETES and SAVES, in order to free up memory.
- 111 String allocation failure. There is not enough free memory to allocate a string variable or temporary.
- 112 Insufficient memory to complete a SAVE or LIST operation.
- 113 Data element overflowed input buffer OR illegal clause used in proceed-mode INPUT statement OR illegal element in proceed-mode INPUT list.
- 114 Memory fragmented.

Group VIII (120-139) User-defined Function Invocation Errors.

- 124 Function assignment error. No value was assigned to the function name by a function subprogram.
- 125 Scalar function error. At the beginning of execution of a statement, an identifier was a variable, but before the end of execution of the statement a user function has been entered with that name.
- 126 Invalid function use. A function name is not valid in the current statement.

Group IX (140-159) User Handler Activation and Return.

- 140 Interrupt error. There is no user handler for an enabled interrupt.
- 141 GOSUB handler existence error. The destination specified by a GOSUB handler does not exist at the time of activation.
- 142 GOSUB handler segment error. The destination specified by a GOSUB handler is not in the current subprogram segment.
- 143 CALL handler in use error. The subprogram specified by a CALL handler is already active in the call sequence.
- 144 CALL handler complete error. The subprogram specified by a CALL handler does not have an END statement.
- 145 CALL handler defined error. The subprogram specified by a CALL handler is not defined.
- 146 CALL handler type error. The name specified for a CALL handler is not a subprogram name.
- 147 Failure on return. It was not possible to return in a valid manner after execution of a handler.
- 148 Improper return. The type of return is not appropriate for the handler or subprogram/ function being returned from.

ERROR MESSAGES

Group X (200-579) Basic Statement Errors.

Each subgroup is preceded by a universal error code that matches any error arising from execution of that BASIC statement, but is not in itself an actual error code.

- * 200 Statement error code for APPEND
- 201 APPEND line limit reached. It is no longer possible to append any more lines without moving existing lines. Lines already appended remain.
- 202 APPEND file format in error. The file format or type of file is not valid for an append.
- * 210 Statement error code for CALL
- 211 Subprogram in use. The subprogram or user function is already active in the call sequence.
- 212 Expression passed to variable parameter. An expression (rather than a variable) was passed as an argument to a variable (reference) parameter.
- 213 CALL name not subprogram. The name called by a subprogram or user function is not a valid subprogram/function name.
- 214 CALL name not defined. The name called by a subprogram or user function is not currently defined.
- 215 Subprogram incomplete. The subprogram or user function that was called does not have an END statement.
- 216 Parameter not defined. An argument passed to a subprogram or user function does not currently have a value.
- 217 Subprogram passed as value parameter. The name of a subprogram or user function was passed to a value parameter.
- 218 Wrong number of arguments. The number of arguments in a call to a subprogram or user function do not match the number of parameters defined for that subprogram.
- 219 Parameter types don't match. The type of an argument passed to a subprogram or user function does not match the type of the parameter passed to.
- * 220 Statement error code for CLOSE
- * 230 Statement error code for COMPRESS
- * 240 Statement error code for COPY
- * 250 Statement error code for DELETE
- 251 Deleted subprogram active. A subprogram to be deleted is currently active in the call sequence.
- 252 DELETE of SUB statement. An attempt was made to delete a SUB or FUNCTION statement without deleting the entire subprogram segment.
- * 260 Statement error code for DIM, INTEGER, LONG
- 261 Illegal DIM type. The type of the identifier in a DIM statement is not valid for array typing.
- 262 Illegal DIM value. The subscript for a DIM statement is not a valid positive number.
- * 270 Statement error code for DIR
- * 280 Statement error code for DISABLE
- * 290 Statement error code for DISMOUNT
- * 300 Statement error code for ENABLE
- * 310 Statement error code for EXIT
- 311 Invalid level count on EXIT. An EXIT statement had a level count greater than the number of nested FOR loops.
- * 320 Statement error code for FOR, NEXT
- 321 FOR not local. The FOR statement and its matching NEXT statement are in different subprogram segments.
- 322 Control variable error. The FOR statement specifies a different control variable than its matching NEXT statement.
- 323 FOR statement deleted. The FOR statement has been deleted from the program list before the FOR loop finished execution.

- 324 Invalid flow of control on FOR. An attempt was made to return to a FOR statement from a NEXT statement without exiting a subprogram or subroutine that had been called since the FOR statement was last executed.
- 325 Nested FOR used same control variable as an outer loop.
- * 330 Statement error code for FORMAT
- * 340 Statement error code for GETMEM
- 341 No buffer specified on a GETMEM.
- * 350 Statement error code for GOSUB, GOTO
- 352 Line number not local. The line to branch to is not in the current subprogram segment.
- * 360 Statement error code for IF
- * 370 Statement error code for INIT
- * 380 Statement error code for INPUT
- 381 Input data not valid for this argument or data ran out before argument list did.
- * 390 Statement error code for assignment, REP\$
- 391 Invalid target variable. The variable being assigned to is not valid for the value being assigned.
- * 400 Statement error code for LIST, SLIST
- 401 Line cannot be uncompiled. A RENUMBER of a line has made it impossible to list that line.
- 402 Invalid line number. The value of a line number in a LIST statement is not valid.
- * 410 Statement error code for LOAD
- * 420 Statement error code for OFF
- * 430 Statement error code for ON
- * 440 Statement error code for OPEN
- * 450 Statement error code for POLL
- 451 Bad primary address in POLL list.
- 452 Bad secondary address in POLL list.
- * 460 Statement error code for PRINT
- * 470 Statement error code for PUTMEM
- 471 PUTMEM overflowed user buffer.
- 472 No buffer specified on a PUTMEM.
- * 480 Statement error code for RBYTE
- 481 Operand(s) invalid in proceed mode.
- * 490 Statement error code for RCALL
- 491 Romcall routine not found. Probably due to an error in typing the romcall routine name.
- * 500 Statement error code for READ
- 501 End of READ data. There was insufficient data to complete the READ item list.
- * 510 Statement error code for RENAME
- * 520 Statement error code for RESTORE
- 522 RESTORE line not DATA statement. The line number referenced by a RESTORE statement is not a DATA statement.
- 523 RESTORE line number not local. The line number referenced by a RESTORE statement is not in the current subprogram segment.
- * 530 Statement error code for SAVE
- 531 Incomplete subprogram error. An attempt was made to save a subprogram segment after saving an incomplete (no END statement) subprogram segment.
- 532 Bad item file format. The file specified for a load or append in item format is either not a program file or is damaged.
- * 540 Statement error code for SELECT
- * 550 Statement error code for SET

ERROR MESSAGES

- 551 Error in SET statement. There is an error in the construction of a SET statement.
- 552 SET argument out of range. An argument to a SET statement is not within a valid range for that argument.
- 553 Invalid FUZZ argument. An invalid number of digits were specified for FUZZ.
- 554 Bad date and time string. The string specifying the date and time in a SET TIME statement does not form a valid date and time.
- * 560 Statement error code for WAIT
- 561 WAIT value not valid. The value specified in a WAIT statement does not represent a valid time unit.
- * 570 Statement error code for WBYTE
- 571 WBYTE operand invalid.
- 572 Operand(s) not valid in proceed mode.
- Group XI (700-769) I/O Errors.**
- 700 Specified driver not present in the system. Probably a misspelled driver name, or a port number for an option that isn't available.
- 701 Too many left parenthesis characters.
- 702 Too many right parenthesis characters.
- 703 Unrecognized parameter. Each driver has a set of legal parameters.
- 704 Missing parameter right half. Each parameter must have a value assigned.
- 705 Boolean parameter has a bad value. Booleans must have values of 0 or 1.
- 706 Integer parameter has a bad value.
- 707 Real parameter has a bad value.
- 708 Clock tick parameter has a bad value.
- 709 Stream spec ended with a quote character.
- 710 Stream spec ended inside a string value.
- 711 Too many colon characters in stream spec.
- 712 Right parenthesis encountered in bracket.
- 713 Non-numeric encountered in numeric value.
- 714 Not enough characters in stream spec.
- 715 No number found by operating system call.
- 716 Missing left half of a parameter.
- 717 Bracket number too big.
- 718 Driver name not satisfied.
- 719 The SYSDEV stream spec has no driver spec component.
- 750 USING syntax error.
- 751 Data type and format do not match.
- 752 Data did not match %, @, | USING format.
- 753 Logical unit not open when it should be.
- 754 No logical unit open when data transfer attempted.
- 755 Input buffer too small.
- 756 Checksum failure on % format data.
- 757 Header failure on @, |, % format data.
- 758 Run out of data on % format data.
- 759 Numeric overflow on binary, octal, or hexadecimal input.

Group XII (770-799) GPIB Function Errors.

- 770 Bad data byte value in ATN function.
- 771 Missing DIO operand in PPC function.
- 772 Missing sense operand in PPC function.
- 773 Invalid DIO operand in PPC function.
- 774 Invalid sense operand in PPC function.
- 775 Invalid listen address. Invalid listen address in PPC function.
- 777 Invalid address in an addressed command function (GTL,SDC,GET,TCT).
- 778 Invalid operand in SRQ function.
- 779 Invalid use of GPIB function.

Group XIII (800-839) GPIB Driver Errors.

- 800 Invalid MA value in stream spec. Value must be in the range 0..30.
- 801 Invalid SC value in stream spec. Value must be either YES (1) or NO (0).
- 802 Invalid PRI value in stream spec. Value must be in the range 0..31. Note that the value of 31 is interpreted as addressing the interface itself, rather than a particular instrument attached to the bus.
- 803 Invalid SEC value in stream spec. Value must be in the range 0..32. Note that the value of 32 is interpreted as a request for no secondary address to be associated with the logical unit.
- 804 Invalid TRA value in stream spec. Value must be either NOR (0), FAS (1), or DMA (2).
- 805 Interface does not support DMA transfer mode. The standard interface supports normal and fast transfer modes, while the optional interface also supports direct memory access mode.

- 806 Invalid IST value in stream spec. Value must be TRU (1) or FAL (0).
- 807 Invalid PNS value in stream spec. Value must be in range 0..63 or 128..191.
- 810 Operation attempted which is not supported. File operations are not supported by the interface.
- 811 Data transfer operation timed out. Each data byte is given a programmable amount of time to be transferred before this error is reported.
- 812 No listener on the bus. This indicates either no instruments on the bus, or no instrument at the particular address specified in the stream spec.
- 814 Serial poll attempted when interface not CIC. Only the controller-in-charge can execute a POLL statement.
- 815 Autopoll attempted when SRQ not true. One or more instruments must be sending SRQ before an autopoll is executed.
- 816 Autopoll failed to detect SRQ source. Autopoll only succeeds if a status byte is received from an instrument with the rsv bit true.
- 817 Instrument failed during autopoll. This error occurs if an instrument returns a status byte with rsv false, and releases SRQ.
- 818 Explicit poll of non-existent instrument. If a list of addresses is supplied in the POLL statement, all addresses must respond with status bytes when polled.
- 820 WBYTE tried to send IFC when interface not SC. Only the system controller can send the interface clear message.
- 821 WBYTE tried to send REN when interface not SC. Only the system controller can send the remote enable message.
- 822 WBYTE tried to send ATN when interface not CIC. Only the controller-in-charge can send the ATN message.

ERROR MESSAGES

- 823 WBYTE tried to output when interface not talk addressed. The interface must be talk addressed before data can be output to the bus.
- 824 RBYTE tried to input when interface not listen addressed. The interface must be listen addressed before data can be input from the bus.
- 825 WBYTE tried to send SRQ when interface not talker/listener. Only a talker/listener can send the service request message.
- 826 Parallel poll attempted when interface not CIC. Only the controller-in-charge can execute a parallel poll (ATN(EOI)).
- 827 WBYTE sent TCT when no talker addressed. Some instrument must be talk addressed before the TCT command is sent.
- 830 Interface unable to take control of bus. The attention message was false even after the interface tried to take control asynchronously. This indicates a hardware problem.
- 831 Take control synchronous operation timed out.
- Group XIV (840-879) Tape Driver Errors.**
- 840 Lamp/Servo failure. The hole detect lamp has burned out or the servo could not get the tape up to speed in allotted time.
- 841 End of tape detected. An attempt to locate a record on the tape has caused the end of tape to be detected. This normally means that an inter-record gap was not detected.
- 842 Cartridge not in place. An attempt was made to access the tape drive without a tape cartridge in the drive.
- 843 Write inhibit. An attempt was made to write to the tape while the write protect tab was set to write protect.
- 844 Data overflow. The data service for the tape was not made in the time required and data was lost.
- 845 Additive checksum failure. The additive checksum for a header did not match the calculated checksum. This means the record data was read in incorrectly.
- 846 Incorrect record header. The record number read was not the record number expected.
- 847 Cyclic redundancy check failure. The cyclic redundancy check at the end of a record was in error. Probable cause: bad tape.
- 848 Gap detected. An inter-record gap was detected while writing or reading.
- 849 Timeout failure. The tape drive did not respond within the time expected.
- 850 File already exists. An attempt was made to create a file as "new" that already exists on the tape.
- 851 No space available. An attempt was made to create a file larger than the available space on the tape or an attempt was made to create a 49th file.
- 852 Illegal record number. An attempt was made to do a physical read/write to a non-existent record number for the tape being used.
- 853 No file exists. An attempt was made to open a file name that does not exist on the tape.
- 854 Write after read. An attempt was made to write to a tape file after one or more reads was made to a file.
- 855 End of file reached. The end of file was reached while accessing a tape file.
- 856 Illegal command. An illegal command was given to the tape drive (firmware error).
- 857 File not open. An attempt was made to access a tape file when the file had not been opened.
- 858 Wrong data type. An attempt was made to access a tape file with the wrong data type. An item file was passed ASCII data or an ASCII file was passed binary data.
- 859 Cannot access directory in physical mode.
- 860 File access failure. An attempt was made to open a tape file for reading while it was already open for writing, OR the tape was in logical mode and an attempt was made to open it in physical mode, OR the tape was in physical mode and an attempt was made to open it in logical mode.

- 861 Invalid tape. The tape inserted in the tape drive is not the original tape for the file being accessed.
- 862 Illegal SYSDEV request or attempt to set tape as system console device. A SET SYSDEV statement attempted to set a parameter to an illegal value (OPE = REP, OPE = UPD, or PHY = YES), or a SET CONSOLE statement attempted to make the tape the system console device.
- 863 Directory read error. The directory could not be read without detecting an error.
- 864 Invalid tape. An attempt was made to close an updated file with a different tape inserted in the drive.
- 865 Read after write failure. The read after write function detected an error.

Group XV (880-899) RS-232 Driver Errors.

- 880 Invalid request in item format. The EOM parameter may specify one or two characters, and each must have an ASCII code in the range 0 thru 127.
- 884 Invalid driver request. An I/O operation was attempted which was not possible to perform on an RS-232 interface.
- 885 Time out limit exceeded. The I/O operation could not be completed within the time specified by the time out parameter.
- 886 Typeahead buffer overflowed. The size of the type-ahead buffer specified is not large enough to hold all incoming data.
- 887 Invalid combination of BITs and PARity parameters. Only some combinations of these are valid.
- 888 Invalid baud rate specified.
- 889 Overrun error. A character received was lost, because a second character was received before the first could be placed in the typeahead buffer.
- 890 Parity error. A character was received with a parity bit that did not have the correct value as specified by the PARity parameter.

- 891 Framing error. The RS-232 hardware did not detect a stop bit or bits (as specified by the STOP parameter) following the character that was received.

Group XVI (900-919) Front Panel Driver Errors.

- 900 Interrupt Output Routine Mistakenly Called. The front panel ACIA has been enabled for output interrupts without a flag being set indicating to the interrupt routine that it is in the output state.
- 901 ACIA overrun error. During data transmission the ACIA data register has been overrun and data has been lost.
- 902 ACIA framing error. The transmission to the ACIA is out of synchronization.
- 903 ACIA fatal error. The ACIA has interrupted without any indication of what the interrupt is.
- 904 Unwanted Query Response. The front panel interrupt routine has been sent a handshake from the front panel processor when it was not expecting one.
- 905 Front Panel Printer Problem. On a print request the front panel processor is reporting that some error has occurred while accessing the printer.
- 906 Illegal wakeup message. The front panel driver has been woken up by a message that was not sent from the interrupt routine.
- 907 Flush Task Code. The operating system has flushed the front panel driver or the driver has timed out.
- 908 Unable to allocate memory. The front panel driver was not able to allocate enough memory to perform the specified task.
- 909 Illegal Request Message. The format of the message sent to the front panel driver is illegal.
- 910 Illegal Driver Request. The front panel driver does not recognize the specified request.
- 911 Front panel timeout.

