ALLEN-BRADLEY

User's Manual

1771-DB Basic Module

Price $25.00

TABLE OF CONTENTS

1	Using This Manual	
1.1	Overview	1-1
1.2	Manual's Purpose	1-1
1.3	Audience	1-1
1.4	Terminology	1-1
1.5	Conventions	1-2
1.6	Manual Design	1-2
1.7	Important Information	1-2
2	Introducing the 1771-DB Basic Module	
2.1	Objectives	2-1
2.2	General Features	2-1
2.3	Hardware Features	2-2
2.4	Software Features	2-3
2.5	Specifications	2-4
3	Installing Your 1771-DB Module	
3.1	Objectives	3-1
3.2	Installation of the 1771-DB Module	3-1
3.2.1	Power Requirements	3-2
3.2.2	Module Location in the I/O Chassis	3-2
3.2.3	Module Keying	3-2
3.2.4	Configuration Plugs	3-2
3.2.5	Module Installation	3-3
3.2.6	Initial Start-up Procedure	3-3
3.3	Using the 1771-DB Module's Programming and Peripheral Communication Ports	3-3
3.3.1	Connecting a 1770-T3/T4 Terminal to the Programming Port	3-3
3.3.2	Connecting a 1770-SB Recorder to the Peripheral Port	3-4
3.3.3	Connecting a 1770-HD Printer	3-4
3.3.4	Connecting Other RS-232-C/RS-423-A Devices	3-4
3.3.5	Connecting RS-422 Devices	3-6
3.3.6	Cable Assembly Parts	3-6
3.4	Module Status Indicator	3-6
3.5	Installing the User PROM	3-7
3.6	Battery Usage	3-7
4	Operating Functions	
4.1	Objectives	4-1
4.1.1	Definition of Terms	4-1
4.2	Description of Commands	4-4
4.2.1	RUN	4-4
4.2.2	CONT	4-5
4.2.3	LIST	4-6
4.2.4	LIST#	4-7
4.2.5	NEW	4-7
4.2.6	NULL	4-7
4.2.7	CTRL C	4-7
4.2.8	CTRL S	4-7
4.2.9	CTRL Q	4-8

4.2.10 Overview of EPROM File Commands 4-8
4.2.11 RAM and ROM 4-8
4.2.12 XFER 4-9
4.2.13 PROG 4-9
4.2.14 PROG1 and PROG2 4-11
4.3 Description of Statements 4-11
4.3.1 CALL 4-11
4.3.2 CLEAR 4-12
4.3.3 CLEARI 4-13
4.3.4 CLOCK1 and CLOCK0 4-13
4.3.5 DATA-READ-RESTORE 4-14
4.3.6 DIM 4-15
4.3.7 DO-UNTIL 4-16
4.3.8 DO-WHILE 4-16
4.3.9 END 4-17
4.3.10 FOR-TO-STEP-NEXT 4-19
4.3.11 GOSUB-RETURN 4-20
4.3.12 GOTO 4-21
4.3.13 ON GOTO-ON GOSUB 4-22
4.3.14 IF-THEN-ELSE 4-22
4.3.15 INPUT 4-23
4.3.16 LET 4-25
4.3.17 ONERR 4-25
4.3.18 ONTIME 4-26
4.3.19 PRINT 4-28
4.3.20 PRINT# 4-30
4.3.21 PHO.,PHI.,PHO.#,PHI.# 4-31
4.3.22 PUSH 4-31
4.3.23 POP 4-32
4.3.24 REM 4-33
4.3.25 RETI 4-34
4.3.26 STOP 4-34
4.3.27 STRING 4-34
4.4 Description of Arithmetic/Logical Operators and Expressions 4-35
4.4.1 Dual Operand (DYADIC) Operators 4-35
4.4.2 Unary Operators 4-37
4.4.2.1 General Purpose 4-37
4.4.2.2 Log Functions 4-38
4.4.2.3 Trig Functions 4-38
4.4.3 Understanding Precedence of Operators 4-40
4.4.4 How Relational Expressions Work 4-41
4.5 Special Operators 4-41
4.5.1 Special Function Operators 4-41
4.5.2 System Control Values 4-43
4.6 Data Transfer CALLS 4-44
4.6.1 Update Block Transfer Read Buffer (timed) (CALL 2) 4-44
4.6.2 Update Block Transfer Write Buffer (timed) (CALL 3) 4-45
4.6.3 Set Block Transfer Write Length (CALL 4) 4-45
4.6.4 Set Block Transfer Read Length (CALL 5) 4-45
4.6.5 Update Block Transfer Write Buffer (CALL 6) 4-45
4.6.6 Update Block Transfer Read Buffer (CALL 7) 4-45
4.6.7 Disable Interrupts (CALL 8) 4-45

4.6.8 Enable Interrupts (CALL 9) 4-46
4.6.9 3-Digit Signed, Fixed Decimal BCD to Internal FP +xXXX (CALL 10) 4-46
4.6.10 16-Bit Binary to Internal FP (CALL 11) 4-46
4.6.11 4-Digit Signed Octal to Internal FP +xXXXX (CALL 12) 4-46
4.6.12 6-Digit, Signed, Fixed Decimal BCD to Internal FP +xXXXXXX (CALL 13) 4-46
4.6.13 4-Digit BCD to Internal FP (CALL 14) 4-47
4.6.14 Internal FP to 3-Digit, Signed, Fixed Decimal BCD +xXXX (CALL 20) 4-47
4.6.15 Internal FP to 16-Bit Binary (CALL 21) 4-47
4.6.16 Internal FP to 4-Digit, Signed Octal (CALL 22) 4-48
4.6.17 Internal FP to 6-Digit, Signed, Fixed Decimal BCD +xXXXXXX (CALL 23) 4-48
4.6.18 Internal FP to 3.3 Digit, Signed, Fixed Decimal BCD +xXXX.XXX (CALL 26) 4-48
4.6.19 Internal FP to 4 digit BCD (CALL 27) 4-48
4.7 Communication Port Support CALLS 4-48
4.7.1 Peripheral Port Support Parameter Set (CALL 30) 4-49
4.7.2 Peripheral Port Support - Display Parameters (CALL 31) 4-50
4.7.3 Save Program to Data Recorder (CALL 32) 4-50
4.7.4 Verify Program to Data Recorder (CALL 33) 4-50
4.7.5 Load Program from Data Recorder (CALL 34) 4-51
4.7.6 Get Numeric Input Character from Peripheral Port (CALL 35) 4-51
4.7.7 Save Labeled Program to Data Recorder (CALL 38) 4-51
4.7.8 Load Labeled Program to Data Recorder (CALL (39) 4-51
4.8 Wall Clock Support CALLS 4-52
4.8.1 Setting the Wall Clock (hour, minute, second) (CALL 40) 4-52
4.8.2 Setting the Wall Clock Date (day, month, year) (CALL 41) 4-52
4.8.3 Date Retrieve Numeric (day, month, year) (CALL 44) 4-53
4.8.4 Time Retrieve Numeric (CALL 46) 4-53
4.9 Description of String Operators 4-54
4.9.1 What Are Strings? 4-54
4.9.2 The ASC Operator 4-54
4.9.3 The CHR Operator 4-56
4.9.4 String Support CALLS 4-57
4.9.4.1 String Repeat (CALL 60) 4-59
4.9.4.2 String Append (CONCATENATION) (CALL 61) 4-60
4.9.4.3 Find a String in a String (CALL 64) 4-61
4.9.4.4 Replace a String in a String (CALL 65) 4-61
4.9.4.5 Insert a String in a String (CALL 66) 4-62
4.9.4.6 Delete a String from a String (CALL 67) 4-62
4.9.4.7 Determine the Length of a String (CALL 68) 4-63
4.10 Memory Support Calls 4-63
4.10.1 ROM to RAM Program Transfer (CALL 70) 4-63
4.10.2 ROM/RAM to ROM Program Transfer (CALL 71) 4-64
4.10.3 ROM/RAM Return (CALL 72) 4-64
4.10.4 Battery-Backed RAM Disable (CALL 73) 4-64
4.10.5 Battery-Backed RAM Enable (CALL 74) 4-64

5	1771-DB Block Transfer Programming	
5.1	Objectives	5-1
5.2	Concepts Behind BASIC Module Block Transfer With a PLC	5-1
5.3	Block Transfer Programming	5-2
5.3.1	PLC-2 Family Processors	5-2
5.3.1.1	Sample Programming-Single Data Set	5-2
5.3.1.2	Sample Programming-Multiple Data Set	5-4
6	Data Types	
6.1	Objectives	6-1
6.2	Data Types-Outputs	6-1
6.2.1	16 Bit Binary	6-1
6.2.2	3 Digit, Signed, Fixed Decimal BCD	6-1
6.2.3	4 Digit, Unsigned, Fixed Decimal BCD	6-1
6.2.4	4 Digit, Signed, Octal	6-1
6.2.5	6 Digit, Signed, Fixed Decimal BCD	6-2
6.2.6	3.3 Digit, Signed, Fixed Decimal BCD	6-2
6.3	Data Types-Inputs	6-2
7	Editing a Procedure	
7.1	Objectives	7-1
7.2	Entering the Edit Mode	7-1
7.3	Editing Commands	7-1
7.3.1	Move	7-1
7.3.2	Replace	7-1
7.3.3	Insert	7-1
7.3.4	Delete	7-1
7.3.5	Retype	7-1
7.3.6	Exits	7-1
7.3.7	Renumber	7-2
7.4	Editing a Simple Procedure	7-2
8	Error Messages and Anomalies	
8.1	Objectives	8-1
8.2	Error Messages	8-1
8.3	Disabling Control-C	8-4
8.4	Anomalies	8-5
Appendix A	Quick Reference Guide	A-1
Appendix B	ASCII Conversion Table	B-1

Chapter 1 - Using This Manual

1.1 Overview

Read this chapter before you use the 1771-DB Module. It will tell you how to use this manual properly and efficiently for the tasks you will have to perform.

1.2 Manual's Purpose

This manual will show you how to install and operate your BASIC module.

This will be accomplished in four areas:

- o Hardware Specifications
- o Installation of the 1771-DB Module
- o Basic instruction set
- o Programming the 1771-DB Module

This manual is not a BASIC tutorial document. It is assumed that the user is familiar with BASIC programming.

1.3 Audience

Before you read this manual or attempt to use the 1771-DB module, you should be familiar with the basic operation of the 1771 I/O structure as it relates to your particular controller. If you are not familiar with this information, refer to the following publications:

Bulletin 1772, Mini-PLC-2 Programmable Controller, Programming and Operations Manual, publication 1772-6.8.4

Bulletin 1772, Mini-PLC-2/15 Programmable Controller (Series B), Programming and Operations Manual, publication 1772-6.8.2

Bulletin 1772, PLC-2/20 Programmable Controller, Programming and Operations Manual, publication 1772-6.8.1

Bulletin 1772, PLC-2/30 Programmable Controller, Programming and Operations Manual, publication 1772-6.8.3

Bulletin 1775, PLC-3 Programmable Controller Programming Manual, publication 1775-6.4.1.

1.4 Terminology

We refer to certain types of equipment throughout this manual. To make the manual easier for you to read and understand, we avoid repeating full product names where possible.

We refer to the:

- o BASIC module (cat. no. 1771-DB) as the 1771-DB module
- o Industrial Terminal System (cat. no. 1770-T3/T4) as the T3/T4 Terminal
- o Data Recorder (cat. no. 1770-SA/SB) as the 1770-SA/SB recorder
- o Those RS-232-C compatible devices which communicate to the 1771-DB, such as the T3/T4 terminal, 1770-SB recorder, computers, robots, barcode readers, or data terminals, as RS-232-C/RS-423-A device.
- o Devices connected to the peripheral port as LIST devices

1.5 Conventions

In this manual, we use certain notational conventions to indicate keystrokes and items displayed on a CRT or printer. A keystroke is shown indicated with brackets such as :

[ENTER]

We describe any exceptions to these conventions where they occur.

1.6 Manual Design

This manual is designed with as many as three divisions per page. These divisions include:

- o Headings describe the contents of the text.
- o Text provides explanations, information, and examples.
- o Figures showing displays, hardware, and diagrams.

1.7 Important Information

In this manual, there are three different types of important information:

- o WARNINGS inform you where personal injury may occur if you do not follow the written procedure.
- o CAUTIONS inform you where damage to your equipment may occur if you do not follow the written procedure.
- o NOTES inform you where exceptions may take place to remind you about information.

Chapter 2-Introducing the 1771-DB Basic Module

2.1-Objectives

This chapter discusses the functions and features of the 1771-DB module. When you finish reading this chapter, you should:

- o Understand and be able to identify the hardware components of the 1771-DB module
- o Understand the basic features and functions of the 1771-DB module

2.2 General Features

The 1771-DB (figure 2.1) will provide math functions, report generation and interactive BASIC language capabilities for any AB PLC that can communicate to the 1771 I/O system via block transfer.

- o Basic programming using the Intel Basic-52 language
- o Math functions consistent with the Basic-52 definition
- o Two serial Ports capable of interfacing to various user devices
- o Each communications port can be configured independently
- o User accessible real-time clock with 5ms resolution
- o User accessible "wall" clock/calendar with 1 second resolution
- o Program generation and editing via a dumb ASCII terminal or 1770-T3/T4 Industrial Terminal in alphanumeric mode
- o Program storage and retrieval using the 1770-SA/SB recorder
- o Block transfer communication capability from a PLC-2 family or PLC-3 programmable controller
- o Data table reads/writes from/to the programmable controllers via block transfer
- o On board program PROM burning

2.3 Hardware Features

The hardware is a one-slot module design having the following functions and features:

- One RS-423A/232C compatible program port which supports ASCII terminals to provide operator program interaction, command level input, printer output, etc. Supports XON/XOFF. On power-up the module does an auto-baud search and sets the program port baud rate equal to the baud rate of the terminal. Typing the space bar initializes the baud rate. Refer to section 3.2.6. Changing the baud rate when switching terminals is accomplished by changing RCAP2.

```
>10 RCAP2 = value
```

The allowable values for RCAP2 are as follows:

Value		Baud
65518	(OFFEEH)	19200
65500	(OFFDCH)	9600
65464	(OFFB8H)	4800
65392	(OFF70H)	2400
65248	(OFEEOH)	1200
64960	(OFDCOH)	600
64384	(OFB80H)	300

- One RS-423A/232C/RS-422 compatible peripheral serial communications port, supporting XON/XCFF software handshaking and RTS/CTS, DTR,DSR,DCD hardware handshaking for interfacing to printers and commercial asynchronous modems. The peripheral port is configured via a CALL routine. Refer to section 4.7.1.

- Interface to the 1771-I/O rack backplane to support block transfer

- Real-time clock/calendar with battery back-up, available for program access

- 13K bytes of battery backed-up RAM for user programs

- Battery replacement can be accomplished without removing the module from the rack

- 32K bytes of EPROM storage for user software routines

- All power is derived from the back plane (1.5 amps).

- Multiple 1771-DBs can reside in the same I/O rack and function independently of each other

2.4 Software Functionality

This module runs user-written Basic language programs in an interactive mode through the dumb terminal/programming port interface, or on power up. The execution of these programs will allow, but does not require, interfacing directly to any programmable controller ladder programs.

The following devices and features will be supported:

- o terminal for programming, editing, system commands, displaying data, and interactive program dialog

- o serial ports for report generation output, upload/download to 1770-SA/SB recorder

- o PLC-2 and PLC-3 data table reads and writes via block transfer

Routines are provided to employ both the real time clock and the "wall" clock/calendar. The "wall" clock time base will be seconds.

Program execution may be started in three different ways:

- o by individually entered commands at the interactive terminal

- o upon recognition of predefined PLC data table values

- o at power-up initialization

User-written programs can be stored and executed in EPROM. The system provides the capability of having one user program resident in RAM and up to 255 (depending on the program sizes) independent user programs residing simultaneously in EPROM memory. Any of these programs are executable by all three of the modes defined above.

The programs will run single-task mode only

Data types to be supported are:

- o PLC-2- 12-Bit BCD, 16 bit binary, 4 digit BCD, 6 digit BCD, 4 digit octal

- o PLC-3- 3 digit signed BCD (counters, integers); 16-bit unsigned binary (outputs, inputs, timers, binary); 4 digit BCD

2.5 Specifications

Isolation

- The Programming Port is isolated from the 1771-I/O backplane. ($\pm$500V)
- The Peripheral Port is isolated from the 1771-I/O backplane. ($\pm$500V)
- The Programming Port is isolated from the Peripheral Port. ($\pm$500V)

Communication Rates

- 300, 600, 1200, 2400, 4800, 9600, 19.2K bits
- Communication rates vs. distance

Communication Rate-bps	Maximum Distance Allowed-Ft. RS-232-C	RS-423	RS-422
300	50	4000	4000
600	50	3000	4000
1200	50	2500	4000
4800	50	800	4000
9600	50	400	4000
19,200	50	200	4000

Wall Clock Accuracy

- Absolute: $\leq \pm$ 5 min/month @ 25°C (adjustable to zero)
- Drift $\leq \pm$ 50 ppm/$^{\circ}$C
- Drift $\leq \pm$ 50 ppm/year @ 25°C

Module location

- Any I/O module slot of a 1771 I/O chassis

Backplane Power Supply Load

- 1.5A

Ambient Temperature Rating

- Operational 0° to 60° (32° to 140°F)
- Storage -40° to 85° (-40° to 185°F)

Relative Humidity Rating

o 5% to 95% (without condensation)

Keying (Top Backplane Connector). The rear panel keying is between pins 8 and 10 and between pins 32 and 34.

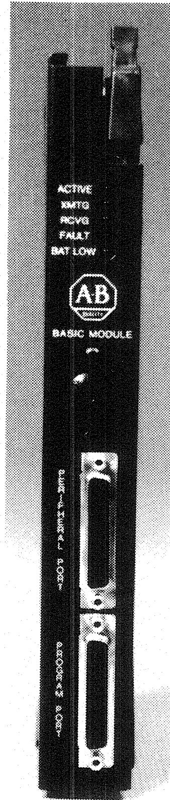


Figure 2-1
BASIC Module Front Edge

Chapter 3-Installing Your 1771-DB Module

3.1 Objectives

This chapter discusses installing your 1771-DB module into a 1771 I/O rack. It also provides information on connecting other serial devices to the communication ports.
After reading this chapter you should be able to:

- Configure the module using the configuration plugs
- Insert the 1771-DB into a 1771 I/O backplane
- Connect a T3/T4 terminal to the 1771-DB
- Connect a 1770-SA/SB recorder
- Connect a 1770-HD printer
- Connect other RS-423-A/232C compatible devices
- Connect RS-422 devices
- Understand module status indicators
- Install additional EPROM's

3.2 Installation of the 1771-DB Module

WARNING: Disconnect and lockout all AC power from the controller and system power supplies before installing modules to avoid injury to personnel and damage to equipment.