**APPENDIX A
Group XIX**

ERROR MESSAGES

| ERROR CODE | DEFINITION | | |
|-----------------------|---|------|--|
| 1201 | Illegal file/volume name. The file or volume name specified contains illegal characters. The legal characters allowed are 'A-Z, 1-9', underscore and period. The underscore and period cannot be the first character, but can be any other character including the last character. | 1208 | Wrong data type. An attempt was made to input item data from a disk file with the target variable being an incorrect data type (i.e. input to a string and data item is a scalar). The next data type can be obtained by issuing a 'type(n)' function for the logical unit assigned to the disk file. |
| 1202 | Insufficient room on disk for file. An attempt was made to create a file with a size specified that exceeds the largest free space on the disk. | 1209 | Rbyte/Wbyte record number out of range. An attempt was made to access a non-existing sector on a disk in physical mode. |
| 1203 | Insufficient space in directory for file specified. An open for 'NEW' was issued with a full directory. If this occurs often and the full disk space is not entirely used, future disks should be formatted with additional directory entries. See the 'ENT=nnn' stream specification parameter for more details. | 1210 | Physical mode access failure. An attempt was made to issue a Rbyte/Wbyte to a logical unit open in logical file mode. To use Rbyte/Wbyte to the disk, the logical unit must be open with the 'PHYSical=yes' stream specification. |
| 1204 | Attempted to open an existing file as 'NEW'. This warning message indicates file already exists. If the file is to be overwritten, use the 'REPlace' parameter for the open type. | 1211 | Directory entries greater than 1/16 of disk. An attempt was made to format a disk specifying more directory entries that can be placed on 1/16 of the disk. The 'ENT' parameter specifies the number of directory entries. The number of sectors used for the directory can be obtained using the following formula. |
| 1205 | Attempted to open a non-existing file. An attempt was made to open a non-existing file with a stream specification stating 'OLD' or 'UPDate'. | 1212 | Illegal physical mode operation. The number of parameters being passed were incorrect or out of range. |
| 1206 | END-OF-FILE reached for disk file specified. While reading from a file an attempt was made to obtain data beyond the last item on the file. If the end-of-file must be detected prior to data access, use the 'ON EOF(n)' feature of 4041 Basic. | 1213 | Write protected media. An attempt was made to write to a file on a disk that had a write protect tab in place. |
| 1207 | Unformatted or illegal disk/diskette. An attempt was made to access a hard disk or floppy disk that was not formatted with 4041 Basic. This message may also be reported when a disk does not contain a disk/diskette or the drive/controller is malfunctioning. | 1214 | Read after Write verification failure. The data read back, from a sector just written, was incorrect. |
| | | 1216 | Parity error on data transfer. The data transferred to/from the disk drive caused a parity error. This error can only occur if the PARity stream specification parameter is set to YES. |
| | | 1217 | An attempt was made to rename a file to an existing file name on the same disk. |

ERROR MESSAGES

NOTE

The following error codes are specific to the SCSI H/W contained on the Option 03 board and normally should not be encountered by the user. If these error codes appear regularly, they may indicate an Option 03 hardware problem.

| | | | |
|------|---|------|--|
| 1221 | SCSI chip detected an illegal command. This error normally indicates that the firmware and hardware are out of sync with each other. | 1226 | SCSI chip data byte timeout error. The SCSI chip indicated a data byte reading was available, but did not present that byte of data within a reasonable period of time. This error indicates possible Option 03 board hardware problems. |
| 1222 | No function complete after SELECT with Attention. When the Option 03 firmware issued a select to a disk device, the first response was not a function complete. This indicates Option 03 board hardware problems. | 1227 | Command Complete not received for last operation issued. This error indicates possible Option 03 board hardware problems. |
| 1223 | SCSI aux register incorrect after SELECT with Attention. This error, similar to error 1222, indicates Option 03 board hardware problems. | 1228 | SCSI chip power-up diagnostic failure. This error indicates the SCSI controller chip did not pass its power-up diagnostic tests. The user may wish to try powering off the 4041 and then re-applying power to see if this error goes away. If the error remains, the SCSI chip is in operative and Tektronix service personnel should be notified. |
| 1224 | No function complete on disconnect. When the disk device tasks were completed, the response received from the Option 03 specified the device had disconnected from the SCSI bus, but the interrupt status register did not contain the correct status for this operation. This also indicates possible Option 03 board hardware problems. | 1229 | SCSI selection timeout. An operation was issued to a device on the SCSI bus with no response received. This error may indicate a hardware problem with the Option 03 board or the device being accessed. |
| 1225 | SCSI Operation Complete error. This error indicates that during the disk operation completion phase, an error was discovered. This error should never be displayed to the user because the error detected was sent from the disk controller and the actual error number sent is mapped into error codes 1240-1299. Thus, if a disk controller sent an error code 01, the user would see error code 1241. Since each manufacturer of disk controllers uses different error numbers, the user needs to subtract the base value of 1240 (decimal) to obtain the actual error code sent from the disk controller. The same error number sent can also be obtained by issuing an 'ASK\$("lu",n)' for the logical unit numbering being used and obtain the error code from the 'SEN-xxx' parameter. | | |

NOTE

The following errors are controller and device specific errors.

1240 - 1499 These errors indicate actual errors received from a disk controller with the base of 1240 (decimal) added. The same error code can be examined using the 'ASK\$("lu",n)' function ('n' being the device's logical unit number) and the stream specification parameter 'SEN=xx' (the sensed error code received from the disk controller). The stream specification parameter is not cleared by Option 03 until the logical unit is closed or until another device error is received.

Error Codes 2000 and Up

Error codes 2000 and up are reported differently. The 4041 adds an offset value of at least 2000, and divisible by 1000, to the error code. The offset value added represents the error code group. Associating an offset value with a particular error code group is dependent on the configuration of the 4041, and can be determined by the command: ASK\$("rompack"). For example, a 4041 with Options 01, 10, and 30 will return the following to the command ASK\$("rompack"):

```
GRPH,2000;PLOT,3000;SGPR,4000;UTL1,5000;
OPT1;PD;IO;XO;UTL2,6000
```

If the 4041 should report an error of, let's say, 6014, the command ASK\$("rompack") would be used to determine the error code group. First, the highest number divisible by 1000 that can be subtracted from the error code, with the result being a positive integer, is subtracted; in this case, 6000. Next, ASK\$("rompack") is used to find the error code group in that range (6000 - 6999). In the above example, it would be found to be "UTL2". Referring now to the error codes listed for UTL2 (below), error 14 (6014 - 6000 = 14) is found to be "array types not the same".

In the above example, error codes for Option 10 are listed in the range 2000 to 5999 (GRPH = Graphics, PLOT = Plotting, SGPR = Signal Processing, UTL1 = Option 10 Utilities), and UTL2 errors begin at 6000. Refer to the Option 10 Instruction manual for error code listings for Option 10.

UTL2 Errors

NOTE

Errors for UTL2 will be reported with an offset value added. Refer to the above discussion, "Error Codes 2000 and Up".

- | | | | |
|---|--|----|--|
| 1 | Incorrect number of parameters passed. | 4 | No data bytes transferred over the IEEE-488 bus. |
| 2 | Integer array type expected as parameter and not found. | 5 | Not enough memory available for data passed over the IEEE-488 bus. |
| 3 | Integer scalar type expected as parameter and not found. | 6 | Bit count is out of range (i.e. $8 \leq \text{bit count} \leq 16$). |
| | | 7 | Odd number of bytes transferred over the IEEE-488 bus. |
| | | 8 | Checksum failure. |
| | | 9 | Parameter type incorrect. |
| | | 10 | Real array type expected and not found. |
| | | 11 | Source array not defined. |
| | | 12 | Segment extends beyond array boundaries. |
| | | 13 | Destination array not large enough for segment. |
| | | 14 | Array types not the same. |
| | | 15 | Destination array parameter not numeric array type. |
| | | 16 | Real scalar type expected as parameter and not found. |
| | | 17 | Destination array dimensioned too small for source array. |
| | | 18 | Source array not real or integer array type. |
| | | 19 | Destination array dimensioned too small for incoming data. |
| | | 20 | No percent sign found in data transferred. |
| | | 21 | Source array not integer array type. |
| | | 22 | Source array not real or integer array type. |
| | | 23 | An attempt was made to reference an array element out of bounds. |
| | | 24 | Integer array expected as destination array and not found. |

Appendix B

ASCII (GPIB) CODE CHART

| B7 B6 B5 BITS | | 0 0 0 | 0 0 1 | 0 1 0 | 0 1 1 | 1 0 0 | 1 0 1 | 1 1 0 | 1 1 1 |
|------------------|---------|-----------------------|-----------------------|---------------------|-------|-------------------|-------|---|-----------------|
| B4 B3 B2 B1 | | CONTROL | | NUMBERS SYMBOLS | | UPPER CASE | | LOWER CASE | |
| 0 | 0 0 0 0 | NUL | DLE | SP | 0 | @ | P | ' | p |
| 1 | 0 0 0 1 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 2 | 0 0 1 0 | STX | DC2 | " | 2 | B | R | b | r |
| 3 | 0 0 1 1 | ETX | DC3 | # | 3 | C | S | c | s |
| 4 | 0 1 0 0 | EOT | DC4 | \$ | 4 | D | T | d | t |
| 5 | 0 1 0 1 | ENQ | NAK | % | 5 | E | U | e | u |
| 6 | 0 1 1 0 | ACK | SYN | & | 6 | F | V | f | v |
| 7 | 0 1 1 1 | BEL | ETB | ' | 7 | G | W | g | w |
| 8 | 1 0 0 0 | BS | CAN | (| 8 | H | X | h | x |
| 9 | 1 0 0 1 | HT | EM |) | 9 | I | Y | i | y |
| 10 | 1 0 1 0 | LF | SUB | * | : | J | Z | j | z |
| 11 | 1 0 1 1 | VT | ESC | + | ; | K | [| k | { |
| 12 | 1 1 0 0 | FF | FS | , | < | L | \ | l | * |
| 13 | 1 1 0 1 | CR | GS | - | = | M |] | m | } |
| 14 | 1 1 1 0 | SO | RS | . | > | N | ^ | n | ~ |
| 15 | 1 1 1 1 | SI | US | / | ? | O | _ | o | DEL (RUBOUT) |
| | | ADDRESSED COMMANDS | UNIVERSAL COMMANDS | LISTEN ADDRESSES | | TALK ADDRESSES | | SECONDARY ADDRESSES OR COMMANDS (PPE) | (PPD) |

KEY

| | | | |
|-------|------------|-----|-----------------|
| octal | 25 | PPU | GPIB code |
| | NAK | | ASCII character |
| hex | 15 | 21 | decimal |

Tektronix
COMMITTED TO EXCELLENCE

REF: ANSI STD X3. 4-1977
IEEE STD 488-1978
ISO STD 646-1973

TEKTRONIX STD 062-5435-00 4 SEP 80
COPYRIGHT © 1979, 1980 TEKTRONIX, INC. ALL RIGHTS RESERVED.

Appendix C

GLOSSARY

| Term | Definition | Term | Definition |
|----------------------|--|-------------------|---|
| abort | To terminate the execution of a program or command. The terminated program or command can be restarted, but cannot be continued from the termination point. | BASIC | An acronym derived from Beginner's All-purpose Symbolic Instruction Code. BASIC is a high-level programming language that uses English-like instructions. |
| active call sequence | The sequence of program segments executing. The sequence always contains the main program segment, and also includes any subprogram segments that are invoked or being returned from. | BASIC interpreter | A set of microprocessor instructions that give the 4041 the ability to understand and execute BASIC statements. The BASIC interpreter resides in Read Only Memory and is part of the operating system. |
| argument | A value operated on by a function or keyword. | binary operator | Operators that return values based on the binary representations of their operands. Binary operators in 4041 BASIC include BAND, BOR, BXOR, and BNOT. |
| arithmetic operator | An operator that describes an arithmetic operation. Arithmetic operators in 4041 BASIC include +, -, *, /, ↑, DIV, MOD, MIN, and MAX. | bit | A binary digit. A unit of data in the binary numbering system; a 1 or 0. |
| array | A collection of data items referenced by a single variable name. In 4041 BASIC, arrays may be one- or two-dimensional, i.e., organized into rows, or rows and columns. | byte | A group of consecutive binary digits operated on as a unit. One ASCII character, for example, is represented by one binary byte. |
| array variable | A name corresponding to a (usually) multi-element collection of data items. | character string | A connected sequence of ASCII characters, sometimes referred to simply as a "string". |
| ASCII Code | A standardized code of alphanumeric characters, symbols, and special "control" characters. Each of the 128 characters in the ASCII character set is uniquely represented by a seven-bit binary code and by a decimal equivalent. ASCII is an acronym for American Standard Code for Information Interchange. | concatenate | To join together two character strings with the concatenation operator (&), forming a longer character string. |
| assignment statement | A statement used to give a value to a variable. | console | The part of the computer system used for communication between the operator and the computer. |
| | | constant | A number that appears in an expression in its actual form. In the following expression, 4 is a numeric constant: $X = 4 * P$. In the following expression, "JOHN" and "DOE" are string constants: $Name\$ = "JOHN" \& CHR\$(32)\& "DOE"$. |

GLOSSARY

| Term | Definition | Term | Definition |
|----------------------------|--|-------------|--|
| debug | The process of locating and correcting errors in a program; also, the process of testing a program to ensure that it operates properly. | function | A special purpose operation referring to a set of calculations within an expression, as in the sine function, square root function, etc. The user may define functions in 4041 BASIC using the FUNCTION statement. |
| default | The property of a computer that enables it to substitute values for statement parameters when none are specified. | GPIO | An acronym for general-purpose interface bus. An eight-bit parallel interface that meets IEEE 488-1978 specifications. |
| delimiter | A character that fixes the limits, or bounds, of a string of characters. | grad | 1/100th of a right angle. |
| degree | 1/90th of a right angle. | handshake | Greeting protocol between two devices. Each device must send and respond to this series of signals prior to data transfers. |
| direct memory access (DMA) | A 4041 optional feature (Option 01) which allows data transfer directly from the GPIB bus to memory without processor intervention. This allows for a very high-speed transfer burst. | index | A number used to identify the position of a specific quantity in an array or string or quantities. (For arrays, also referred to as a "subscript".) |
| driver | A special firmware program that controls an external device or interface between devices. | initialize | Set all parameters to their initial power-up or default values. |
| dyadic | Referring to an operator having two operands. | input | Data transferred to the 4041 memory through a firmware driver. |
| error | A discrepancy between an expected or required condition and the observed condition; a fault or failure. | integer | A whole number in the range from -32768 to 32767. It cannot contain a decimal point. |
| error handler | Instructions within a program that determine what action is taken when an error is detected. | interrupt | To cause an operation to be halted in such a fashion that it can be resumed at a later time. |
| execute | To perform the operations indicated by a statement or group of statements. | justify | To align a set of characters to the right or left of a reference point. |
| expression | A collection of variables, constants, and functions connected by operators in such a way that the expression as a whole can be reduced to a constant. Expressions can be either numeric expressions or string expressions. | keyboard | The device that encodes data when keys are pressed. |
| | | keyword | An alphanumeric code that the 4041 recognizes as a function to be performed. |

| Term | Definition | Term | Definition |
|--------------------|---|------------------|--|
| line number | A number establishing the sequence of execution of lines in a program. In 4041 BASIC, line numbers must be whole numbers in the range from 1 to 65535. | numeric function | Special purpose mathematical operations that reduce their associated parameters (or arguments) to a numeric constant. |
| logical operator | Operators that return logical 1's and 0's; specifically, the AND, OR, XOR, and NOT operators. "True" operations return "1", "false" operations return "0". | numeric variable | A variable that can contain one or more numeric values. Simple (or "scalar") numeric variables contain a single numeric value; array numeric variables may contain more than one. |
| logical unit | A way of symbolically referencing an I/O driver. The OPEN statement connects the symbolic logical unit number with the I/O driver and its parameters (I/O stream specification). | operand | Any of the quantities involved in an operation. Operands may be numeric expressions or constants. In the numeric expression $A = B + 4 * C$, the numeric variables B and C and the numeric constant 4 are operands. |
| mantissa | In scientific notation, that part of the number that precedes the exponent. For example, the mantissa in the number 1.234E+ 200 is 1.234. | operator | A symbol indicating the operation to be performed on one or two operands. |
| memory | Generally refers to the random access memory, RAM, used to store 4041 BASIC programs and data, as opposed to the read-only memory, or ROM. | output | Data transferred from the 4041 memory to a driver. |
| monadic | Referring to an operator that has only one operand. | parameter | A quantity whose value affects the way the 4041 executes statements. |
| numeric constant | Any real number entered as numeric data; also, the contents of a numeric variable. | polling | Interrogation of devices to determine status or to avoid contention. |
| numeric expression | Any combination of numeric constants, numeric variables, array variables, subscripted array variables, numeric functions, or string relational comparisons enclosed in parentheses, joined together by one or more arithmetic, logical, or relational operators in such a way that the expression as a whole can be reduced to a single numeric constant. | program | A series of instructions in a form understandable to a computer. |
| | | program segment | A section of a program bounded by either: a) the program start and an END statement (the main program segment) or b) a SUB or FUNCTION statement and an END statement (a subprogram segment). |
| | | protocol | A code or precedence that must be strictly adhered to. |
| | | radian | A unit of arc equal to $360^\circ / (2 * \pi)$. |