Read this installation section completely before beginning work. Double check all option selections and connections before you begin programming.

Before installing your BASIC module in the I/O chassis you must:

1. Calculate the power requirements of all the modules in each chassis (section 3.2.1).

2. Determine the location of the module in the I/O chassis (section 3.2.2).

3. Key the backplane connectors in the I/O chassis (section 3.2.3).

4. Set the module configuration plugs. See the section titled "Configuration Plugs" (section 3.2.4).

3.2.1 Power Requirements

Your module receives its power through the 1771 I/O chassis backplane from the chassis power supply. It does not require any other external power supply to function. When planning your system you must consider the power usage of all modules in the I/O chassis to prevent overloading the chassis backplane and/or power supply. Each 1771-DB requires 1.5 amps at +5V DC. Add this to the requirements of all other modules in the I/O chassis.

Caution: Do not insert or remove modules from the I/O chassis while system power is on. Failure to observe this rule may result in damage to module circuitry.

3.2.2 Module Location in the I/O Chassis

Place your module in any I/O slot of the I/O chassis except for the extreme left slot. This slot is reserved for PC processors or adapter modules. Do not put the 1771-DB in the same module group as a discrete high density module.

3.2.3 Module Keying

Initially you can insert the 1771-DB into any I/O module slot in the I/O chassis. However, once a slot has been designated for a module you must not insert other types of modules into those slots. We strongly recommend that you use the plastic keying bands shipped with each I/O chassis, to key I/O slots to accept only one type of module. Your BASIC module is slotted in two places on the rear edge of the board. The position of the keying bands on the backplane connector must correspond to these slots to allow insertion of the module. Any I/O rack connector may be keyed to receive the module assembly. Snap the keying bands onto the upper backplane connectors between these numbers printed on the backplane (figure 3.1).

o between 8 and 10
o between 32 and 34

You may change the position of these bands if subsequent system design and rewiring makes insertion of a different type of module necessary. Use needle-nose pliers to insert or remove keying bands.

3.2.4 Configuration Plugs

Referring to Figure 3-2, there are four sets of configuration plugs which must be set for proper operation of the 1771-DB BASIC Module. Each of the plugs is shown with a short explanation of how to set them.

3.2.5 Module Installation

Now that you have determined the configuration, power requirements, location, keying and wiring for your 1771-DB, you are ready to install it in the I/O chassis.

Step 1 -Turn off power to the I/O chassis.

Step 2 -Insert your module in the I/O rack. Plastic tracks on the top and bottom of the slots guide the module into position. Do not force the module into its backplane connector. Apply firm, even pressure on the module to seat it properly. Note the rack, module group and slot numbers and enter them in the module address section of the block transfer instructions.

Step 3 - Snap the I/O chassis latch over the module. This will secure the module in place.

3.2.6 Initial Start-up Procedure

The following procedure must be used when powering up the 1771-DB module for the first time. This procedure is a continuation of the installation procedure presented above.

Step 4 -Connect your cable from your program terminal to the 1771-DB program port.

Step 5 -Turn on your program terminal. Select the Alpha Numeric mode (if required). Select the baud rate. Hit the return key.

Step 6 -Turn on power to the rack. The FLT LED and ACTIVE LED will turn on momentarilly. The FLT LED will turn off and the ACTIVE LED will remain on.

Step 7 -Hit the space bar. You should see the sign on prompt and then >READY.

You are now ready to begin BASIC programming.

3.3 Using the 1771-DB Module's Programming and Peripheral Communication Ports

Figure 3.3 PROGRAM and PERIPHERAL PORTS.

3.3.1 Connecting A T3/T4 Terminal to the Programming Port

An Industrial Terminal System can be used as the programming system for the 1771-DB module. The 1771-DB should be connected to CHANNEL B only. Cable can be constructed for distances up to 50 feet. The pinout of CHANNEL B (figure 3.4) identifies the pins used. The cable is constructed as shown in (figure 3.4). Note that pins 7,18,25 on the Industrial Terminal CRT Monitor end of the cable must be wired together.

Additional information on the Industrial Terminal System can be found in catalog No. 1770-805 Users Manual.

3.3.2 Connecting A 1770-SA/SB Recorder to the Peripheral Port

The 1770-SB Data Cartridge Recorder or 1770-SA Digital Cassette Recorder can be used to save and load programs to the 1771-DB module. The recorder must be connected to the peripheral port of the 1771-DB. The cable pinout for this cable is shown in (figure 3.5). Note that the standard cable will not interface properly with the BASIC module. For additional information on the 1770-SB and for information on the 1770-SA refer to Cat. No. 1770-6.5.4 and 1770-6.5.1 respectively. Familiarity with the 1770-SA/SB recorder is assumed.

Additional information on saving and loading programs can be found in Chapter 4 of this manual.

3.3.3 connecting A 1770-HD Printer

A 1770-HD Printer can be connected to the peripheral port of the 1771-DB module for program listing, report generation, etc. Cable connections for connection to the 1771-DB peripheral port are diagrammed in (figure 3.6). Refer to the 1770-HD printer manual (Anadex DP-9725B Product Manual).

We reccommend enabling XON/XOFF on the peripheral port (see section 4.7.1) and selecting XON/XOFF(DEC) protocal on the 1770-HD (switch selectable). Refer to printer manual.

Additional information on printing reports and listing programs can be found in Chapter 4 of this manual.

3.3.4 Connecting Other RS-423-A/232C Compatible Devices

There are many devices that can be connected to the 1771-DB modules peripheral port. The only precaution is to make sure your particular peripheral device is compatible with the peripheral port of the 1771-DB modules signal levels and wiring connections.

Figure 3.7 lists the port wiring connections for the 1771-DB peripheral port. Consult your particular devices connection requirements for compatible connection.

The following pin descriptions are provided:

Chassis/Shield (Pin 1)

This pin is used for shielding purposes. It is connected to the chassis ground.

TXD (Pin 2)

TXD is a RS-423A compatible serial output port.

RXD (Pin 3)

RXD is a RS-423A compatible serial input data port.

RTS (Pin 4)

RTS is a RS-423 compatible hardware handshaking output line. This line should transition to a mark(1) state when the 1771-DB has data in the output queue which it is requesting permission to send to the data communications equipment.

CTS (Pin 5)

CTS is a RS- 423A compatible hardware handshaking input line. This line must be in a mark(1) state in order for the 1771-DB to be able to transmit out the peripheral port. If no corresponding signal exists on the data communications equipment, then CTS should be shorted to RTS. It does not matter whether RS-423A or RS-422 protocol is being used, this pin must be a mark(1) to transmit.

DSR (Pin 6)

DSR is a general purpose RS-423A compatible input pin. The 1771-DB will transmit a receive independent of the state of the line.

Signal common (Pins 7, 9, and 10)

These are the signal common pins and should be used to reference all the RS-423A/RS-422 compatible signals.

DCD (Pin 8)

If DCD is enabled via CALL 30, the 1771-DB will not transmit or receive characters until the DCD line is in the mark(1) state. When disabled, the 1771-DB ignores the state of this line.

422 TDX (Pins 14, 25)

These two lines are the RS-422A compatible equivalent of the RS-423A TXD line. These two lines are differential serial output lines.

422RXD (Pins 16,18)

These are the differential RS-422A compatible serial input lines.

DTR (Pin 20)

DTR is a RS-423A compatible hardware handshaking output line. This line will transition to a space(0) state when the 1771-DB input queue has accumulated more than 223 characters. The DTR line will transition to a mark(1) state when the input queue contains less than 127 characters.

3.3.5 Connecting RS-442 Devices

The 1771-DB module has the capability to communicate to various RS-422 devices. The RS-422 signals to both send and receive data are located on the Peripheral Port of the 1771-DB. The correct signal connections are shown in Figure 3-7. The RS-422 port is tri-stated when not sending characters and is capable of being operated in a multi-dropped configuration. It is EIA standard RS-422-A compatible.

When you are using the peripheral port as a 422 port, you must short pins 4 to 5 and 6 to 20 on the peripheral port.

Attention should be given to understanding your particular peripheral devices connection requirements.

3.3.6 Cable Assembly Parts

You must supply cables for interfacing the devices to the program and peripheral ports. You may construct the cables with the parts listed in figure 3.8.

3.4 Module Status

There are five LEDs (figure 3.9) on the front panel of the 1771-DB which help in diagnosing the status of the module. The green ACTIVE LED should activate automatically upon power-up and should remain on to indicate power is being applied. The green XMTG LED will activate when data is being transmitted from the 1771-DB to a peripheral module via the Peripheral Port. The green RCVG LED will activate when the 1771-DB is receiving data via the Peripheral Port. The red FAULT LED will automatically light upon the application of power to the 1771-DB module. This is normal. After the 1771-DB has passed internal diagnostics, the FAULT indicator should deactivate. If the FAULT indicator persists, this means that the module did not pass its internal diagnostics and should be replaced. The red BAT LOW LED will activate when the battery voltage drops below about 3.0V dc.

3.5 Installing the User Prom

The 1771-DB has a 32K byte EPROM installed (figure 3.10) and is configured for such. It is highly recommended that spare EPROMs be purchased. Any JEDEC standard 8K, 16K, or 32K byte EPROM which uses 12.5Vdc programming voltage will work. Some suggested devices are: INTEL D2764A (8K bytes), INTEL D27128A (16K bytes), AMD AM27256-25DC (32K bytes), AMD AM27256-25DC (32K bytes), HITACHI HN27256-25(32K bytes), INTEL D27256-25(32K bytes), and INTEL D27256(32K bytes).

TO replace the EPROM:

1)turn the small screw (in the socket just above the chip,(figure 3.10) 1/4 turn counterclockwise

2)remove the old chip

3)insert the new chip with pin one down, center notch down as in the diagram on the socket

4)turn the small screw (in the socket just above the chip) 1/4 turn clockwise

5)Refer to section 3.2.4 for the proper setting of the corresponding configuration plug

3.6 Battery Usage

The 13.5 K byte of user RAM and the clock/calendar are battery backed. The drain on the battery should be less than 0.5mA dc during battery back-up (no power) and less than 50 micro A while the 1771-DB is under power. For a Tadiran lithium battery (Tadiran catalog #15-51-03-210-000), the battery life under no power conditions is about 2000 hours. The batteries shelf life is about 20,000 hours. If the 1771-DB loses power just as the BAT LOW indicator was about to activate, then the above mentioned battery should maintain the 1771-DB clock and program data for about three days.

To replace the battery (figure 3.11):

1)place a screwdriver edge in the battery cover slot

2)press inwards slightly

3)rotate the screwdriver and battery cover counterclockwise 1/4 turn

4)release the pressure - the battery cover should come off

5)replace the battery noting the + (positive) side of the battery should be out

6)replace the battery cover

The BAT LOW indicator should go out.

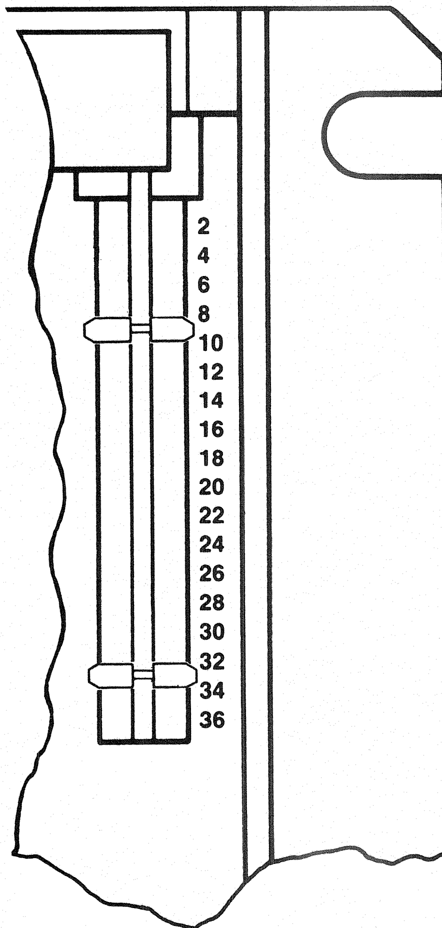


Figure 3-1
Keying Diagram for Placement of Module Keying Bands

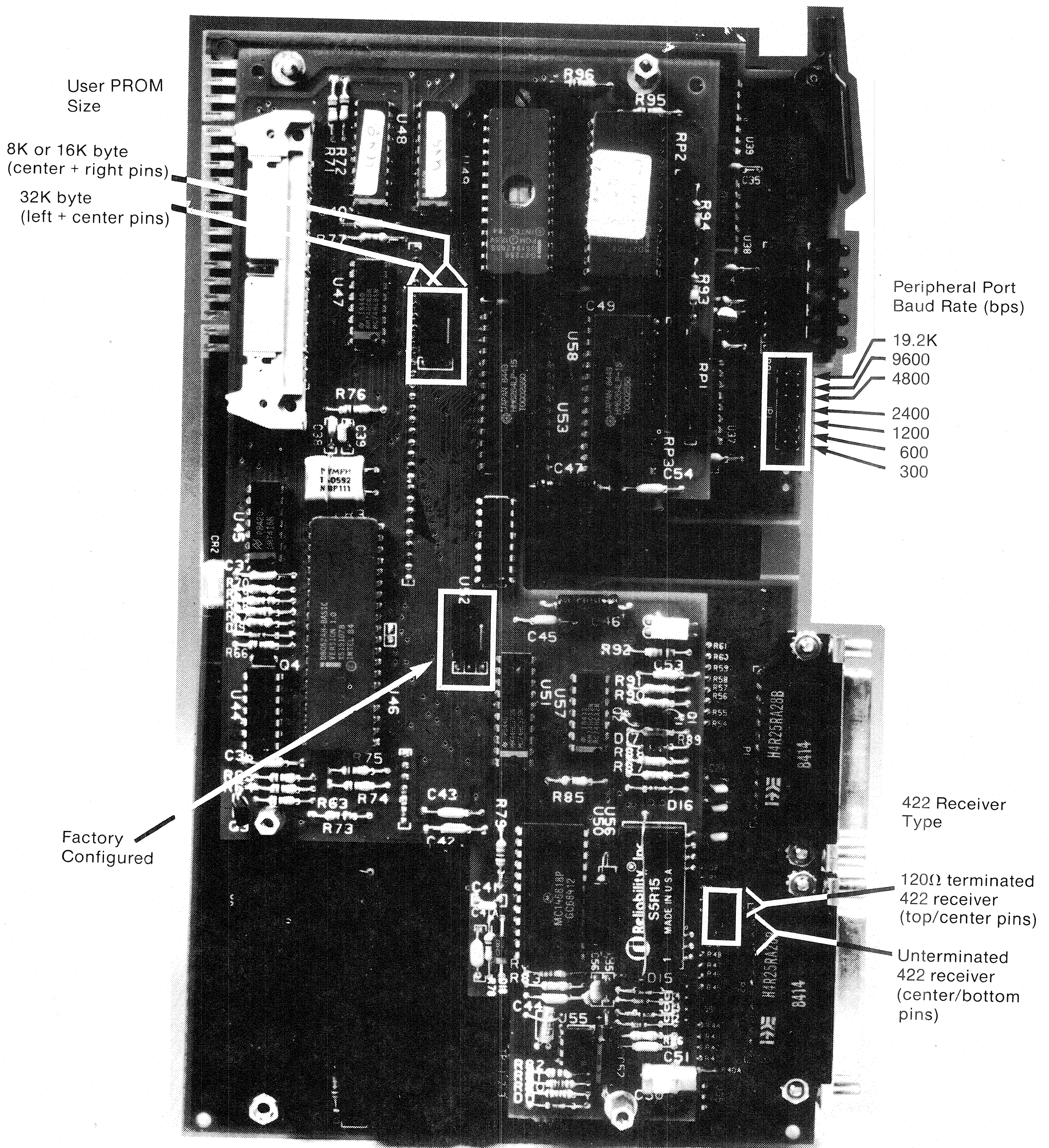


Figure 3-2
Configuration Plugs

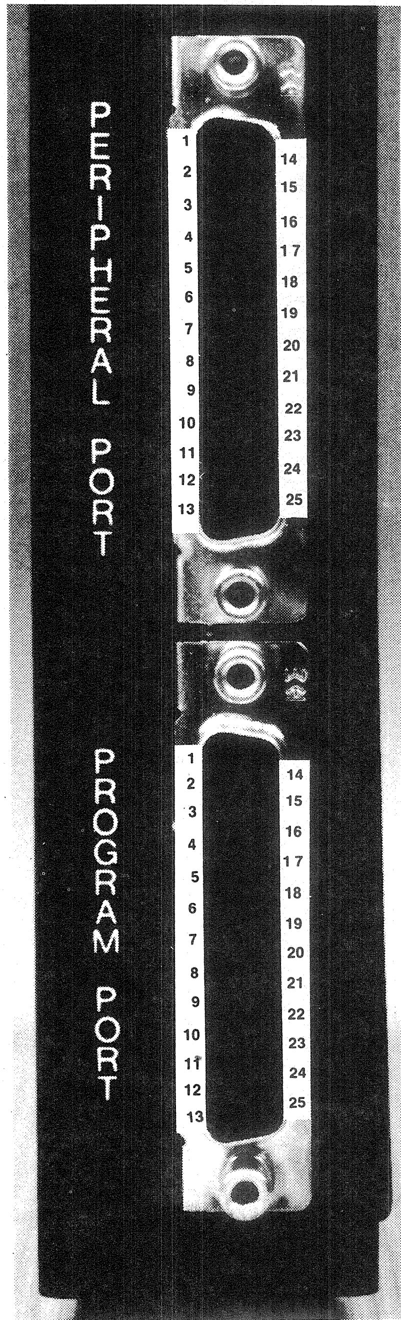
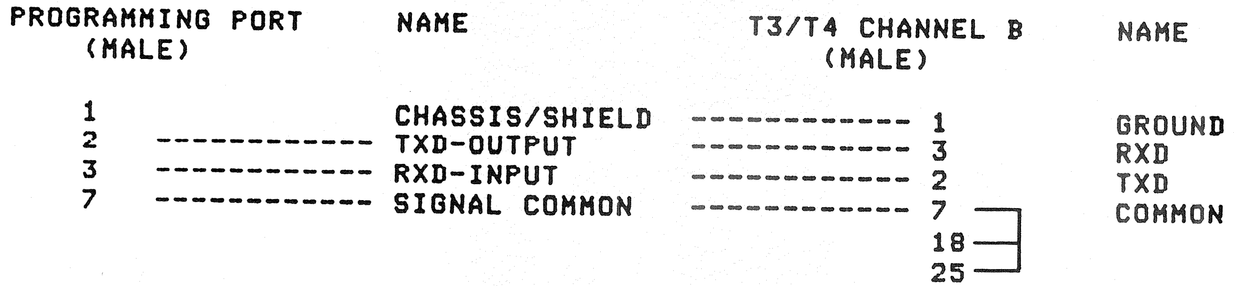