GLOSSARY

| Term | Definition | Term | Definition |
|---------------------|---|-------------------|--|
| RAM | Random access memory; the memory used for temporary storage of programs and data and as workspace during program execution. | statement | A keyword plus any associated parameters. |
| relational operator | An operator that causes a comparison of two operands and returns a logical result. Comparisons that are "true" return a "1", comparisons that are "false" return a "0". The relational operators in 4041 BASIC are = , < > , < , > , < = , and > = . | string | A connected sequence of alphanumeric characters. |
| ROM | Read-only memory; that portion of the system memory that cannot be changed by the user. The 4041 BASIC operating system and firmware resides in ROM. | string constant | A string of characters of fixed length enclosed in quotation marks; also, the contents of a string variable. |
| ROM pack | A portion of ROM supplied in a form that can be added to or removed from the 4041. It generally supplies additional instructions or features to the standard ROM. | string expression | An expression that evaluates to a string constant. |
| rounding | Adjusting the least significant digits retained in truncation to partially reflect the dropped portion. For example, when rounded to three digits, the number 2.7561 becomes 2.76. | string function | Special purpose functions that manipulate character strings and produce string constants. |
| scalar | A single numeric or string value. | string variable | A variable that contains only alphanumeric characters, or "strings". String variables have a dollar sign (\$) in the last character of their variable names. They have a default length of 20, i.e., they can contain up to 20 characters without being dimensioned in a DIM statement. |
| scientific notation | A format representing numbers as a fractional part, or mantissa, and a power of 10, or characteristic, as in 1.23E45. | subprogram | A segment of a 4041 program bounded by a SUB or FUNCTION statement and an END statement. Sometimes used to denote specifically subprograms that begin with SUB statements. Subprograms are usually arranged in such a way that control passes to the subprogram and returns to the point at which the subprogram was called/invoked when the subprogram has completed execution. |
| segment | A portion of a 4041 program. Segments can be of two types: (1) "main" program segments, bounded by the first statement in the program and the first END statement; or (2) "subprogram" segments, bounded by a SUB or FUNCTION statement and an END statement. | subscript | An index used to refer to an element of an array. In the following example, 10 is a subscript that refers to the 10th element of array B: B(10). |
| | | substring | A portion of a larger string; "BC", for example, is a substring of the string "ABCD". |

| Term | Definition | Term | Definition |
|-----------------|--|---------------|--|
| target line | A program line specified as a destination for program control or alteration. | variable | A symbol, corresponding to a location in memory, whose value may change as a program executes. |
| target variable | Any variable specified as a target to receive incoming data or the results of an operation. | variable name | A name selected by the programmer to represent a specific variable. |
| truncating | Reducing the number of least significant digits present in a number, without reflecting the dropped portion. For example, the number 5 is the result of truncating the fractional part of the number 5.76. | | |

Appendix D

STREAM SPECIFICATIONS

When an I/O operation is to be performed using a device other than the system console, the user must specify the path the data will take. The user does this by means of stream specifications and logical units.

A stream specification ("stream spec", for short) selects a driver for an I/O operation and specifies parameters (i.e., settings) for the driver to use.

Thus, stream specifications define paths for data during I/O operations.

FORM OF A STREAM SPECIFICATION

Stream specifications are string expressions that take the form

```
driver-spec  
or  
file-spec  
or  
driver-spec file-spec
```

If a stream specification does not include a driver spec, the current SET SYSDEV driver spec (default:"TAPE:") is assumed.

Including the colon in the stream spec is very important, because if no colon is included the 4041 will assume that the spec is a file spec, and that the user wants the I/O operation performed on the "TAPE:" driver.

Example. Consider the stream spec

```
"gpiB0:"
```

This stream spec designates the standard GPIB interface port as the data path for an I/O operation.

Now consider the stream spec

```
"gpiB0"
```

Since no colon is included in the stream spec, the 4041 interprets it as a file spec. Thus, the 4041 assumes the user wishes the I/O operation to be performed with a file named "GPIB0" on the DC-100 tape.

FORM OF A DRIVER SPECIFICATION

Driver specifications take the form

```
driver-name[(driver-parameter[,driver-parameter]...):
```

Driver specifications consist of a driver name, followed optionally by one or more parameters in parentheses describing or modifying the operation of the driver, FOLLOWED BY A COLON.

Legal driver names consist of four characters, followed optionally by a port number for GPIB and COMM drivers. If no port number is included in the driver name, the port number "0" is assumed.

The legal driver names can be discovered by invoking the ASK\$("DRIVER") function, e.g., executing the statement PRINT ASK\$("DRIVER") in immediate mode.

FORM OF A FILE SPECIFICATION

File specifications take the form

```
file-name[(file-parameter[,file-parameter]...)]
```

File specifications, used with directory-oriented file-structured devices, consist of a file name followed optionally by one or more file parameters.

If a stream spec does not include a driver spec, the current SET SYSDEV driver spec is chosen by default, and the stream spec is interpreted as a file spec (see the preceding example).

PARAMETERS

Parameters are "settings" that the user can adjust to modify the way I/O operations are performed.

Examples of these settings include:

- The baud rate for the RS-232-C interface(s), controlled by the "BAUD" parameter associated with the COMM driver(s).
- The primary address of a device on the GPIB (when I/O is addressed to a particular GPIB device), controlled by the "PRI" parameter associated with the GPIB driver(s).
- The rate at which characters are scrolled across the alphanumeric display, controlled by the "RATE" parameter associated with the FRTP driver.
- The End-Of-Message delimiter, controlled by the "EOM" parameter associated with each driver.

Parameter names are only significant to three characters. Additional characters may be included for documentation or readability, but are ignored otherwise. Parameter names are given in this section with significant characters in upper case and additional characters (if any) in lower case.

Within a stream spec, parameters may be specified in any order.

Parameters are of three types: "logical", "physical", and "ASK\$".

LOGICAL PARAMETERS

Most driver parameters are "logical" parameters. These parameters are associated with the characteristics of the message and the way it is input or output.

Example.

```
"frtp(view=2.0,rate=.25):"
```

This stream spec specifies that output is to appear on the front-panel alphanumeric display and remain on the display for two seconds before the next statement is executed. (If the next statement is not a PRINT statement to the front panel, the message will remain on the display.) If the message takes up more than twenty characters, characters after the twentieth "scroll" in to the right-hand side of the display at the rate of one every 0.25 second.

Setting Logical Parameters

A logical parameter can be set within any statement that includes a stream spec by specifying the value of the parameter within parentheses after the driver/file name.

Default values are used for any logical parameters whose values are not specified within the stream spec.

Example.

```
Save "file1(ope=new,cli=yes)"
```

This command saves a file called File1 on the DC-100 tape, and specifies that File1 is a new file (not already on the tape). The SIZE parameter defaults to one-half of the largest free block remaining on the tape. The file will be "clipped" when closed, i.e., any unused space previously reserved for the file is deleted from the file and returned to the available space pool.

When a logical unit is opened, the logical parameters associated with the stream spec used in the OPEN statement become associated with that logical unit. Any unspecified parameters in the stream spec go to their default values.

Example.

```
100 Open #5:"gpib1(pri=5):"
111 Dim setting$ to 300
120 Input #5 prompt "set?":setting$
```

Line 100 assigns the stream spec "GPIB1(PRI = 5):", designating the device at primary address 5 on the optional GPIB interface port, and assigns it to logical unit 5. All other logical parameters of this stream spec (SEC, EOU, EOQ, etc.) go to their default values.

Line 120 then asks for input from this device, via logical unit 5.

PHYSICAL PARAMETERS: COMM, GPIB, AND OPT2 DRIVERS

The COMM, GPIB and OPT2 drivers each have several "physical" parameters. These parameters are not associated with the logical characteristics of individual message transfers, but with the device drivers themselves (and hence only one value can be typically set at the start of program execution and not changed thereafter. [EXCEPTION: The GPIB drivers' IST (Interface Status) parameter, used during parallel polls (see description following).]

Examples of physical parameters include the baud rate (BAUD), number of stop bits (STOP), and parity (PARITY) for the COMM drivers, and the system controller (SC) and the 4041's primary address (MA) for the GPIB drivers.

Physical parameters are set by means of the SET DRIVER statement. For example, the command in Example D-1 tells the 4041 that the device connected to the standard RS-232-C interface port will communicate at a baud rate of 4800 bps, transmit 8-bit bytes with even parity, and use two stop bits for synchronization.

Setting Physical Parameters: The SET DRIVER Statement

Physical parameters are set by including their values in a stream spec inside a SET DRIVER statement.

Both physical and logical parameters may appear in the stream spec used with the SET DRIVER statement, but only physical parameters are affected.

Defaults: COMM Drivers

Physical COMM parameters whose values are not specified within a stream spec contained in a SET DRIVER statement go to their default values.

Example.

```
200 Set driver "comm0(bau=300,bit=7,par=odd):"
.
500 Set driver "comm0(par=no):"
```

Line 200 sets the standard RS-232-C interface port to a baud rate of 300 bps, exchanging seven-bit characters with odd parity.

Line 500 changes the setting of the PAR parameter to NO parity; however, it also changes the baud rate back to its default value of 2400, and the BIT parameter back to its default value of 8.

Defaults: GPIB and OPT2 Drivers

GPIB and OPT2 physical parameters whose values are not specified within a stream spec contained in a SET DRIVER statement remain unchanged from their previous values.

Example.

```
100 Set driver "gpib0(sc=no,ma=29,ist=fal):"
.
200 Set driver "gpib0(ist=tru):"
```

Line 100 specifies that the 4041 is not the system controller on the bus attached to the standard GPIB interface port, that the 4041's primary address on that bus is 29, and that the value of its Interface Status message (used for responding to parallel polls) is false.

Line 200 sets the value of the Interface Status message to true, but does not change the other physical parameters.

(Note the difference between the way the GPIB drivers and the COMM drivers handle the default values for physical parameters.)

```
120 Set driver "comm(bau=4800,par=even,bits=8,stop=2):"
```

Example D-1.

STREAM SPECIFICATIONS

ASK\$ PARAMETERS

Certain parameters on the GPIB and COMM drivers cannot be specified by the user, but are assigned values by the 4041 during execution. These values provide the user with information about the status of the driver.

The user can examine the values of these parameters for a given logical unit by invoking the ASK\$("LU") function. (See Section 5, *Environmental Control*, for more information about ASK and ASK\$ functions.)

Example.

```
200 Open #3:"gpib0(pri=3):"  
210 Print ask$("lu",3)
```

Line 200 assigns the device at primary address 3 on the bus connected to the standard GPIB interface port to logical unit 3.

```
GPIB0(MA=30,SC=YES,CIC=4,TL=0,ENA=0,PEN=0,PRI=3,...
```

Line 210 invokes the ASK\$("LU") function, and prints the current stream spec associated with that logical unit on the system console device. Assuming that the standard GPIB interface port is set to its default settings, the stream spec contains the settings shown in Example D-2.

- "CIC = 4" Indicates that the 4041 is in the controller active state (CACS).
- "TL = 0" Indicates that neither the 4041 nor any other device on the bus is addressed to talk or listen.
- "ENA = 0" Indicates that no GPIB interrupts are enabled.
- "PEN = 0" Indicates that no GPIB interrupts are pending.

Example D-2.

LOGICAL UNITS

LOGICAL UNITS AS “SHORTHAND” FOR STREAM SPECIFICATIONS

A logical unit can be used as a “shorthand” way of referencing a stream spec in an INPUT, PRINT, RBYTE, or WBYTE statement.

A logical unit is associated with a particular stream spec via an OPEN statement. After the logical unit is opened, that stream spec can be referenced by means of the logical unit in a “#” clause.

Example 1.

```
1150 Print #frtp(vie=2.5, rat=0.5):"a,b,c
```

Example 2.

```
1140 Open #1: "frtp(vie=2.5, rat=0.5):"
1150 Print #1:a,b,c
```

Examples 1 and 2 produce the same output, with the same parameters, on the front-panel alphanumeric display. (The second example would run faster, and hence shows the preferred method.)

Logical unit numbers may range from 0 to 32767, inclusive.

THE “#” CLAUSE

When a “#” is followed by a string expression in an INPUT, PRINT, RBYTE, or WBYTE statement, the expression is interpreted as the stream spec to be used for the I/O operation.

When a “#” is followed by a numeric expression in an INPUT, PRINT, RBYTE, or WBYTE statement, the expression is interpreted as the logical unit to be used for the I/O operation.

When the 4041 recognizes a “#” followed by a string expression, it automatically opens a temporary logical unit, performs I/O on that logical unit, then closes the logical unit.

When the 4041 recognizes a “#” followed by a numeric expression, it performs I/O on that logical unit without closing it when the I/O is complete.

LOGICAL UNIT DEFAULTS

Logical units 0 through 30 default to the corresponding primary addresses on the standard GPIB interface; logical units 31 through 32767 default to the system console device.