Figure 3-3
Program/Peripheral Port



NOTE: CHASSIS SHIELD SHOULD BE CONNECTED ONLY AT THE TERMINAL END

NOTE: PINS 7, 18, 25 SHOULD BE JUMPER WIRED TOGETHER

Figure 3.4
Cable Connections to T3/T4 Terminal from the Program Port

PERIPHERAL PORT (MALE)		NAME		1770-SA/SB PORT (MALE)	NAME
1	-------------	CHASSIS/SHIELD	-------------	1	SHIELD
2	-------------	TXD-OUTPUT	-------------	2	TD
3	-------------	RXD-INPUT	-------------	3	RD
4	-------------	RTS-OUTPUT	-------------	4	RTS
5	-------------	CTS-INPUT	-------------	5	CTS
6	-------------	DSR-INPUT	-------------	6	DSR
7	-------------	SIGNAL COMMON	-------------	7	GROUND
8	-------------	DCD-INPUT	-------------	8	DCD
20	-------------	DTR-OUTPUT	-------------	20	DTR

Figure 3.5
Cable Connections to 1 7 70-SA/SB from the Peripheral Port

PERIPHERAL PORT		NAME		PRINTER PORT (J2)	NAME
1	-------------	CHASSIS/SHIELD	-------------	1	GROUND
2	-------------	TXD-OUTPUT	-------------	3	RD
3	-------------	RXD-INPUT	-------------	2	TD
7	-------------	SIGNAL COMMON	-------------	7	COMMON

4 ⌐
5 ⌐

6 ⌐
20 ⌐

NOTE: Pins 4 and 5, and pins 6 and 20 should be jumper wired together.

Figure 3.6
Cable Connection to 1 7 70-HD from the Peripheral Port

PERIPHERAL PORT (MALE)	CIRCUIT NAME
1	CHASSIS/SHIELD
2	TXD-OUTPUT
3	RXD-INPUT
4	RTS-OUTPUT
5	CTS-INPUT
6	DSR-INPUT
7	SIGNAL COMMON
8	DCD-INPUT
9	SIGNAL COMMON
10	SIGNAL COMMON
11	NC
12	NC
13	NC
14	RS-422 TXD
15	NC
16	RS-422 RXD
17	NC
18	RS-422-RXD'
19	NC
20	DTR-OUTPUT
21	NC
22	NC
23	NC
24	NC
25	RS-422 TXD'

*NC = No Connection

Figure 3. 7
Peripheral Port Wiring Connections

PART	MANUFACTURERS PART NO.
25 Pin Female Connector	Cannon Type DB-25S, or equiv.
25 Pin Male Connector	Cannon Type DB-25P, or equiv.
Plastic Hood	Amp Type 205718-1
2 Twisted pair 22 gauge, individually shielded cable	Cat. No. 1778-CR Belden 6723 or equiv.

Figure 3.8
Cable Parts

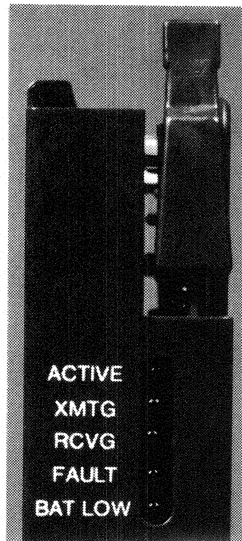


Figure 3-9
Module Status Indicators

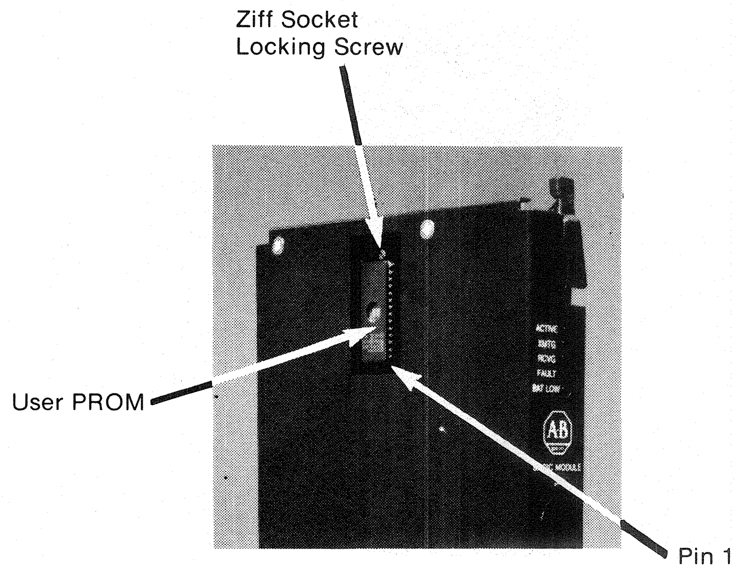


Figure 3-10
User PROM

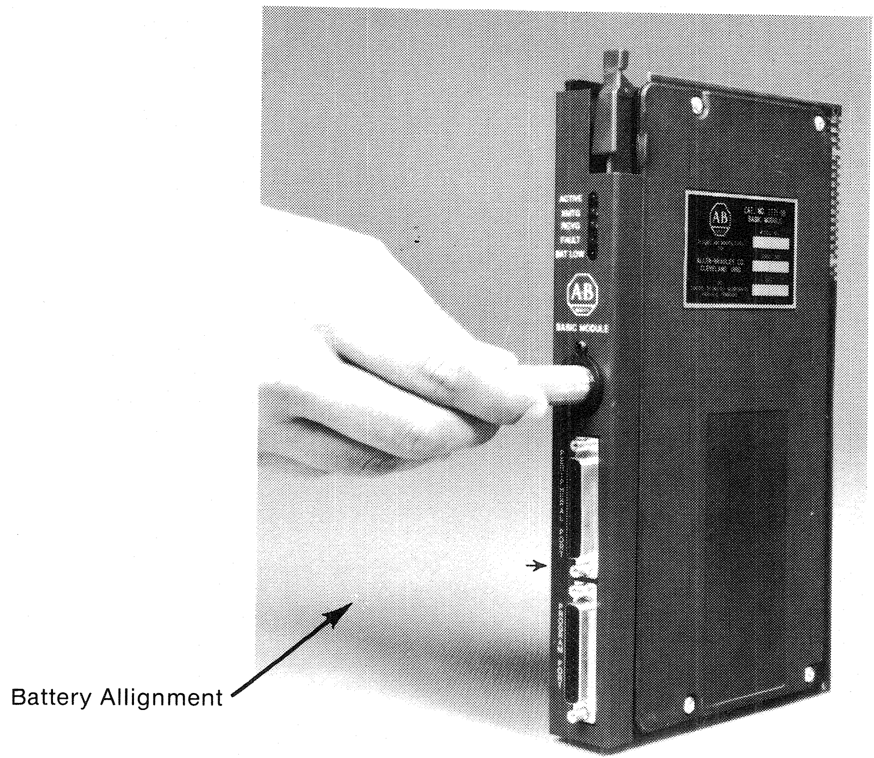


Figure 3-11
Battery Holder

Chapter 4 Operating Functions

4.1 Objectives

After reading this chapter you should be familiar with the 1771-DB's BASIC instruction set and be ready to begin BASIC programming. It will be necessary to refer to this chapter often in order to program the 1771-DB. It will be your reference chapter for future programming problems. This chapter is not intended to be a BASIC programming tutorial. It is assumed that the user is familiar with BASIC programming.

4.1.1 Definition of Terms

Commands:

The BASIC module operates in two modes, the command or direct mode and the interpreter or run mode. You can only enter commands when the processor is in the command or direct mode. This document will use the terms run mode and command mode to refer to the two different operation modes.

Statements:

A BASIC program is comprised of statements. Every statement begins with a line number, followed by a statement body, and terminated with a carriage return (cr), or a colon (:) in the case of multiple statements per line.

There are three types of statements: assignments, input/output, and control.