STREAM SPECIFICATIONS

**Table D-1
PHYSICAL PARAMETERS FOR "COMM:" DRIVERS**

| Parameter | Possible Values | Default | Comments |
|------------------|---|----------------|--|
| BAUd | 75, 150, 300, 600, 1200, 2400, 4800 | 2400 | Baud rate, transmitting and receiving, bits/sec. |
| IBaud | Any integer from 2 to 9600 or 0 | | Integer baud rate: any integer specified (except 0) takes precedence over BAUD. 0 means use the BAUD parameter value. |
| BIT | 5, 6, 7, 8, 9 | 8 | # bits/character. Includes parity bit if used. If BIT = 5, cannot use parity. If BIT = 9, must use parity. |
| PARity | NO, ODD, EVen, HIGH, LOW | NO | <p>PAR = NO: No parity bits are set on output, no parity checking performed on input.</p> <p>PAR = ODD, EVen: On output, parity bit is set to make number of 1's in each character odd or even, as appropriate; on input, characters are checked for specified parity.</p> <p>PAR = HIGH, LOW: on output, bit of 1 or 0 is added to each character; on input, characters are checked for bit of 1 or 0.</p> <p>If BIT = 5, cannot use parity. If BIT = 9, MUST use parity.</p> |
| STOp | 1 or 2 | 2 | # of stop bits appended to each character transmitted. When BITs = 5 and STOp = 2, actual stop value used is 1.5. |
| FLAgging | NO, INPut, OUTPut, BIDirectional, MODem, AMODem, EMODem | OUTPut | <p>FLA = NO: No flagging.</p> <p>FLA = INP: Stop input into 4041 when typeahead buffer is ¾ full by transmitting DC3. When typeahead is empty, 4041 requests input continue by transmitting DC1.</p> <p>FLA = OUT: 4041 stops output when DC3 received (CTRL-S from user terminal). 4041 resumes output when DC1 received (CTRL-Q from user terminal).</p> <p>FLA = BID: Combine INPut and OUTPut.</p> <p>FLA = MOD: uses CTS/DTR lines for flagging.</p> <p>FLA = AMO: uses CTS/RTS lines for flagging.</p> <p>FLA = EMO: uses CTS/DSR and DTR/RTS line pairs for flagging.</p> |
| EDIt | RASter, STOrage, 402, ANSi, 850 | RASter | <p>EDI = RAS: "generic" raster-scan terminal.</p> <p>EDI = STO: storage or non-video terminal.</p> <p>EDI = 402: TEKTRONIX 4020-Series terminal.</p> <p>EDI = ANS: ANSI "standard" terminal (must have Kill-to-end-of-line, Delete-character, Insert capabilities).</p> <p>EDI = 850: CT8500 terminal.</p> |
| FORmat | ASCIi, ITEm | ASCIi | <p>If FOR = ASCIi, character codes between 128 and 255 have 128 subtracted from them, thus mapping them to the ASCII codes 0 to 127.</p> <p>If FOR = ITEm, each character code is used as-received.</p> |
| TYPeahead | = > 100 | 100 | <p>Size of typeahead input buffer in bytes.</p> <p>Maximum size: up to 32767 bytes, limited by amount of memory available.</p> <p>If a size smaller than 100 bytes is specified, a size of 100 is used.</p> |

(continued)

Table D-1 (cont)
PHYSICAL PARAMETERS FOR "COMM:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|--|
| DSR | OFF, ON | ON | Data Set Ready modem control line (output). |
| CTS | OFF, ON | ON | Clear To Send modem control line (output). |
| DCD | OFF, ON | ON | Data Carrier Detect modem control line (output). |
| ERR | LOG, REPort | REP | LOG adds parity, framing, or overrun errors into #ERror counter, but no immediate report of the error is given. REPort immediately displays error messages on the console for parity, framing, or overrun errors. |

Table D-2
LOGICAL PARAMETERS FOR "COMM:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|---|---|---|
| ECHo | YES, NO | YES | If ECHo = YES, 4041 echoes all characters received. |
| CONtrol | YES, NO | YES | If NO, control characters are transmitted as a caret followed by the control character's upper-case equivalent. If YES, control characters are transmitted as control characters (i.e., ASCII range 0 through 31). |
| CR | CR, CRLf, LFCr | CRLf | How carriage return is transmitted. |
| LF | LF, CRLf, LFCr | CRLf | How line feed is transmitted. |
| TIMEout | Any positive # | 2.14748E7 | Maximum time 4041 will wait for COMM input (seconds). |
| TMStop | Any positive # | 60 (FLA = MOD, AMO, EMO); 2.14748E7 otherwise | Maximum time (seconds) 4041 will wait before outputting consecutive characters to the COMM. |
| EOM | Any ASCII character in the range <0> - <255>; OR One or two printable ASCII characters; OR A string of one or two-character length. | <13> | End-of-Message string (EOM = <0> means no EOM character). |
| EOA | Any printable ASCII character; OR any ASCII character in the range <0> - <255> | <0> | End-of-Argument character. |
| EOH | Any printable ASCII character; OR any ASCII character in the range <0> - <255> | <0> | End-of-Header character. |
| EOU | Any printable ASCII character; OR any ASCII character in the range <0> - <255> | <9> | End-of-Message-Unit character. |

STREAM SPECIFICATIONS

Table D-3
ASK\$ PARAMETERS FOR "COMM:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|--------------------------|---------|---|
| #TY | Any non-negative integer | | Returns number of characters currently in typeahead buffer. |
| #ER | Any non-negative integer | | Returns number of parity, framing, or overrun errors since last ASK\$("LU") on COMM driver; used only when ERR = LOG. |
| RTS | OFF, ON | | Returns status of RTS (Request to Send) line. |
| DTR | OFF, ON | | Returns status of DTR (Data Terminal Ready) line. |

Table D-4
LOGICAL PARAMETERS FOR "FRTP:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|--|-----------------|---|
| RATe | Any positive number | 0.25 | Amount of time display "stops" before next character scrolls into right-hand side of display. |
| VIEw | Any positive number | 1.5 | Amount of time line is displayed before next statement is executed. |
| TIMEout | Any positive number | 2.14748E7 | Maximum amount of time (in seconds) front panel will wait for input before an error is generated. |
| EOA | Any printable ASCII character; OR any ASCII decimal equivalent enclosed in angle brackets. | <0> (no output) | End-of-Argument character. |
| EOH | Same as EOA | <0> (no output) | End-of-Header character. |
| EOU | Same as EOA | <32> (space) | End-of-Message-Unit character. |

Table D-5
PHYSICAL PARAMETERS FOR "GPIB:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|---|
| MA | 0-30 | 30 | 4041's primary address. |
| SC | YES/NO | YES | States whether or not 4041 is system controller. When set to YES, IFC is pulsed for 100 microseconds, then REN is asserted. |
| PNS | 0-63, 128-191 | 0 | Polled with Nothing To Say. 4041 responds with PNS when polled by C-I-C and not asserting SRQ. |
| IST | TRUe/FALse | FALse | Interface STatus. Value of interface status message, used when parallel polled. |
| DEL | NORmal, FASt | NORmal | GPIB chip T1 delay parameter. Useful with TRA = DMA, when exchanging messages with very fast acceptors. |
| TC | SYN/ASYN | SYN | TC = SYN: take control synchronously on TCT (default); TC = ASY: take control asynchronously on TCT (may corrupt data). |

Table D-6
LOGICAL PARAMETERS FOR "GPIB:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|--|---|--|
| PRI | 0-31 | 31 | 0-30: primary address of device on bus. 31: Interface port itself. |
| SEC | 0-32 | 32 | 0-31: Secondary address of device on bus. 32: No secondary address. |
| EOA | Any printable ASCII character; OR any ASCII decimal equivalent enclosed in angle brackets (< >). | < 44 > (comma) | End-of-Argument character. |
| EOH | Same as EOA | < 32 > (space) | End-of-Header character. |
| EOM | Same as EOA, plus CRLF | < 255 > (CR/LF) | End-of-Message character (EOM = < 0 > means delimit on EOI line only). |
| EOQ | Same as EOA | < 0 > (no output) | End-of-Query character (appended to messages sent with PROMPT clause on INPUT). |
| EOU | Same as EOA | < 59 > (semi) | End-of-Message-Unit character. |
| TIM | Any positive number | 2.14748E7 (open lu); 4 (lu not open prior to I/O) | Data transfer timeout. |
| SPE | Any positive number | 10 msec | Serial Poll Timeout. Amount of time 4041 waits for response from a device before going on to next device or generating Error 818 ("Explicit Poll of Non-existent Instrument"). Used with SELECT statement. |
| TRA | NORmal/FASt/DMA | NOR | Data TRAnsfer Mode. NOR = Normal (interrupt-driven); FAS = Fast (sense status, no proceed mode); DMA = Direct Memory Access (requires Option 1 hardware). |

Table D-7
ASK\$ PARAMETERS FOR "GPIB:" DRIVERS

| Parameter | Possible Values | Default | Comments |
|-----------|------------------|---------|---|
| CIC | 0, 1, 2, 3, 4, 5 | | <p>Controller-in-charge status:</p> <p>0 = power-up state, no bus activity yet.</p> <p>1 = controller idle (4041 in talker/listener mode).</p> <p>2 = controller addressed (TCT command received).</p> <p>3 = controller transferring control (TCT command sent).</p> <p>4 = controller active (ATN message being sent TRUE).</p> <p>5 = controller standby (data transfer in progress; ATN message being sent FALSE).</p> |
| TL | 0-63 | | <p>Talker/listener status. Bit-encoded number, with following values:</p> <p>0 = bus unconfigured.</p> <p>1 = serial poll active; SPE has been sent. (Only valid if 4041 is CIC).</p> <p>2 = 4041 is talk addressed.</p> <p>4 = 4041 is listen addressed.</p> <p>8 = some device is talk addressed (may be 4041; only valid if 4041 is CIC).</p> <p>16 = at least one device is listen-addressed (may be 4041; only valid if 4041 is CIC).</p> <p>32 = parallel poll is active (only valid if 4041 is CIC).</p> |
| ENA | 0-127 | | <p>Enabled GPIB Interrupts. Bit-encoded number, with following values:</p> <p>0 = no interrupts enabled.</p> <p>1 = IFC (received while CIC).</p> <p>2 = SRQ</p> <p>4 = EOI (seen during remote data transfer).</p> <p>8 = MLA and byte available on bus.</p> <p>16 = DCL or SDC</p> <p>32 = MTA and all listeners ready.</p> <p>64 = TCT</p> |
| PEN | 0-127 | | <p>Pending GPIB Interrupts. Same values as ENA.</p> |
| SRQ | 0, 1 | | <p>If controller-in-charge: SRQ = 1 if SRQ message is being received true, else SRQ = 0.</p> <p>If not controller-in-charge: SRQ = 1 if 4041 is sending SRQ message true, else SRQ = 0.</p> |

Table D-8
PHYSICAL PARAMETERS FOR "OPT2:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|--|
| IREg | 0-127 | | Number of register to be written into to turn off "SRQ" interrupt. |
| IVAl | 0-255 | | Number to send to IREg register to turn off "SRQ" interrupt. |

Table D-9
ASK\$ PARAMETERS FOR "OPT2:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|---|
| ICN | any integer | 0 | Number of interrupts sensed since "SRQ" interrupt was last disabled on OPT2 driver. |

Table D-10
LOGICAL PARAMETERS FOR "PRIN:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|--|-----------------|--|
| INDent | Any positive number | 3 | Number of spaces by which successive lines of long output messages (i.e., > 20 characters) are indented. |
| EOA | Any printable ASCII character; OR an?t enclosed in angle brackets. | <0> (no output) | End-of-Argument character. |
| EOH | Same as EOA | <0> (no output) | End-of-Header character. |
| EOU | Same as EOA | <32> (space) | End-of-Message-Unit character. |

Table D-11
LOGICAL PARAMETERS FOR "TAPE:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|-----------------|---------|--|
| VER | YES/NO | YES | Read-after-write verification. |
| DIR | YES/NO | YES | Automatic directory updating when files are closed. WARNING: if DIR = NO, an explicit DISMOUNT statement must be executed when files are closed to force new directory information to be written to the tape. |
| PHY | YES/NO | NO | YES = Tape is in physical mode (I/O via RBYTE, WBYTE only; all files must be closed). NO = Tape is in logical mode (I/O via INPUT, PRINT to files). |
| LON | YES/NO | NO | YES = Print long-form directory when executing DIR command; NO = Print short-form directory when executing DIR command. |

(continued)

Table D-11 (cont)
LOGICAL PARAMETERS FOR "TAPE:" DRIVER

| Parameter | Possible Values | Default | Comments |
|-----------|--|-----------------|--|
| EOA | Any printable ASCII character; OR Any ASCII decimal equivalent enclosed in angle brackets. | <0> (no output) | End-of-Argument character. |
| EOH | Same as EOA | <0> (no output) | End-of-Header character. |
| EOM | Same as EOA | <13> (cr) | End-of-Message character (EOM = <0> means NO EOM). |
| EOU | Same as EOA | <32> (space) | End-of-Message-Unit character. |

Table D-12
LOGICAL PARAMETERS FOR TAPE FILES^a

| Parameter | Possible Values | Default | Comments |
|-----------|---------------------------|-------------------------|--|
| OPEn | NEW, OLD, REPlace, UPDate | OLD | NEW = Write only. Open new file, position tape head at beginning. Error if file already exists on tape. REPlace = Write only. Delete file if already on tape. Open new file. Position tape head at beginning. UPDate = Write only. Open file, position tape head at end of file. Error if file is not already on tape. OLD = Read or write. Open existing file, position tape head at beginning of file. File is open for reading until first write. Writing over-writes data previously in the file after position of last read. Error if file does not already exist on tape. |
| CLIp | YES/NO | NO | YES = Clip files (remove unused space before end-of-file) upon closing. |
| SIze | Any positive number | ½ of largest free block | Maximum size of file. If more data are to be written than the size allows, the tape driver will expand the file, if enough free space is available on tape. |
| FORmat | AScii/ITEm | AScii | Format in which data are stored in file. AScii: Data are stored as ASCII characters. ITEm: Data are stored in 4041 internal representation. |

^a TAPE driver parameters may appear along with the file parameters on the "file spec" side of TAPE stream specs.

Notes on the TAPE Driver Parameters

If VER = YES, a read-after-write verification is done after each record is written to the tape.

If DIR = YES, the tape directory is written out to the tape whenever a file has been created or updated and the file is closed. If DIR = NO, the directory will NOT be written out to the tape.

If PHY = YES, the tape becomes a record-addressable memory, only accessible via RBYTE and WBYTE statements. If PHY = NO, the tape becomes a file-structured storage device.

Notes on the TAPE File Parameters

Tape file names must be unique in the first six letters.

Legal values for SIZE are any integer value corresponding to available bytes of storage on the tape (that is, bytes that are not currently being used by an existing file, and not exceeding the number of bytes available in the largest contiguous collection of bytes on the tape). The default is a value equal to one half the number of bytes available in the largest contiguous collection of bytes on the tape.

Appendix E

INTRODUCTION TO GPIB CONCEPTS

The General Purpose Interface Bus (GPIB) is a digital interface that enables the components of an instru-

mentation system to communicate.

MECHANICAL ELEMENTS OF IEEE-488 STANDARD

CONNECTOR

IEEE Standard 488-1978 specifies a standard connector/cable system for instruments. Standardizing the connectors and cables ensures that standard-conforming instruments are pin-compatible when linked.

The GPIB connector has 24 pins, with 16 assigned to specific signals and eight to shields and grounds.

ALLOWABLE CONFIGURATIONS

Instruments can be connected to the GPIB in "linear" or "star" configurations, or in a combination of both (Figure E-1).

Restrictions

Instruments connected to a single bus cannot be separated by more than two meters for each device on the bus. In addition, the total cable length of the bus cannot exceed 20 meters.

To maintain proper electrical characteristics on the bus, a device load must be connected for every two meters of cable length, and at least two-thirds of the instruments connected to the bus must be powered on. (For further details, consult IEEE Standard 488-1978, Section 6.2.4, "Devices Powered Off and On.")

Although instruments are usually spaced no more than two meters apart, they can be separated further if the required number of device loads is lumped at any point on the bus.

More than 15 devices can be interfaced to a single bus if they do not connect directly to the bus, but are interfaced through a primary device. Such a scheme can be used with programmable plug-ins attached to a central device, where the central device is addressed with a primary address code and the plug-ins are addressed with a secondary address code.

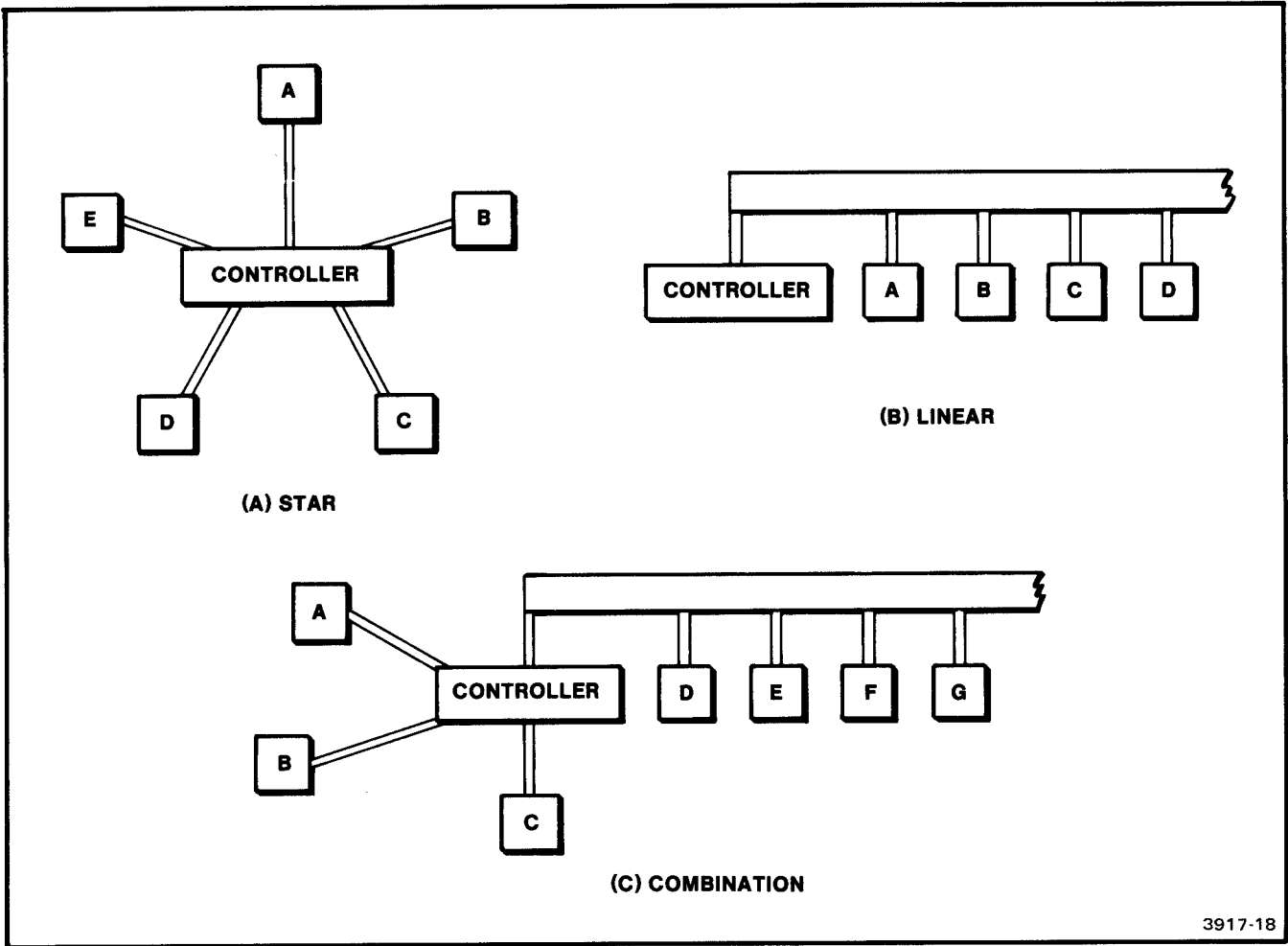


Figure E-1. Allowable Configurations for GPIB Devices.

ELECTRICAL ELEMENTS

The IEEE-488 standard defines the voltages and current values required at connector nodes. All specifications are based on the use of TTL technology. The logical states are defined as follows:

| Logical State | Electrical Signal Levels |
|---------------|--|
| 0 | corresponds to $2.0\text{ V} \leq \text{volts} \leq 5.2\text{ V}$ (called high state) |
| 1 | corresponds to $0\text{ V} \leq \text{volts} \leq 0.8\text{ V}$ (called low state) |

Messages can be sent as either active or passive true signals. Passive true signals occur in the high state and must be carried on a signal line using open collector devices. (Driver requirements are expanded upon in IEEE Standard 488, Section 3.3.)

FUNCTIONAL ELEMENTS: THE 10 INTERFACE FUNCTIONS

Interface functions are the system elements that provide the basic operational facilities through which devices receive, process, and send messages.

Ten different interface messages are defined in all.

The AH (Acceptor Handshake) function provides a device with the capability to guarantee proper reception of remote multiline messages. The AH function delays initiation or termination of a multiline message transfer until the device is ready to receive the next data byte.

The SH (Source Handshake) function works in concert with the AH function on a listening device to guarantee proper transfer of multiline messages. The SH function controls the initiation and termination of the transfer of multiline message bytes.

The L (Listener) and LE (Extended Listener) functions provide a device with the capability to receive device-dependent data over the interface. This capability exists only when the device is addressed to listen. The L function uses a 1-byte address; the LE function used a 2-byte address. In all other respects, the capabilities of both functions are the same.

The T (Talker) and TE (Extended Talker) functions provide a device with the capability to send device-dependent data over the interface. This capability exists only when the device is addressed to talk. The T function uses a 1-byte address; the TE function uses a 2-byte address. In all other respects, the capabilities of both functions are the same.

The DC (Device Clear) function provides a device with the capability to be cleared (initialized), either individually or as part of a group of devices.

The DT (Device Trigger) function provides a device with the capability to have its basic operation started, either individually or as part of a group of devices.

The RL (Remote/Local) function provides a device with the capability to select between two sources of input information. The function indicates to the device that either input information from the front panel controls (local) or corresponding information from the interface (remote) is to be used.

The SR (Service Request) function provides a device with the capability to request service asynchronously from the controller in charge of the interface.

The PP (Parallel Poll) function provides a device with the capability to present a status message to the controller in charge without being previously addressed to talk.

The C (Controller) function provides a device with the capability to send device addresses, universal commands and addressed commands to other devices over the interface. It also provides the capability to determine which devices require service.

ADDRESSES: PRIMARY, TALK, LISTEN, AND SECONDARY

PRIMARY ADDRESSES

Every instrument connected to the bus has a unique primary address. On most devices, this address is set by a system of binary switches located at the rear of the device. The primary addresses of most of these devices can be set by the user.

Some devices, however, have their primary addresses pre-set by the manufacturer. The primary addresses of many devices in this group cannot be changed except by qualified service personnel. If in doubt, check the user's manual for each device.

A "typical" binary switch system is shown in Figure E-2.

Each switch in the system represents a binary digit, with value equal to 1 if the switch is in the "on" position, or 0 if it is in the "off" position. Reading from right to left, the place value of each succeeding switch increases by a factor of two; thus, the rightmost switch has a place value of 1, the second switch from the right a place value of 2, the third a place value of 4, and so on.

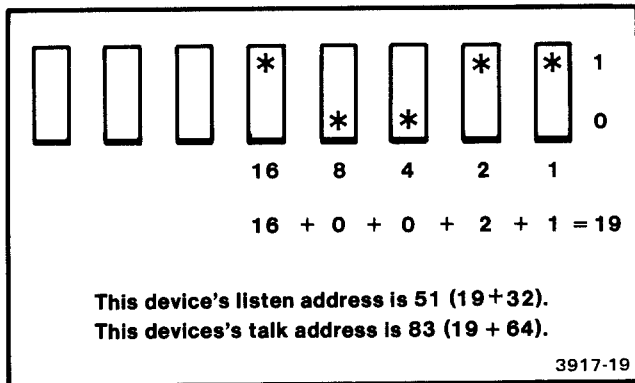


Figure E-2. "Typical" Binary Switch System, With Address Set to 19.

To determine the value of the address represented by a given switch setting, simply add up the place values of all the switches in the "on" position. In the example shown, switches with place values of 16, 2, and 1 are in the "on" position. Since $16 + 2 + 1 = 19$, an instrument on the GPIB with switches set in these positions would have a primary address of 19.

No two instruments on the GPIB can have the same primary address. Primary addresses can vary from 0 to 30.

LISTEN ADDRESSES

A device's Listen Address is determined by adding 32 to its Primary Address. Thus, a device with Primary Address 19 has a Listen Address of $19 + 32 = 51$.

When a device senses its Listen Address being sent over the data lines while the ATN line is asserted, the device prepares itself to "listen" (i.e., accept data sent over the data lines) when ATN becomes unasserted.

Any number of devices on the bus may be addressed to listen at the same time.

TALK ADDRESSES

A device's Talk Address is determined by adding 64 to its Primary Address. Thus, a device with Primary Address 19 has a Talk Address of $19 + 64 = 83$.

When a device senses its talk address being sent over the data lines while the ATN line is asserted, the device prepares itself to "talk" (i.e., send data over the data lines) when ATN becomes unasserted.

Only one device on the bus may be addressed to talk at one time. When a device previously addressed to talk senses another device's talk address being sent over the data lines with ATN asserted, the first device automatically "un-talks" itself.

SECONDARY ADDRESSES

Some devices support a special addressing scheme called "secondary addressing," which uses addresses in the range from 96 to 126. Not all IEEE-488 compatible devices, however, support secondary addressing.

Among devices that do support secondary addressing, different manufacturers implement it in different ways. In fact, different devices from the same manufacturer may implement secondary addressing in different ways.

For example, for some devices secondary addressing is an addressing method that allows two or more devices to share the same primary address.

For other devices, however, secondary addressing is a method of transmitting certain device-dependent commands to a device.

There are several other variations of secondary addressing implementations. When in doubt, check the manual for the device.

DATA, MANAGEMENT, AND "HANDSHAKE" BUSES

The 16 signal lines can be divided into three "buses" (Figure E-3) as follows:

Eight signal lines make up the data bus, which carries the data to be transferred on the GPIB; five lines make up the management bus, used to control the data transfers; and three lines make up the transfer or "handshake" bus, used to synchronize data transfers between instruments.

DATA BUS

The data bus contains eight bidirectional signal lines, numbered DIO1 through DIO8. One byte of information (eight bits) is transferred over the bus at a time. DIO1 carries the least significant bit of the byte; DIO8 carries the most significant bit.

Each byte of information transferred over the data bus represents either a command, a device address, or a device-dependent message. Data bytes can be formatted in ASCII code or in device-dependent binary code.

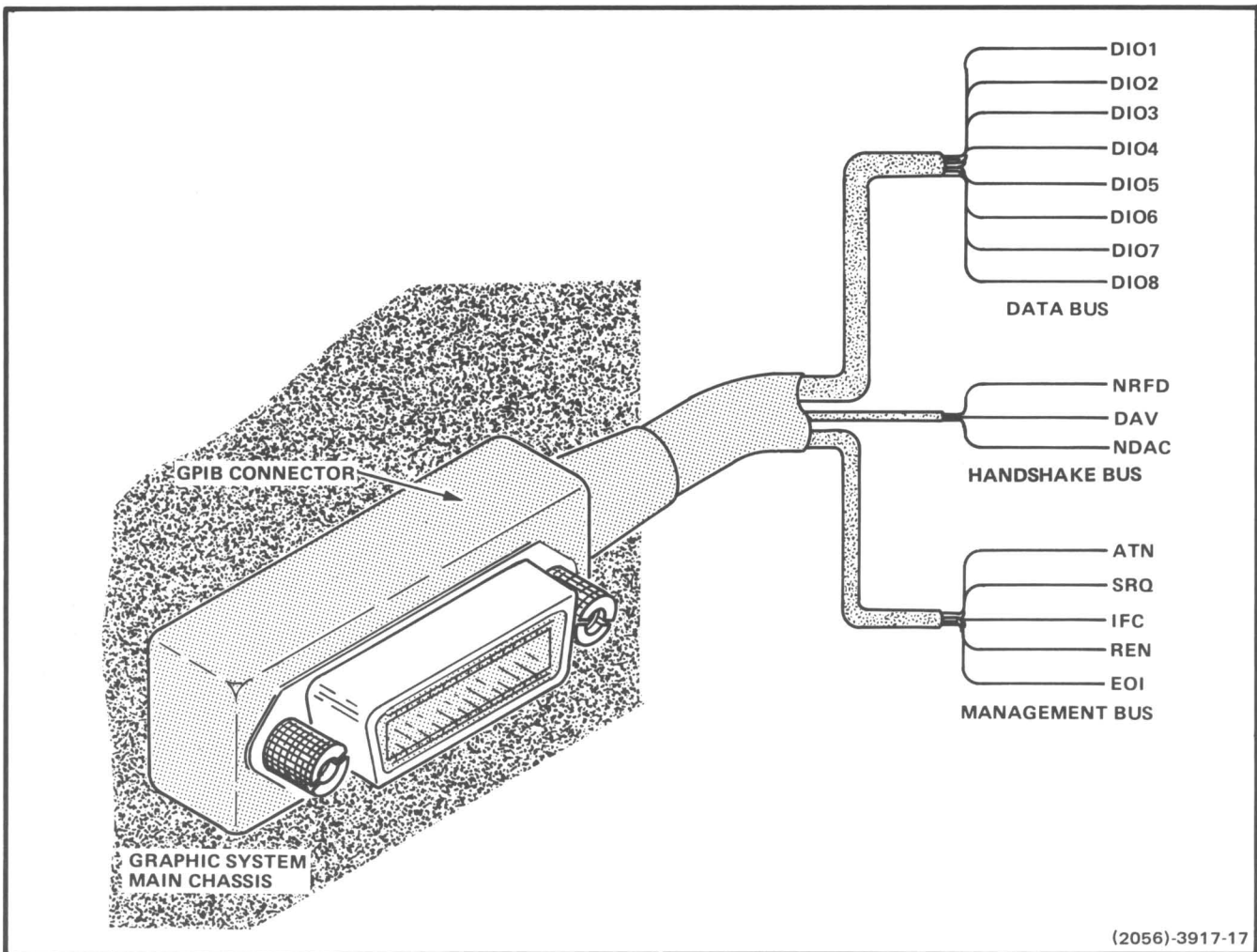


Figure E-3. Data, Management, and "Handshake" Buses.

MANAGEMENT BUS

The Management Bus is a group of five signal lines (ATN, EOI, IFC, REN, and SRQ) used in managing data transfers over the data bus. Signal definitions for each line follow.

| Line | Definition |
|-------------|--|
| ATN | <p>The ATN (Attention) management line is activated by the controller to send universal and addressed commands, and to designate peripheral devices as talkers and listeners for an upcoming data transfer.</p> <p>When ATN is asserted, messages sent over the data bus are interpreted as commands or addresses.</p> <p>When ATN is unasserted, messages sent over the data bus are interpreted as device-dependent messages. Only peripheral devices addressed by the controller to talk or listen (with ATN asserted) take part in a device-dependent data transfer when ATN becomes unasserted.</p> |
| EOI | <p>The EOI (End-Or-Identify) signal line can be used by any talker to indicate the end of a data transfer sequence. Talkers that use EOI activate the EOI line as the last byte of information is being transmitted.</p> |
| IFC | <p>The IFC (InterFace Clear) line is activated by the system controller to (1) unaddress all talk-addressed and listen-addressed devices on the bus, (2) take controller-in-charge status, and (3) take all devices out of serial poll mode (same as sending SPD). Only the system controller can activate this signal line.</p> |
| REN | <p>The REN (Remote Enable) signal line is activated by the system controller to give all devices on the bus the capability of being placed under remote (program) control.</p> <p>When the REN signal line is activated, devices receiving their listen addresses over the data bus will accept and execute commands from the controller-in-charge.</p> |

| Line | Definition |
|-------------|--|
| | <p>When REN is de-activated, all devices on the bus revert to front-panel control.</p> |
| SRQ | <p>The SRQ (Service Request) line can be activated by any device on the bus to request service from the controller. The controller responds (if programmed to do so) by serial polling all devices on the bus in order to find the device requesting service. The SRQ line is de-activated when the device requesting service is polled.</p> |

TRANSFER OR “HANDSHAKE” BUS

Three lines (NRFD, DAV, and NDAC) make up the transfer or “handshake” bus. These three lines control the sequence of operations each time a byte is transferred over the data bus. This sequence is NOT under user control, but information about the three transfer lines is presented here for completeness.

| Line | Definition |
|-------------|--|
| NRFD | <p>An active NRFD (Not Ready For Data) line indicates that one or more listeners are not ready to receive the next data byte. When the NRFD line goes inactive, the talker places the next data byte on the data bus and activates the DAV signal line.</p> |
| DAV | <p>The DAV (Data Valid) line is activated by the talker shortly after the talker places a valid data byte on the data bus. This tells each listener to capture the data byte currently on the bus.</p> |
| NDAC | <p>The NDAC (Not Data ACcepted) line is held active by each listener until the listener captures the data byte currently on the data bus. When all listeners have captured the data byte, NDAC goes inactive. This tells the talker to take the byte off the data bus.</p> |

GPIB COMMUNICATIONS PROTOCOL: CONTROLLERS, TALKERS, AND LISTENERS

We can think of the 16 data, management, and "handshake" lines as a "common" area that all devices on the bus monitor constantly.