- Every line in a program must have a statement line number ranging between 0 and 65535 inclusive which is used by BASIC to order the program statements in sequence.
- A statement number can be used only once in any program.
- You need not enter statements in numerical order because BASIC automatically orders them in ascending order.
- A statement may contain no more than 72 characters.
- Blanks (spaces) are ignored by BASIC which automatically inserts blanks during List.
- You may put more than one statement on a line, if separated by a colon (:), but only one statement number is allowed per line.

Format Statements:

Format statements may be used within the print statement. The format statements include TAB([expr]), SPC([expr]), USING(special symbols), and CR (carriage return with no line feed).

Data Format:

The range of numbers that can be represented in the BASIC module is:

±1E-127 to ±.99999999E+127

There are eight significant digits. Numbers are internally rounded to fit this precision. You may enter and display numbers in four formats: integer, decimal, hexadecimal, and exponential.

Example: 129, 34.98, OA6EH, 1.23456E+3

Integers:

In the 1771-DB module, integers are numbers that range from 0 to 65535 or OFFFFH. All integers can be entered in either decimal or hexadecimal format. A hexadecimal number is indicated by placing the character "H" after the number (e.g., 170H). If the hexadecimal number begins with A-F, then it must be preceded by a zero (i.e. A567H should be entered as OA567H). When an operator, such as .AND. requires an integer, the BASIC module truncates the fraction portion of the number so it will fit the integer format. This document will refer to integers and line numbers, respectively, in the following manner:

[integer] -- [ln num]

Constants:

A constant is a real number that ranges from ±1E127 to ±.99999999E + 127. A constant can be an integer. This document will refer to constants in the following manner:

[const]

Operators:

An operator performs a predefined operation on variables and/or constants. Operators require either one or two operands. Typical two operand or dyadic operators include ADD (+), SUBTRACT (-),
MULTIPLY (*), and DIVIDE (/). Operators that require only one operand are often referred to as unary operators. Some typical unary operators are SIN, COS, and ABS.

Variables:

A variable can be either a letter, (i.e. A, X, I), a letter followed by a one dimensioned expression, (i.e. J(4), G(A + 6), I(10*SIN(X)), or a letter followed by a number followed by a one dimensioned expression i.e., A1(8), P7(10*SIN(X)), W8(A + B). Variables that include a one dimensioned expression [expr] are often referred to as dimensioned or arrayed variables. Variables that only involve a letter or a letter and a number are called scaler variables. This document will refer to variables as:

[var].

The BASIC module allocates variables in a "static" manner. That means each time a variable is used, BASIC allocates a portion of RAM memory (8 bytes) specifically for that variable. This memory cannot be de-allocated on a variable to variable basis.That means that if you execute a statement like Q = 3, later on you cannot tell BASIC that the variable Q no longer exists, so "free up" the 8 bytes of memory that belong to Q. The only way you can clear the memory that is allocated to variables is to execute a clear statement. The clear statement "frees" all memory allocated to variables.

Note: Relative to a dimensioned variable, it takes the BASIC interpreter a lot less time to find a scalar variable because there is no expression to evaluate in a scalar variable. So, if you want to make a program run as fast as possible, use dimensioned variables only when you have to. Use scalars for intermediate variables, then assign the final result to a dimensioned variable.

Expressions:

An expression is a logical mathematical expression that involves operators (both unary and dyadic), constants, and variables. Expressions can be simple or complex, i.e. 12*EXP(A)/100, H(1) + 55, or (SIN(A)*SIN(A)+COS(A)*COS(A))/2. A "stand alone" variable [var] or constant [const] is also considered an expression. This document will refer to expressions as:

[expr].

Relational Expressions:

Relational expressions involve the operators EQUAL(=), NOT EQUAL(<>), GREATER THAN(>), LESS THAN(<), GREATER THAN OR EQUAL TO(>=), and LESS THAN OR EQUAL TO (<=). You use them in control statements to test a condition (i.e. IF A < 100 THEN....). Relational expressions always require two operands. We will refer to relational expressions as:

[rel expr].

System Control Values:

The system control values include the following: LEN (which returns the length of your program) and MTOP (which is the last RAM memory location that is assigned to BASIC). See section 4.5.2 for further discussion of system control values.

Argument Stack:

The argument stack is used to transfer values between BASIC and the CALL's. Values sent to the CALLed routine are PUSHed before the CALL, and values returned to BASIC from the CALL are POPed after the CALL. Each number placed on the argument stack requires 6 bytes of storage. Therefore, only 34 values may be stored on the argument stack without an overflow error. Special care should be exercised in PUSHing and POPing the correct number of values onto and off of the argument stack to prevent an overflow. The CLEARS command resets the stack to its initialization value.

4.2. Description of Commands

4.2.1 Command: RUN(cr)

Action Taken:

After you type RUN(cr), all variables are set equal to zero, all BASIC evoked interrupts are cleared and program execution begins with the first line number of the selected program. The RUN command and the GOTO statement are the only way you can place the 1771-DB interpreter into the RUN mode from the COMMAND mode. Program execution may be terminated at any time by typing a control-C on the console device.

Variations:

Unlike some Basic interpreters that allow a line number to follow the RUN command (i.e. RUN 100), The 1771-DB does not permit such a variation on the RUN command. Execution always begins with the first line number. To obtain the same functionality as the RUN[ln num] command, use the GOTO[ln num] statement in the direct mode. See statement GOTO.

Example:

```
>10 FOR I=1 TO 3
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3

READY
>
```

4.2.2 Command: CONT(cr)

Action Taken:

If you stop a program by typing a control-C on the console device or by execution of a STOP statement, you can resume execution of the program by typing CONT(cr). Between the stopping and the restarting of the program you may display the values of variables or change the values of variables. However, you may not CONTinue if the program is modified during the STOP or after an error.

Variations:

None.

Example:

```
>10 FOR I=1 TO 10000
>20 PRINT I
>30 NEXT I
>RUN

 1
 2
 3
 4
 5    - (TYPE CONTROL-C ON CONSOLE)

STOP - IN LINE 20

READY
>PRINT I
 6

>I=10

>CONT

 10
 11
 12
```

4.2.3 Command: LIST(cr)

Action taken:

The LIST(cr) command prints the program to the console device. Note that the list command "formats" the program in an easy to read manner. Spaces are inserted after the line number and before and after statements. This feature is designed to aid in the debugging of 1771-DB programs. The "listing" of a program may be terminated at anytime by typing a control-C on the console device. The listing may be interrupted and continued using control S and control Q.

Variations:

Two variations of the LIST command are possible with the 1771-DB.

They are:

LIST [ln num] (cr) and

LIST [ln num] - [ln num] (cr)

The first variation causes the program to be printed from the designated line number (integer) to the end of the program. The second variation causes the program to be printed from the first line number (integer) to the second line number (integer). NOTE - the two line numbers must be separated by a dash -.

Example:

```
READY
>LIST
 10 PRINT "LOOP PROGRAM"
 20 FOR I=1 TO 3
 30 PRINT I
 40 NEXT I
 50 END

READY
>LIST 30
 30 PRINT I
 40 NEXT I
 50 END

READY
>LIST 20-40
 20 FOR I=1 TO 3
 30 PRINT I
 40 NEXT I
```

4.2.4 Command: LIST#(cr)

Action taken:

The LIST#(cr) command lists the program to the device attached to the peripheral port (LIST device). All comments that apply to the LIST command apply to the LIST# command.The LIST#(cr) command is included to permit you to make "hardcopies" of a program. The baud rate is set by a configuration plug and must match your list device (see section 3.2.4). Also, the peripheral port parameters must be configured to match your particular list device (see section 4.7.1).

4.2.5 Command: NEW(cr)

Action taken:

When you enter NEW(cr), the 1771-DB deletes the program that is currently stored in RAM memory. In addition, all variables are set equal to ZERO, all strings and all BASIC evoked interrupts are cleared. The REAL TIME CLOCK, string allocation, and the internal stack pointer value are not affected. In general, NEW (cr) is used simply to erase a program and all variables.

4.2.6 Command: NULL [integer](cr)

Action taken:

The NULL[integer] (cr) command determines how many NULL characters (00H) The 1771-DB will output after a carriage return. After initialization NULL = 0. The NULL command was more important back in the days when a "pure" mechanical printer was the most common I/O device. Most modern printers contain some kind of RAM buffer that virtually eliminates the need to output NULL characters after a carriage return.

Variations:

None.

4.2.7 Command: Control C

This command will stop the execution of the current program and return the 1771-DB to the COMMAND mode. The operation of control C can also be disabled for those situations where program operation must not be aborted (see section 8.2).

4.2.8 Command: Control S

This command will interrupt the BASIC program from scrolling during the execution of a LIST command.

4.2.9 Command: Control Q

This command restarts a LIST command that has been interrupted by a Control S.

4.2.10 Overview of EPROM File Commands

One of the unique and powerful features of the 1771-DB is that it has the ability to execute and SAVE programs in an EPROM. The 1771-DB actually generates all of the timing signals needed to program most EPROM devices.

The 1771-DB can save more than one program in an EPROM. In fact, it can save as many programs as the size of the EPROM memory permits (up to 255). The programs are stored sequentially in the EPROM and any program can be retrieved and executed. This sequential storing of programs is referred to as the EPROM FILE. The following commands permit you to generate and manipulate the EPROM FILE.