Each device on the bus at any given time may be either a talker, a listener, or a controller. Some instruments have two or even all three of these capabilities, i.e., some instruments are talk-only, others are listen-only, others can talk and listen, others can talk, listen, and control.

Talkers are instruments that have the capability to put data out onto the eight data lines. Once on the data lines, data can be read by any active listener. Only one device is allowed to talk at one time, to eliminate possible confusion.

Listeners, conversely, are instruments that can read information from the data lines. Any number of devices can be listening to the talker at the same time. The rate at which the talker can put information onto the data lines is restricted, by means of the "three-wire handshake" sequence, to the rate at which the slowest active listener on the bus can accept data.

Controllers are devices that assign talk and listen status to other devices on the bus. Since not every device can be allowed to talk at the same time, and it is seldom desirable for every device to listen, we need a controller to designate which device is to talk and which are to listen during any data transfer.

Controllers and talker/listeners each have special properties and requirements that programmers must remember in writing programs to control instrument networks. The next two subsections deal with the special properties and requirements of controllers and of talker/listeners, respectively.

CONTROLLERS

Two kinds of controllers are allowed on the GPIB: (1) a system controller, and (2) a controller-in-charge.

At any time, a GPIB network can have at most one instrument acting as system controller and one instrument acting as controller-in-charge. The system controller and the controller-in-charge may be the same instrument.

The configuration may, however, include any number of instruments capable of acting as system controller or controller-in-charge, subject only to the limitations on the total number of instruments allowed on the bus.

System control cannot be passed from instrument to instrument; controller-in-charge status can. Once an instrument becomes system controller, no other instrument on the bus can assume that role without (essentially) powering down the entire system and powering it up again.

Only the system controller can affect the status of the InterFace Clear (IFC) and Remote ENable (REN) management lines. Asserting the IFC line makes the system controller the controller-in-charge, and untalks, unlistens, and disables serial polling for all devices.

Asserting the REN line enables instruments on the bus to be remotely operated via the controller. (NOTE: Asserting REN does not automatically put all instruments on the bus into "Remote State" (REMS), but simply allows the controller to put them into this state.)

Any instrument acting as controller-in-charge may pass control to any other instrument on the bus capable of assuming control.

The controller-in-charge is the only instrument on the bus capable of sending messages with the ATN line asserted. Messages sent with ATN asserted are interpreted as either primary addresses, secondary addresses, universal commands, or addressed commands.

To summarize:

1. Any GPIB network can have at most one system controller and one controller-in-charge at one time. The system controller and the controller-in-charge may be the same instrument.
2. Controller-in-charge status can be passed; system control cannot.
3. The system controller can take controller-in-charge status from another controller on the bus by asserting the IFC (InterFace Clear) line.
4. The controller-in-charge is the only device that can send messages with the ATN line asserted.

TALKERS

A talker is a device that has sensed its talk address being sent over the data lines with the ATN line asserted.

After a device has been addressed to talk, it will send data over the data lines as soon as the ATN line becomes un-asserted.

Only one device on the bus may be addressed to talk at one time.

When a device previously addressed to talk senses another device's talk address being sent over the data lines with ATN asserted, the first device automatically "un-talks" itself (i.e., will not put data out over the data lines when ATN becomes unasserted).

Devices may also "un-talk" themselves when they sense their listen addresses being sent over the data lines with ATN asserted. In this case, the device becomes a listener as soon as the ATN line is un-asserted.

LISTENERS

A listener is a device that has sensed its listen address being sent over the data lines with the ATN line asserted.

After a device has been addressed to listen, it will accept data coming over the data lines as soon as the ATN line becomes un-asserted.

Any number of devices on the bus may be addressed to listen at the same time.

When a device previously addressed to listen senses its talk address being sent over the data lines with ATN asserted, the device may "un-listen" itself and become a talker as soon as the ATN line is un-asserted.

UNIVERSAL COMMANDS

Universal commands are values the controller sends over the data lines with ATN asserted. These commands are obeyed by all devices on the bus with the appropriate subsets (see below).

The universal commands include DCL (Device Clear), LLO (Local LockOut), PPU (Parallel Poll Unconfigure), SPD (Serial Poll Disable), and SPD (Serial Poll Disable).

Also included in this description are UNL (Unlisten) and UNT (Untalk). Although not "commands" in the strictest sense, the values of UNL and UNT act like universal commands when sent over the data lines with ATN asserted.

DCL (DEVICE CLEAR)

To send the DCL (Device Clear) command, the controller sends a value of 20 over the data lines with ATN asserted.

The DCL command "clears" (initializes) all devices on the bus that have a DC1 or DC2 subset of the DC interface function.

LLO

To send the LLO (Local LockOut) command, the controller sends a value of 17 over the data lines with ATN asserted.

The LLO command "locks out" the front panels of all devices on the bus that have an RL1 subset of the RL interface function. (Devices with RL0 or RL2 subsets of the RL interface function ignore LLO.) After receiving the LLO command, all devices ignore any subsequent inputs from front panel keys that have corresponding remote controls, and only obey commands coming to them through the GPIB interface.

PPU

To send the PPU (Parallel Poll Unconfigure) command, the controller sends a value of 21 over the data lines with ATN asserted.

The PPU command unconfigures all devices on the bus for parallel polling.

SPD

To send the SPD (Serial Poll Disable) command, the controller sends a value of 25 over the data lines with ATN asserted.

The SPD command returns all devices on the bus from the Serial Poll Enabled state.

SPE

To send the SPE (Serial Poll Enable) command, the controller sends a value of 24 over the data lines with ATN asserted.

The SPE command puts all devices on the bus with an SR1 subset of the SR interface function into the Serial Poll Enabled state. In this state, each device will send the controller its status byte when the device receives its talk address over the data lines with ATN asserted.

UNL

To send the UNL (Unlisten) "command", the controller sends a value of 63 over the data lines with ATN asserted.

The UNL command takes all listen-addressed devices on the bus out of the listen-addressed state.

UNT

To send the UNT (Untalk) "command", the controller sends a value of 95 over the data lines with ATN asserted.

The UNT command takes any talk-addressed device on the bus out of the talk-addressed state.

ADDRESSED COMMANDS

Addressed commands are values the controller sends over the data lines with ATN asserted that are intended for specific devices.

These commands include GET (Group Execute Trigger), GTL (Go To Local), PPC (Parallel Poll Configure), SDC (Selected Device Clear), and TCT (Take Control).

All the above commands except TCT require that the device receiving the command be listen-addressed. The TCT command requires that the device be talk-addressed.

GET

To send the GET (Group Execute Trigger) command, the controller sends a value of 8 over the data lines with ATN asserted.

The GET command causes all listen-addressed devices incorporating a DT1 subset of the DT interface function to start their basic operation (e.g., measurement devices to make their measurements, output devices to output their signals, etc.).

GTL

To send the GTL (Go To Local) command, the controller sends a value of 1 over the data lines with ATN asserted.

The GTL command causes all listen-addressed devices to obey incoming commands from their front-panel control buttons. These devices may store, but not respond to, commands coming through the GPIB interface until the device is once again listen-addressed.

PPC

To send the PPC (Parallel Poll Configure) command, the controller sends a value of 5 over the data lines with ATN asserted.

The PPC command enables a listen-addressed device incorporating a PP1 or PP2 subset of the PP interface function to respond to a parallel poll.

PPE

The PPE command is used to designate the sense and the DIO line on which devices incorporating a PP1 subset of the PP interface function will respond to a parallel poll. If the device's individual status bit matches the assigned sense at the time the parallel poll is executed, the device asserts its assigned data line.

Devices incorporating a PP2 subset of the PP interface function have their senses and DIO lines hardwired; such devices cannot be configured by the controller to respond to a parallel poll. They may be enabled or disabled for parallel poll by the controller.

PPD

The PPD message unconfigures a previously configured device from responding to a parallel poll. (PPD acts as a "local" PPU message.)

PPU

The PPU command unconfigures all previously configured devices on the bus from responding to a parallel poll.

SDC

To send the SDC (Selected Device Clear) command, the controller sends a value of 4 over the data lines with ATN asserted.

The SDC command "clears" (initializes) all listen-addressed devices.

TCT

To send the TCT (Take Control) command, the controller sends a value of 9 over the data lines with ATN asserted.

The TCT command passes controller-in-charge status to a talk-addressed device.

SERIAL POLLING

STATUS BYTES

Each device on the GPIB incorporating the SR1 subset of the SR interface function has an eight-bit "status byte". The status byte's contents describe (by means of a device-dependent code) the device's status.

REQUESTING SERVICE

The "coding" of a device's status byte is almost entirely up to the device designer, with one restriction: bit 7 (the second-most-significant bit) is reserved to indicate whether or not a device is requesting service from the controller-in-charge.

Bit 7 of the status byte is known as the "rsv", or "requesting service", bit. A value of 1 in this bit indicates that a device is requesting service from the controller-in-charge. A value of 0 in this bit indicates that the device is not requesting service.

Devices that require service do two things in order to request it: (1) set the rsv bit of the status byte to "1"; and (2) assert the SRQ line.

CONDUCTING SERIAL POLLS

When the controller senses a device on the bus asserting the SRQ line, it customarily generates an interrupt and serially polls each device on the bus, in order to find out which device is requesting service.

(NOTE: A device need not actually be requesting service for the controller to initiate a serial poll.)

The controller conducts the poll by sending out the SPE (Serial Poll Enable) command, followed by a sequence of listen addresses. As each listen address is sent, the device that is listen-addressed sends its status byte to the controller. The controller then checks the status byte to see if the rsv bit is set. The act of receiving the status byte from the device requesting service "clears" the rsv bit of that device (i.e., sets it back to 0).

When the device asserting SRQ is discovered, the controller usually terminates the serial poll (by sending out the SPD command), and transfers control to a user-defined handler for the device requesting service. The act of reading the device's status byte clears the device's rsv bit; however, the factors that caused the device to request service in the first place must be handled, or the device will simply request service again and again.

PARALLEL POLLING

Parallel polling is a means of simultaneously reading the individual status messages of all devices configured to respond to a parallel poll.

INDIVIDUAL STATUS MESSAGES

All devices with a PP function have an individual status message. This message is capable of being set to “true” or “false”, either remotely (i.e., by the controller) or locally (i.e., by the device itself), depending on the device’s design.

CONFIGURING THE BUS FOR PARALLEL POLL

A series of PPC commands is used to configure the bus for parallel polls. “Configuring” the bus means designating which devices are to respond to parallel polls, which data line each device is to respond on, and on which “sense” of its individual status message the device is to assert its assigned data line. (Devices incorporating the PP2 subset of the PP interface function are configured locally, meaning that their sense and data line is set by the manufacturer, and cannot be changed by the controller.)

CONDUCTING THE PARALLEL POLL

The controller conducts a parallel poll by asserting the ATN and EOI lines simultaneously.

When the devices configured for parallel polling sense the ATN and EOI lines asserted simultaneously, each device checks its individual status message and asserts its assigned line if the message sense matches the sense assigned for the device to assert its line.

The result of the parallel poll is a bit-encoded integer that the controller can use to calculate which devices asserted their data lines during the parallel poll.

Appendix F

INTRODUCTION TO TEK CODES AND FORMATS FOR GPIB

INTRODUCTION

As their measurements grew more numerous and complex, instrument users realized their need for instruments that could be combined into interactive and automated systems that would:

1. Reduce labor costs;
2. Increase the effective use of research and design skills by freeing people for creative work;
3. Provide insight into products and processes by coupling analysis with measurements; and
4. Reduce human errors in applications requiring precise, repeated measurements.

Meeting these needs would require instruments, controllers, and peripheral devices that would be “compatible” — easy to use with each other.

The first major step toward device compatibility was taken in 1975 when the IEEE published the 488 standard, defining an interface for programmable instruments. This interface is usually called the GPIB, or General Purpose Interface Bus.

Before GPIB, connecting programmable instruments to a computer or to a desktop calculator was a major job because each instrument's interface was different. The IEEE-488 standard defines an interface that makes it much easier to put together computer-controlled instrument systems.

The IEEE-488 standard defines three aspects of an instrument's interface:

1. mechanical — the connector and the cable;
2. electrical — the electrical levels for logical signals and how the signals are sent and received;
3. functional — the tasks that an instrument's interface is to perform, such as sending data, receiving data, triggering the instrument, etc.

Using this interface standard, instruments can be designed to have a basic level of compatibility with other instruments that meet the standard. However, this is only the first step toward complete compatibility.

Tektronix has taken the next step by adopting a new standard called the Tek Codes and Formats for GPIB. It is intended to:

1. define device-dependent message formats and codings and thus enhance compatibility among instruments that comply with IEEE-488;
2. reduce the cost and time required to develop system and application software by making it easier for people to generate and understand the necessary device-dependent coding.

Beyond the Codes and Formats standard, there is also a need for a philosophy of designing instruments to be friendly to the user. They should be controlled over the bus with easily understood commands and should be resistant to operator errors. Since the application of this philosophy is different for each type of instrument, it is not included as a specific standard.

COMPATIBILITY

Using the GPIB is like using the telephone system. In both cases, a physical connection can be established and data can be transmitted, i.e., one person or one device can talk to another.

However, on the telephone system, unless both people speak and understand the same language, very little communication can take place. Beyond having a common language, they must also share a common vocabulary.

Similar problems can arise between instruments that exchange data.

The IEEE-488 standard defines a "telephone system" describing how the physical communications system is to be used, but it does not define the "language" sent over the bus. This can cause incompatibilities, even among devices that meet the standard.

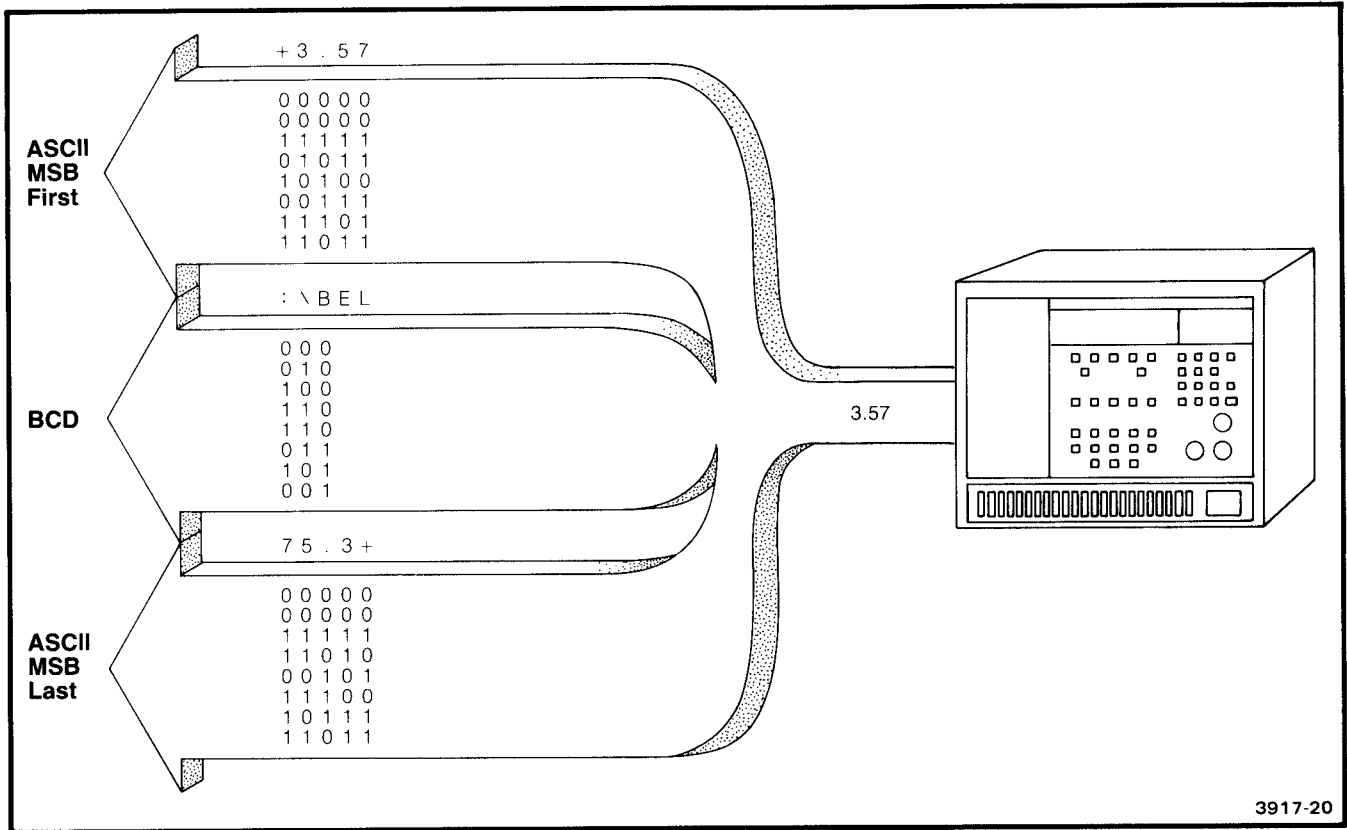
For example, suppose a digital multimeter (DMM) has made a measurement of + 3.75 volts and now has to transmit this information over the GPIB to a computer. Eight data lines are available to send information in byte-serial fashion.

The first question is: "What codes should be used to encode the five characters?" The 488 standard recommends ASCII code, but the DMM designer is free to choose any available one. If BCD is selected but the computer only understands ASCII, the DMM and the computer will be incompatible, even though both devices meet the IEEE-488 standard.

Suppose the DMM does send ASCII code. The question now is: "What format is the data to be in?" Does the DMM send the character sequence + 3.75 most-significant-byte first, least-significant-byte first, or some other way?

Again, the IEEE-488 standard says nothing, and the designer is free to create incompatibility. Figure F-1 shows three possible formats for transmitting data from a DMM.

For system products, it is important that designers use a common format. Thus, the Tek Codes and Formats Standard specifies that instruments are to send ASCII data with the most significant byte first. This makes it easier for users to configure instruments into systems.



3917-20

Figure F-1. Three Valid Data Representations for the Same Data on the GPIB. Tektronix Codes and Formats Standardizes on ASCII Code With the Most Significant Byte First.

HUMAN INTERFACE

People designing measurement systems must be intimately involved with the instrument-to-instrument communication process.

People designing instruments should design them to communicate with one another in ways that can be easily understood by the people designing measurement systems.

For example, suppose a GPIB-programmable power supply needs to be set to 20.0 volts. The power supply can be designed in one of two basic ways: easy on the designer or easy on the user.

The first way is to design the instrument with minimal intelligence, so that it can accept some "hieroglyphics", which it can in turn interpret and execute.

Some power supplies must receive the ASCII character sequence "08E3" in order to put out 20 volts. The "0"

stands for the 0-to-36 volt range, and the "8E3" is the ASCII representation in hexadecimal of the machine-language instructions required to carry out the command.

The second way is to design the instrument with a microprocessor and intelligence to accept easily understood numbers. A power supply designed this way would output 20 volts when it received the character sequence "VPOS 20".

This second method of interacting with the user is obviously a great deal more convenient for people, not only when the program is first written but also later, when someone other than the original programmer has to find out what the program is supposed to do.

In the future, most instruments will be "intelligent" and designed to interact with people. The Tek Codes and Formats standard promotes this type of instrument "friendliness".

REPRESENTING NUMBERS

Because most GPIB instruments use ASCII-code characters to send and receive data, Tektronix has chosen ASCII coding as standard.

In addition, most instruments that send or receive numbers use the ANSI X3.42 standard format. This format states in effect that there are three types of numbers — integers, reals, and reals with exponents — and that they should be sent with the most significant character first. Table F-1 shows examples of these formats.

Unless there are numeric needs that cannot be met by this standard format, present and future Tektronix instruments will also use this format.

Table F-1

NUMBER FORMATS (ANSI X3.42)

| Format | Example | Notes |
|--------|--|---|
| NR1 | 375 + 8960 - 328 + 00000 | Value of "0" must not contain a minus sign. |
| NR2 | + 12.589 1.37592 -00037.5 0.000 | Radix point should be preceded by at least one digit. |
| NR3 | -1.51E+ 03 + 51.2E-07 + 00.0E+ 00 | Value of "0" must contain NR2 zero followed by a zero exponent. |

DEVICE-DEPENDENT MESSAGE STRUCTURE

A message represents a given amount of information whose beginning and end are defined. It is communicated between a device functioning as a talker and one or more devices functioning as listeners.

The Tek Codes and Formats Standard defines the structures of device-dependent messages as follows:

A message begins when the ATN line becomes unasserted after the transmitting device is addressed to talk and the receiving device(s) are addressed to listen.

A message is composed of one or more message units separated by message unit delimiters (semicolons).

A message ends when the talking device asserts the EOI line.

There are two message unit types: Mixed Data Message Units, and Query Message Units.

MIXED DATA MESSAGE UNITS

There are two acceptable formats for mixed data message units: Header Format, and Non-Character Argument Format.

Header Format

A header-format message unit consists of a "header", or sequence of ASCII characters describing the contents of the message unit, followed by a space, followed by a sequence of arguments applying to the header (separated by commas, if more than one argument).

Header format message units are usually used to transfer programming information about a device. For example, sending the message "FUNC SINE;OUT ON" to an FG5010 function generator causes the function generator to start outputting ("OUT ON") a sine wave ("FUNC SINE") at its current settings.

Non-Character Argument Format

A message unit in Non-Character Argument format consists of a sequence of non-character arguments, separated by commas.

Non-Character Argument format is usually used to transfer measurement data. There are six non-character argument types: number arguments; string arguments; ISO block arguments; binary block arguments; end block arguments; and link arguments.

Table F-2 lists and gives examples of the various non-character argument types, and gives the definition and purpose of each.

QUERY MESSAGE UNITS

A query message unit consists of a character argument such as "SET", "ID", or "FREQ" followed by a question mark. Query message units are normally used to interrogate a device for data or settings.

Table F-2

NON-CHARACTER ARGUMENTS

| Type (Example) | Definition | Purpose |
|----------------------------|--|--|
| Number (-12.3) | Numeric value in any of the formats shown in Table F-1. | Used to pass numeric values in ASCII. |
| String ("Remove Probe") | Opening delimiter (single or double quote) followed by a series of any ASCII characters except opening delimiter, and a closing delimiter identical to opening delimiter. | Provides a means for transmitting ASCII text to an output device. |
| Binary Block | "%" followed by two-byte binary integer arrays of numeric data specifying number of data bytes to be transmitted plus 1 for a checksum byte. | Used to transfer large arrays of numeric data such as waveforms in binary format. |
| End Block | "" followed by a block of data, with EOI set concurrent with last byte. End block can only be the last argument in a message and cannot be followed by a message unit delimiter. | Used when a block of data must be sent and neither the amount of data nor its format is known. |
| Link Argument (NR.PT:1024) | Character argument (label) followed by ":" and a value in one of the above argument types. | Used to attach a name or label to another argument. |

MESSAGE CONVENTIONS

While standardizing the “language” used on the bus fosters greater compatibility between devices, it alone does not solve all compatibility problems. Well-defined operational conventions are also needed.

END-OF-MESSAGE

For example, both talking and listening devices should agree on when a message ends. Obvious difficulties can arise when talker and listener don't agree; if the listener thinks the message has ended too soon, it will miss part of the message, while if it doesn't think the message is ended when it actually has, the listener will “hang” the bus waiting for a message that will never come.

Other complications can arise as well. Figure F-2 illustrates what can happen if a talking device (a DMM) delimits messages by transmitting carriage-return/line-feed while the listening device (in this case, the controller) delimits on carriage-return alone.

When the DMM sends the character sequence “+ 3.75<cr> <lf>”, the controller receives only the characters “+ 3.75<cr>”, leaving the “<lf>” in the DMM's output buffer.

When the DMM sends another character sequence, the controller receives the “<lf>” character first. The controller doesn't know what to make of a number preceded by a line feed character, so it stops and indicates an error.

Figure F-3 illustrates another complication. Suppose the listening device understands <cr><lf> as a message terminator. Further suppose that the talking device is sending binary data, as many devices do that have to transmit large amounts of data. If the talking device sends a sequence that coincidentally has the same binary values as <cr><lf>, the listener will stop listening. Any incoming data after that point are lost.

Use of the EOI line as a message terminator avoids such problems. The Tek Codes and Formats Standard specifies that talking devices are to assert the EOI line concurrent with the last byte of their messages. Listening devices are to understand the EOI line's becoming asserted to signify that the last byte of a message has been transmitted.

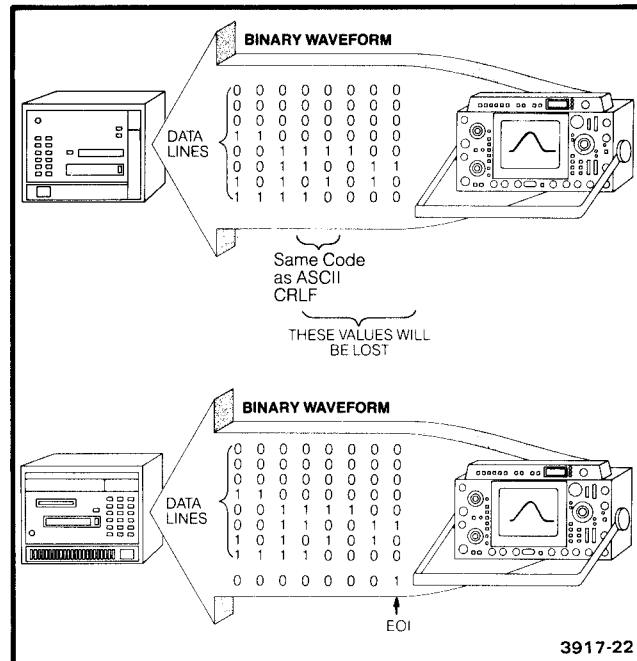
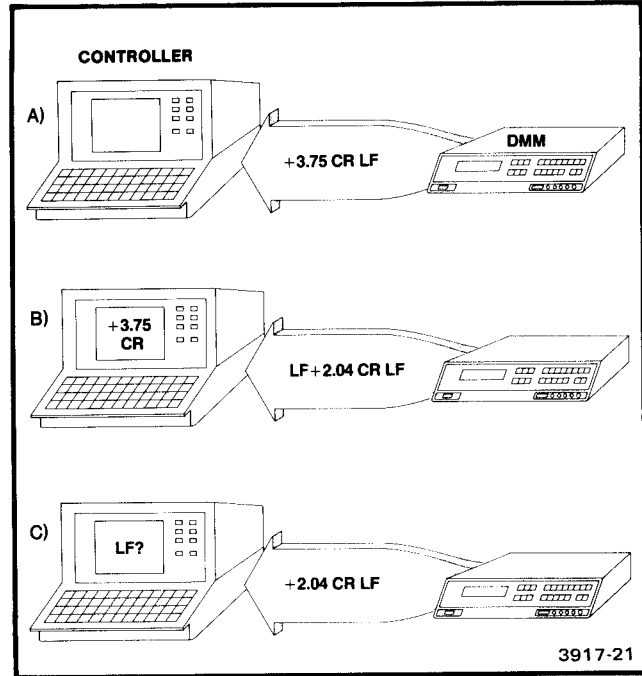


Figure F-3. Problem: Sending Binary Data, Using <CR><LF> as Message Terminator.

STATUS BYTES

The IEEE-488 standard defines a facility for an instrument to send a byte of status data to the computer, but, except for bit 7, the standard does not define the meaning of the bits. (The IEEE-488 standard defines bit 7 to tell whether or not a device is requesting service.)

However, there is a common need for instruments to report certain kinds of status or errors to the controller, so the Tek Codes and Formats Standard establishes a status byte convention to do this.

One common need is for instruments to report if they are processing or executing a command, or ready to receive another command. Bit 5 is used for this purpose.

Another common need is for instruments to report if they are encountering abnormal conditions. Examples of abnormal conditions are internal error conditions within the device functions, erroneous program data sent to a device, incomplete or erroneous measurement data, or device-dependent limit or alarm conditions. Bit 6 of the status byte indicates abnormal conditions. There are more complex conditions besides busy/ready or normal/abnormal. These are listed in Table F-3.

Certain instruments may have conditions that are peculiar to them. To report these status states, bit 8 is used to indicate that the status byte is not the common type but particular to the instrument.

Providing a standard coding for the status byte enhances the convenience to the person programming the system. If all the instruments have common status byte codings, then a common status byte handling routine can be written for all instruments, instead of a separate one for each.

Table F-3
STATUS BYTE DEFINITIONS

| Conditions | Binary | Decimal | |
|---|--------------|---------|-------|
| | | X = 0 | X = 1 |
| Abnormal | | | |
| ERR query requested | 011X 0000 | 96 | 112 |
| Command error | 011X 0001 | 97 | 113 |
| Execution error | 011X 0010 | 98 | 114 |
| Internal error | 011X 0011 | 99 | 115 |
| Power fail | 011X 0100 | 100 | 116 |
| Execution error warning | 011X 0101 | 101 | 117 |
| Internal error warning | 011X 0110 | 102 | 118 |
| Normal | | | |
| No status to report out of the ordinary | 000X 0000 | 0 | 16 |
| SRQ query request | 010X 0000 | 64 | 80 |
| Power on | 010X 0001 | 65 | 81 |
| Operation complete | 010X 0010 | 66 | 82 |

QUERIES

Even with all the possibilities allowed by the status bytes, it is often necessary to send more detailed information from an instrument to a computer. This can be done via “queries”.

Queries take the form of a header followed by a question mark (see Figure F-4). Here are some queries and their uses:

- **ERR?** is used for investigating detailed error conditions in an instrument. The response an instrument sends back is ERR followed by NR1 numbers that code the particular problem.
- **SET?** requests an instrument to send the controller its present settings and other current state information. Sending this information back to the instrument at a later time returns the instrument to the state it

was in when queried. This query makes it possible to develop a program using an instrument’s front panel as input to the computer. Using this feature, a programmer never needs to know the instrument’s GPIB commands.

- **ID?** makes an instrument identify itself by sending such information as its instrument type, model number, version of firmware, etc. This feature is useful for identifying a particular device in the field and potentially for self-configuring systems.

Defining a standard way to elicit responses from an instrument enhances the convenience to the system designer. When all instruments in a system use the same form to perform similar functions, the designer has to learn only one convention, not many.

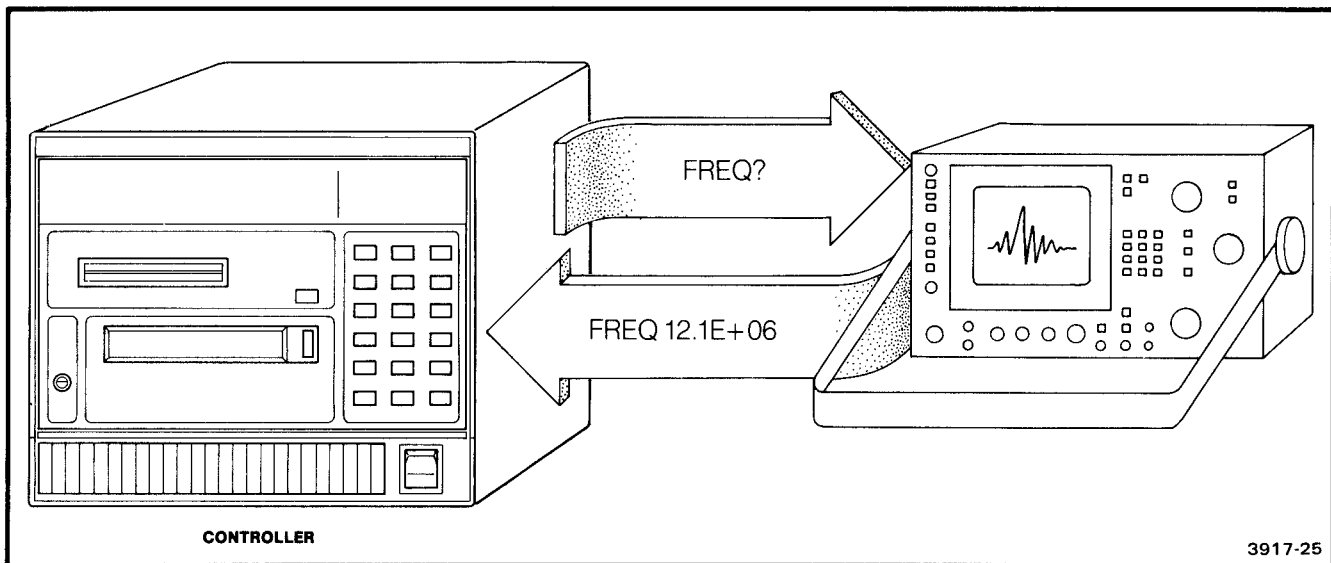


Figure F-4. Query Commands Are Formed by Adding a Question Mark to the Mnemonic for the Setting to be Queried.