4.2.11 Commands: RAM(cr) and ROM [integer] (cr)

Action taken:

These two commands tell the the 1771-DB interpreter whether to select the current program out of RAM or EPROM. The current program is the one that will be displayed during a LIST command and executed when RUN is typed.

RAM

When you enter RAM(cr), the 1771-DB selects the current program from RAM MEMORY. The following formula is used to calculate the available user RAM space:

```
      LEN system control value which contains current RAM program length
      # bytes allocated for strings (first value in the STRING instruction)
      6 * (each array size +1). (The asterisk means multiply).
      8 * each variable used (including each array name)
    + 511
    ------
      V
```

Available user RAM = MTOP - V

ROM

When you enter ROM [integer] (cr), the 1771-DB selects the currentprogram out of EPROM memory. If no integer is typed after the ROMcommand (i.e. ROM (cr)) the 1771-DB defaults to ROM 1. The programs are stored sequentially in EPROM. The integer following the ROM command selects that program. If you attempt to select a program that does not exist (i.e. you type in ROM 8 and only 6 programs are stored in the EPROM) the message ERROR: PROM MODE will be displayed.

The 1771-DB does not transfer the program from EPROM to RAM when the ROM mode is selected. So, you cannot EDIT a program in the ROM mode. If you attempt to edit a program in the ROM mode, by typing in a line number, the message ERROR: PROM MODE will be displayed. The following command to be described, XFER, permits one to transfer a program from EPROM to RAM for editing purposes. The previous RAM content is lost.

Since the ROM command does NOT transfer a program to RAM, it is possible to have different programs in ROM and RAM simultaneously. You can "flip" back and fourth between the two modes at any time. Another added benefit of not transferring a program to RAM is that all of the RAM memory can be used for variable storage if the program is stored in EPROM. The system control value - MTOP always refers to RAM not EPROM, and the system control value, LEN, refers to the currently selected program, whether it resides in RAM or ROM.

Variations: None.

4.2.12 Command: XFER(cr)

Action taken:

The XFER (transfer) command transfers the current selected program in EPROM to RAM and then selects the RAM mode. If you type XFER while the 1771-DB is in the RAM mode, the program stored in RAM is transferred back into RAM and the RAM mode is selected. The net result is that nothing happens except that a few milliseconds of CPU time is used to do a wasted move. After the XFER command is executed, you may edit the program in the same manner any RAM program may be edited. Any program that exists in RAM prior to executing the XFER command, will be cleared by the XFER command.

Variations: None.

4.2.13 Command: PROG(cr)

Action taken:

Note: Before you attempt to program a PROM, read the PROG, PROG1, and PROG2 sections of this chapter thoroughly.

The PROG command programs the resident EPROM with the current selected program. The current selected program may reside in either RAM or EPROM. See section 3.5 for additional information concerning EPROM's.

After you type PROG(cr), the 1771-DB displays the number in the EPROM FILE the program will occupy. Depending upon the length of your program, you may have to wait from several seconds to 30 or 40 seconds for programming to complete.

NOTE: You must disable interrupts prior to the PROG, PROG1, or PROG2 via CALL 8 and enable interrupts after the PROM is burned via CALL 9.

Example:
```
>LIST
10      FOR I=1 TO 10
20      PRINT I
30      NEXT I

READY
>CALL 8 :REM DISABLE INTERRUPTS
>PROG
12

READY
>ROM 12

READY
>CALL 9 :REM ENABLE INTERRUPTS
>LIST
10      FOR I=1 TO 10
20      PRINT I
30      NEXT I

READY
>
```

In this example, the program just placed in the EPROM is the 12th program stored.

Variations: None.

The following program will print out the available PROM space. Enter this program exactly as shown:

```
10      V=8011H
30      IF XBY(V)=1 THEN 50 ELSE V=V+XBY(V)
40      GOTO 30
50      IF XBY(V+1)=55H THEN V=V+2:GOTO 30
60      INPUT "ENTER PROM SIZE 8,16,32K",K
70      K=(K*1024)+8000 H
80      L=K-V
90      PRINT "IF THE LENGTH OF THE PROGRAM TO BE BURNED IS LESS THAN",L
100     PRINT "THERE IS SUFFICIENT SPACE."
110     END
```

Important: If the PROM space available is is exceeded, the PROM will no longer be able to be programmed unless it is erased. Also, in some cases the previously stored programs may be altered. Be sure to use the above listed program to determine memory space prior to burning.

4.2.14 Commands: PROG1(cr) and PROG2(cr)

Action taken:

PROG1

Normally, after power is applied to the 1771-DB device, you must type a "space" character to initialize the 1771-DB's serial port. As a convenience, the 1771-DB contains a PROG1 command. What this command does is program the resident EPROM with the baud rate information. So, the next time the 1771-DB device is "powered up" the module will read this information and initialize the program port with the stored baud rate. The "sign-on" message will be sent to the console immediately after the 1771-DB device completes it's reset sequence. The "space" character no longer needs to be typed. Of course, if the baud rate on the console device is changed a new EPROM must be programmed to make the 1771-DB compatible with the new console.

PROG2

The PROG2 command does everything the PROG1 command does, but instead of "signing-on" and entering the command mode, the 1771-DB device immediately begins executing the first program stored in the resident EPROM. However, if a PROG1 instruction has already been used, then the PROG2 instruction cannot be used.

By using the PROG2 command it is possible to RUN a program on power up and never connect the 1771-DB module to a console. In essence, saving PROG2 information is equivalent to typing a ROM 1, RUN command sequence. This is ideal for control applications, where it is not always possible to have a terminal present. In addition, this feature permits you to write a special initialization sequence in BASIC and generate a custom "sign-on" message for specific applications.

4.3 - Description of Statements

4.3.1 - Statement: CALL [integer]

Mode: COMMAND AND/OR RUN

Type: CONTROL

You use the CALL [integer] STATEMENT to call specially written 1771-DB application programs. Specific call numbers are defined later in this chapter.

Variations:

None.

4.3.2 Statement: CLEAR

Mode: COMMAND AND/OR RUN

Type: CONTROL

The CLEAR statement sets all variables equal to 0 and resets all BASIC evoked interrupts and stacks. This means that after the CLEAR statement is executed an ONTIME statement must be executed before the 1771-DB will acknowledge the internal timer interrupts. ERROR trapping via the ONERR statement will also not occur until an ONERR[integer] statement is executed. The CLEAR statement does not affect the real time clock that is enabled by the CLOCK1 statement. CLEAR also does not reset the memory that has been allocated for strings, so it is not necessary to enter the string [expr], [expr] statement to re-allocate memory for strings after the CLEAR statement is executed. In general, CLEAR is simply used to "erase" all variables.

Variations:

None.

4.3.3 Statement: CLEARI (clear interrupts)

Mode: COMMAND AND/OR RUN

Type: CONTROL

CLEARI

The CLEARI statement clears all of the BASIC evoked interrupts. Specifically, the ONTIME interrupt is DISABLED after the CLEARI statement is executed. The real time clock which is enabled by the CLOCK1 statement is not affected by CLEARI. You can use this statement to selectively DISABLE interrupts during specific sections of your BASIC program. The ONTIME statement must be executed again before the specific interrupts will be enabled.

NOTE: When the CLEARI statement is LISTED it will appear as CLEAR I.

4.3.4 Statements: CLOCK1 and CLOCK0

Mode: COMMAND AND/OR RUN

Type: CONTROL

CLOCK1

The CLOCK1 statement enables the real time clock feature resident on the the 1771-DB device. The special function operator TIME is incremented once every 5 milliseconds after the CLOCK1 statement has been executed. The CLOCK1 STATEMENT uses TIMER/COUNTER 0 in the 13-bit mode to generate an interrupt once every 5 milliseconds. Because of this, the special function operator TIME has a resolution of 5 milliseconds. The crystal value is 11.0592 MHz.The special function operator TIME counts from 0 to 65535.995 seconds. After reaching a count of 65535.995 seconds TIME overflows back to a count of zero. The interrupts associated with the CLOCK1 statement cause the 1771-DB programs to run at about 99.6% of normal speed. That means that the interrupt handling for the REAL TIME CLOCK feature only consumes about .4% of the total CPU time.

CLOCK0

The CLOCK0 (zero) statement disables or "turns off" the real time clock feature. After CLOCK0 is executed, the special function operator TIME will no longer increment. The CLOCK0 statement also returns control of the interrupts associated with TIMER/COUNTER 0 back to the user. CLOCK0 is the only 1771-DB statement that can disable the real time clock. CLEAR and CLEARI will NOT disable the real time clock.

Variations:

None

Note: CLOCK1 and CLOCK0 are independent of the wall clock.

4.3.5 Statements: DATA - READ - RESTORE

Mode: RUN

Type: Assignment

DATA

DATA specifies expressions that may be retrieved by a READ statement. If multiple expressions per line are used, they MUST be separated by a comma.

READ

READ retrieves the expressions that are specified in the DATA statement and assigns the value of the expression to the variable in the READ statement. The READ statement must always be followed by one or more variables. If more than one variable follows a READ statement, they must be separated by a comma.

RESTORE

RESTORE "resets" the internal read pointer back to the beginning of the data so that it may be read again.