ADDITIONAL FEATURES

Besides standardizing the language that instruments use to communicate, the Tek Codes and Formats Standard specifies certain instrument characteristics that guarantee maximum friendliness and dependable operation. Here are some examples:

1. An instrument always says something when made a talker. If it has nothing to say, it sends a byte of all ones concurrent with EOI. This lets the listening device know that no meaningful data is forthcoming, and prevents tying up the GPIB while one device waits for another to talk.
2. A listening device always handshakes. It does not stop handshaking just because it doesn't understand or can't execute a message. After EOI is received, if the device is confused, it sends out a service request and, when serial polled, notifies the controller that the command cannot be executed as sent (Figure F-5). **UNDER NO CIRCUMSTANCES DOES A DEVICE EXECUTE A MESSAGE IT DOES NOT UNDERSTAND.**
3. Instruments always send numbers in correct NR1, NR2, or NR3 formats, but receive numbers "forgivingly", e.g., they accept values such as "-0" or NR3 numbers without decimal points.
4. If an instrument receives a number whose precision is greater than the instrument can handle internally, the number is rounded off (not truncated) to enhance accuracy.
5. Instruments recognize both spaces and commas as argument delimiters. Multiple spaces or commas are NOT construed as delimiters for null arguments.
6. Instruments receive both characters and arguments in upper and lower case and equate them, e.g., a = A, b = B, etc. This is important, because some computer terminals cannot send both upper and lower case characters.
7. An instrument sending data about its front panel uses headers and character arguments that correspond to the front panel's nomenclature.

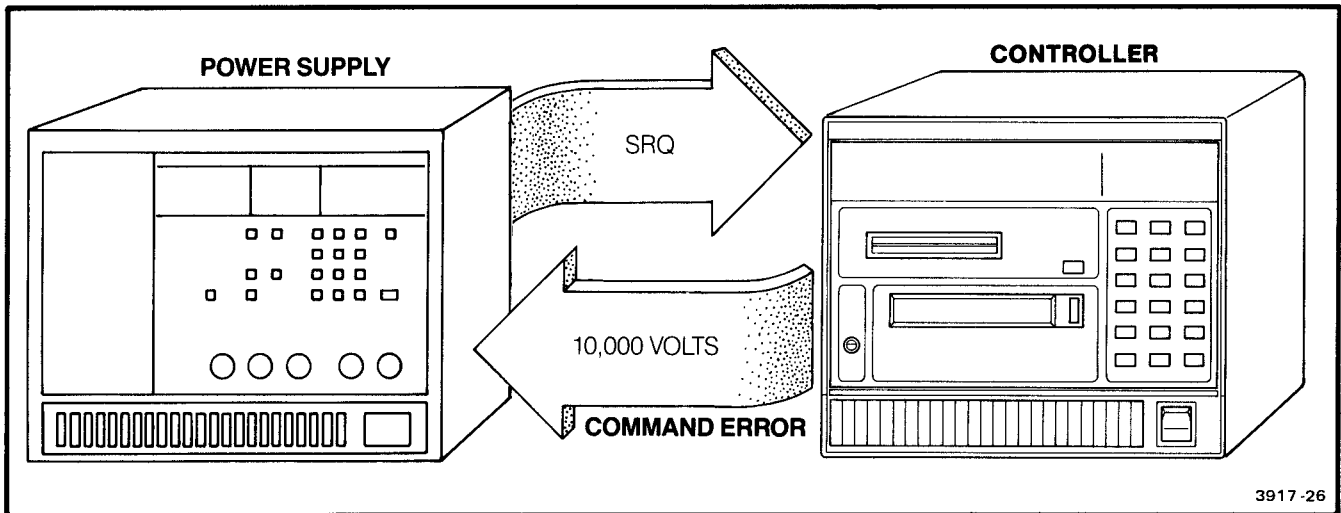


Figure F-5. Devices Should Assert SRQ When an Illegal Command is Received.

INDEX

- ABORT Interrupt Condition, 12-6
- ABS function, 2-14
- ACOS function, 2-14
- Addressed commands (GPIB), E-11
- Addresses, GPIB, E-4
- ADVANCE, 12-19
- ALTER Clause, 8-16
- AND Operator, 2-7
- APPEND, 13-2
- ASC function, 2-16
- ASCII (GPIB) Code Chart, B-1
- ASIN function, 2-14
- ASK functions, 5-3
 - "ANGLE", 5-3
 - "AUTOLOAD", 5-3
 - "BUFFER", 5-4
 - "CHPOS", 5-5
 - "IODONE", 5-6
 - "KEY", 5-7
 - "MEMORY", 5-7
 - "PROCEED", 5-8
 - "SEGMENT", 5-8
 - "SPACE", 5-9
 - "TIME", 5-9
 - "UPCASE", 5-9
- ASK\$ functions, 5-10
 - "CONSOLE", 5-10
 - "DRIVER", 5-11
 - "ERROR", 5-11
 - "ID", 5-12
 - "LU", 5-12
 - ,with GPIB devices, 9-5
 - "PATH", 5-13
 - "ROMPACK", 5-14
 - "SELECT", 5-14
 - "SELFTEST", 5-14
 - "SYSDEV", 5-15
 - "TIME", 5-15
 - "VAR", 5-16
 - "VOLUME", 5-16
- ATAN function, 2-14
- ATN (WBYTE function), 9-12
- ATN(EOI), 9-22
- AUTOLD File, 1-3
- BAU (parameter), D-6
- Binary Operators (BAND, BOR, BXOR, BNOT), 2-8
- BIT (parameter), D-6
- BRANCH, 12-20
- BREAK, 4-2
- BUFFER clause
 - with GETMEM, 8-10
 - with INPUT, 8-16
 - with PRINT, 8-33
 - with PUTMEM, 8-43
- CALL, 7-2
- CHR\$ function, 2-16
- CIC (parameter), D-10
- # Clause
 - with GETMEM, 8-10
 - with INPUT, 8-15
 - with PRINT, 8-33
 - with PUTMEM, 8-43
 - with RBYTE, 8-44
 - with WBYTE, 8-50
- CLI (parameter), D-12
- CLOSE, 8-7
- Codes & Formats, GPIB, App. F
- COMPRESS, 6-2
- CON (parameter), D-7
- CONNECT, 4-4
- Conditions, Interrupt, 12-1
 - ABORT, 12-6
 - ERROR, 12-8
 - GPIB, 12-12
 - IODONE, 12-14
 - SRQ INTERRUPTS (OPT2 DRIVER), 12-15
 - USER-DEFINABLE FUNCTION KEYS, 12-17
- Control Character Functions, 10-4
- Controller-In-Charge (GPIB), E-8
- COPY, 8-8
- COS function, 2-14
- CR (parameter), D-7
- CTS (parameter), D-7
- DATA, 8-9
- DCD (parameter), D-7
- DCL (WBYTE function), 9-13
- DEBUG, 4-5
- DEL (parameter), 9-3, D-8
- DELETE ALL, 6-2
- DELETE FILE, 14-2
- DELETE VAR, 6-3
- DELN Clause
 - with GETMEM, 8-10
 - with INPUT, 8-17
- DELS Clause
 - with GETMEM, 8-10
 - with INPUT, 8-17
- DIM, 6-4

INDEX

- DIR (statement), 14-3
- DIR (parameter), D-11
- DISABLE, 12-21
- DISMOUNT, 14-4
- DIV Operator, 2-7
- DSR (parameter), D-7
- DTR (parameter), D-8
- ECH (parameter), D-7
- EDI (parameter), D-6
- ENA (parameter), D-10
- ENABLE, 12-22
- END, 7-4
- Environments
 - inherited, 11-5
 - local, 11-5
- EOA (parameter), 8-30, D-7, D-8, D-9, D-11, D-12
- EOF, 14-5
- EOH (parameter), 8-30, D-7, D-8, D-9, D-11, D-12
- EOI (WBYTE function), 9-13
- EOI (interrupt), 12-12
- EOM (parameter), 8-30, 8-31, D-7, D-9, D-12
- EOQ (parameter), 8-18, 9-5, D-9
- EOU (parameter), 8-30, D-7, D-8, D-9, D-11, D-12
- #ER (parameter), D-8
- ERR (parameter), D-7
- ERROR Interrupt Condition, 12-8
- Error Messages, App. A EXIT, 7-5
- EXP function, 2-14
- Expressions, 2-12
- FLA (parameter), D-6
- Flagging (parameter), 10-2
- Flow (TRACE Flag), 4-18
- FOR (parameter), D-6, D-12
- FOR, 7-7
- FORMAT, 14-6
- Front Panel, 3-1
- Function, 11-8
- Functions, Numeric, 2-14
- Functions, String, 2-15
- GET (WBYTE function), 9-13
- GETMEM, 8-10
- Glossary, App. C GOSUB, 7-9
- GOTO, 7-9
- GPIB Interrupt Conditions
 - (DCL, EOI, IFC, MLA, MTA, SRQ, TCT), 12-12
- GTL (WBYTE function), 9-14
- IBA (parameter), D-6
- IF..THEN..ELSE, 7-11
- IFC (WBYTE function), 9-14
- IFC (interrupt), 12-12
- IMAGE, 8-11
- IND (parameter), D-11
- INIT, 5-17
- INPUT, 8-12
 - with GPIB driver, 9-6
 - with TAPE driver, 14-7
- INPUT USING
 - modifiers (table), 8-19
 - operators (table), 8-19
- Instrument Options, 3-12
- INT function, 2-14
- INTEGER, 6-5
- Interrupt Conditions, 12-1
- IST (parameter), 9-3, D-8
- ITEM Format, RS-232-C, 10-7
- Keywords, 2-21
- Labels, 2-20
- LEN function, 2-16
- LET, 6-6
- LF (parameter), D-7
- LGT function, 2-14
- Line Numbers, 2-20
- LIST, 4-7
- Listeners (GPIB), E-9
- LLO (WBYTE function), 9-14
- LOAD, 13-4
- Local environments, 11-5
- LOG function, 2-14
- Logical Operators (AND, OR, XOR, NOT), 2-7
- Logical Units, 8-4
- LON (parameter), D-11
- LONG, 6-7
- MA (parameter), 9-3, D-8
- MAX Operator, 2-7
- MIN Operator, 2-7
- MLA (WBYTE function), 9-14
- MLA (interrupt), 12-12
- MOD Operator, 2-7
- MONITOR, 12-24
- MTA (WBYTE function), 9-15
- MTA (interrupt), 12-12
- NEXT, 7-7
- NOBREAK, 4-8
- NOT Operator, 2-7
- NOTRACE, 4-9
- Number Representation, 2-3
- Numeric Variables, 2-5
- OFF, 12-25
- ON, 12-26
- OPE (parameter), D-12
- OPEN, 8-28
 - with GPIB driver, 9-4
 - with TAPE driver, 14-8
- Operators, 2-7

- OPT2 Driver, 12-15, D-11
- Opt2in (romcall), 7-13
- Opt2out (romcall), 7-13
- Options, Instrument, 3-12
- Option 2 (TTL Interface Port), 3-13, 7-13
- OR Operator, 2-7
- Parallel Polling, 9-22
- Parameters
 - reference (subprograms), 11-2
 - stream specifications, D-2
 - value (subprograms), 11-2
- Parity, 10-2
- PAR (parameter), 10-2, D-6
- P/D Keyboard, 3-5
- PEN (parameter), D-10
- PHY (parameter), D-11
- Physical Mode (TAPE), 8-46, 8-52, 14-12, 14-15
- PI function, 2-14
- PNS (parameter), D-8
- POLL, 9-19
- POS function, 2-16
- POSN function, 2-16
- PPC (WBYTE function), 9-15
- PPU (WBYTE function), 9-16
- PRI (parameter), 9-5, D-9
- PRINT, 8-30
 - with GPIB driver, 9-8
 - with TAPE driver, 14-9
- PRINT USING
 - modifiers (table), 8-35
 - operators (table), 8-35
- Proceed Mode, 8-14, 8-31
 - Errors, 12-10
- Program Development Keyboard, 3-5
- PROGram (TRACE Flag), 4-18
- PROMPT Clause, 8-18
- PUTMEM, 8-43
- RAT (parameter), D-8
- RBYTE, 8-44
 - with GPIB driver, 9-10
 - with TAPE driver, 14-12
- RCALL, 7-13
 - use with Option 2 (TTL Interface Port), 7-13
- READ, 8-47
- Reference Parameters, 11-2
- REM, 4-11
- REN (WBYTE function), 9-16
- RENAME, 14-13
- RENUMBER, 4-12
- REP\$ Assignment Statement, 2-17
- Reserved Keywords, 2-21
- RESTORE, 8-48
- RESUME, 12-28
- RETRY, 12-29
- RETURN, 7-15
- RND function, 2-14
- ROUND function, 2-14
- RTS (parameter), D-8
- RUN, 7-16
- SAVE, 13-5
- SC (parameter), 9-2, D-8
- SDC (WBYTE function), 9-16
- SEC (parameter), 9-5, D-9
- SEG\$ function, 2-18
- Segment Structure, 4041
- Program, 1-3
- SELECT, 8-49
- Serial Polling, 9-19
- SET, 5-18
 - ANGLE, 5-19
 - AUTOLOAD, 5-19
 - CONSOLE, 5-20
 - DEBUG, 5-20
 - DRIVER, 5-21
 - FUZZ, 5-22
 - PROCEED, 5-24
 - SYNTAX, 5-24
 - SYSDEV, 5-25
 - TIME, 5-26
 - UPCASE, 5-27
- SGN function, 2-14
- SIN function, 2-14
- SIZ (parameter), D-12
- SLIST, 4-17
- SPD (WBYTE function), 9-16
- SPE (WBYTE function), 9-17
- SPE (parameter), 9-5, D-9
- SQR function, 2-14
- SRQ (WBYTE function), 9-17
- SRQ (interrupt), 12-12
 - with OPT2 Driver, 12-15
- STO (parameter), 10-2, D-6
- STOP, 7-17
- Stop bits, 10-2
- Statement Numbers, 2-20
- Statements, 2-20
- Stream Specifications, App. D
- String Constants, 2-4
- String Variables, 2-6
- STR\$ function, 2-18
- SUB
 - statement, 11-10
 - TRACE flag, 4-18
- SUM function, 2-14
- System Console, 1-3, 5-20
- System Controller (GPIB), E-8

INDEX

- Talker/Listeners (GPIB), E-9
- Talkers (GPIB), E-9
- TAN function, 2-14
- TCT (WBYTE function), 9-18
- TCT (interrupt), 12-12
- TIM (parameter), D-7, D-8, D-9
- Timeout (serial poll),
 - see "*SPE*" TL (parameter), D-10
- TMS (parameter), D-7
- TRA (parameter), 9-5, D-9
- TRACE, 4-18
- TRIM\$ function, 2-18
- TTL Interface Port (Option 2), 3-12, 7-13
- #TY (parameter), D-8
- TYP (parameter), D-6
- TYPE, 14-14
- Typeahead Buffer, 10-2
- Universal commands (GPIB), E-10
- UNL (WBYTE function), 9-18
- UNT (WBYTE function), 9-18
- VAL function, 2-19
- VALC function, 2-19
- Value Parameters, 11-2
- VAR
 - reference parameter indicator, 11-2
 - TRACE Flag, 4-18
- Variables
 - Numeric, 2-5
 - String, 2-6
- VER (parameter), D-11
- VIE (parameter), D-8
- VIEW (TRACE flag), 4-19
- WAIT, 12-31
- WBYTE, 8-50
 - with GPIB driver, 9-12
 - with TAPE driver, 14-15
- XOR Operator, 2-7