Example:

```
>10 FOR I=1 TO 3
>20 READ A,B
>30 PRINT A,B
>40 NEXT I
>50 RESTORE
>60 READ A,B
>70 PRINT A,B
>80 DATA 10,20,10/2,20/2,SIN(PI),COS(PI)
>RUN

     10  20
      5  10
      0 -1
     10  20
```

Variations:

None

Explanation of previous example:

Every time a READ statement is encountered the next consecutive expression in the DATA statement is evaluated and assigned to the variable in the READ statement. DATA statements may be placed anywhere within a program, they will not be executed nor will they cause an error. DATA statements are considered to be chained together and appear to be one big DATA statement. If at anytime all the data has been read and another READ statement is executed then the program is terminated and the message ERROR: NO DATA - IN LINE XX is printed to the console device.

4.3.6 - Statement: DIM

Mode: COMMAND AND/OR RUN

Type: Assignment

DIM reserves storage for matrices. The storage area is first assumed to be zero. Matrices in the 1771-DB may have only one dimension and the size of the dimensioned array may not exceed 254 elements. Once a variable is dimensioned in a program it may not be re-dimensioned. An attempt to re-dimension an array will cause an ARRAY SIZE ERROR. If an arrayed variable is used that has not been dimensioned by the DIM statement, BASIC will assign a default value of 10 to the array size. All arrays are set equal to zero when the RUN command, NEW command, or the CLEAR statement is executed. The number of bytes allocated for an array is 6 times the (array size plus 1). So, the array A(100) would require 606 bytes of storage. Memory size usually limits the size of a dimensioned array.

Variations:

More than one variable can be dimensioned by a single DIM statement.
i.e. DIM A(10), B(15), A1(20).

Example: Default error on attempt to re-dimension array.

```
>10 A(5)=10      - BASIC ASSIGNS DEFAULT OF 10 TO ARRAY SIZE HERE
>20 DIM A(5)     - ARRAY CANNOT BE RE-DIMENSIONED
>RUN

ERROR: ARRAY SIZE - IN LINE 20

20    DIM A(5)
----------X
```

4.3.7 Statements: DO - UNTIL [rel expr]

Mode: RUN

Type: CONTROL

The DO - UNTIL [rel expr] instruction provides a means of "loop control" within a 1771-DB program. All statements between the DO and the UNTIL [rel expr] will be executed until the relational expression following the UNTIL statement is TRUE. DO - UNTIL loops may be nested.

Examples:

SIMPLE DO-UNTIL

```
>10 A=0
>20 DO
>30 A=A+1
>40 PRINT A
>50 UNTIL A=4
>60 PRINT "DONE"
>70 END
>RUN


 1
 2
 3
 4
DONE

READY
>
```

NESTED DO-UNTIL

```
>10 DO
>20 A=A+1
>30 DO
>40 B=B+1
>50 PRINT A,B,A*B
>60 UNTIL B=3
>70 B=0
>80 UNTIL A=3
>90 END

 1  1  1
 1  2  2
 1  3  3
 2  1  2
 2  2  4
 2  3  6
 3  1  3
 3  2  6
 3  3  9

READY
>
```

Variations:

None

4.3.8 Statements: DO - WHILE [rel expr]

Mode: RUN

Type: CONTROL

The DO - WHILE [rel expr] instruction provides a means of "loop control" within a 1771-DB program. This operation of this statement is similar to the DO - UNTIL [rel expr] except that all statements between the DO and the WHILE [rel expr] will be executed as long as the relational expression following the WHILE statement is true. DO - WHILE and DO - UNTIL statements can be nested.

Examples:

SIMPLE DO-WHILE

```
>10 DO
>20 A=A+1
>30 PRINT A
>40 WHILE A<4
>50 PRINT "DONE"
>60 END
>RUN

 1
 2
 3
 4
DONE

READY
>
```

NESTED DO-WHILE - DO-UNTIL

```
>10 DO
>20 A=A+1
>30 B=B+1
>40 PRINT A,B,A*B
>50 WHILE B<>3
>60 B=0
>70 UNTIL A=3
>80 END

 1  1  1
 1  2  2
 1  3  3
 2  1  2
 2  2  4
 2  3  6
 3  1  3
 3  2  6
 3  3  9

READY
>
```

Variations:

None

4.3.9 Statement: END

Mode: RUN

Type: CONTROL

The END statement terminates program execution. The continue command, CONT will not operate if the END statement is used to terminate execution (i.e. a CAN'T CONTINUE ERROR will be printed to the console). The last statement in a 1771-DB program will automatically terminate program execution if no END statement is used. As a good programming practice, an END statement should always be used to terminate a program.

Examples:

LAST STATEMENT TERMINATION

```
>10 FOR I=1 TO 4
>20 PRINT I
>30 NEXT I
>40 END
>RUN
 1
 2
 3
 4

READY
>
```

END STATEMENT TERMINATION

```
>10 FOR I=1 TO 4
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I
>110 RETURN
>RUN

 1
 2
 3
 4

READY
>
```

Variations:

None

4.3.10 Statements: FOR - TO - (STEP) - NEXT

Mode: RUN

Type: CONTROL

The FOR - TO - (STEP) - NEXT statements are used to set up and control loops.

Example:

```
5  B=0 : C=10 : D=2
10 FOR A=B TO C STEP D
20 PRINT A
30 NEXT A
40 END
```

Since B=0, C=10, and D=2, the PRINT statement at line 20 will be executed 6 times. The values of "A" that will be printed are 0, 2, 4, 6, 8, 10. "A" represents the name of the index or loop counter. The value of "B" is the starting value of the index, the value of "C" is the limit value of the index, and the value of "D" is the increment to the index. If the STEP statement and the value "D" are omitted, the increment value defaults to 1, therefore, STEP is an optional statement. The NEXT statement causes the value of "D" to be added to the index. The index is then compared to the value of "C", the limit. If the index is less than or equal to the limit, control will be transferred back to the statement after the FOR statement. Stepping "backwards" (i.e., FOR I = 100 TO 1 STEP-1) is permitted in the 1771-DB. The index may not be omitted from the NEXT statement in the 1771-DB (e.g., The NEXT statement must always be followed by the appropriate variable). FOR-NEXT loops may be nested up to 9 times.

Examples:

```
>10 FOR I=1 TO 4
>20 PRINT I,
>30 NEXT I
>40 END
>RUN

 1 2 3 4

READY
>
```

```
>10 FOR I=0 TO 8 STEP 2
 >20 PRINT I
>30 NEXT I
>40 END
>RUN

 0
 2
 4
 6
 8

READY
>
```

4.3.11 Statements: GOSUB[ln num] - RETURN

Mode: RUN

Type: CONTROL

GOSUB

The GOSUB [ln num] statement will cause the 1771-DB to transfer control of the program directly to the line number ([ln num]) following the GOSUB statement. In addition, the GOSUB statement saves the location of the statement following GOSUB on the control stack so that a RETURN statement can be performed to return control.

RETURN

This statement is used to "return" control back to the statement following the most recently executed GOSUB STATEMENT. The GOSUB-RETURN sequence can be "nested" meaning that a subroutine called by the GOSUB statement can call another subroutine with another GOSUB statement.

Examples:

SIMPLE SUBROUTINE

```
>10 FOR I=1 TO 5
>20 GOSUB 100
>30 NEXT I
>40 END
>100 PRINT I
>110 RETURN
>100 PRINT I,
>RUN
1
2
3
4
5

READY
>
```

NESTED SUBROUTINES

```
>10 FOR I=1 TO 3
>20 GOSUB 100
>30 NEXT I
>40 END
>110 GOSUB 200
>120 RETURN
>200 PRINT I*I
>210 RETURN


>RUN

1 1
2 4
3 9

READY
>
```

4.3.12 Statement: GOTO [ln num]

Mode: COMMAND AND/OR RUN

Type: CONTROL

The GOTO [ln num] statement will cause BASIC to transfer control directly to the line number ([ln num]) following the GOTO statement.

Example:

```
50 GOTO 100
```

Will, if line 100 exist, cause execution of the program to resume at line 100. If line number 100 does not exist the message ERROR: INVALID LINE NUMBER will be printed to the console device.

Unlike the RUN command the GOTO statement, if executed in the COMMAND mode, does not clear the variable storage space or interrupts. However, if the GOTO statement is executed in the COMMAND mode after a line has been edited, the 1771-DB will clear the variable storage space and all BASIC evoked interrupts. This is a necessity because the variable storage and the BASIC program reside in the same RAM memory. So editing a program can destroy variables.

NOTE - Because of the way the 1771-DB's text interpreter processes a line, when an INVALID LINE NUMBER ERROR occurs on the GOTO, GOSUB, ON GOTO, and ON GOSUB statements the line after the GOTO or GOSUB statement will be printed out in the error message.

Example:

```
>10 GOTO 100
>20 PRINT X
>RUN

ERROR: INVALID LINE NUMBER - IN LINE 20

20    PRINT X
-------------X
```

4.3.13 Statements: ON [expr] GOTO[ln num], [ln num],...[ln num]

ON [expr] GOSUB[ln num], [ln num],...[ln num]

Mode: RUN

Type: CONTROL

The value of the expression following the ON statement is the number in the line list that control will be transferred to.

Example: 10 ON Q GOTO 100,200,300

If Q was equal to 0, control would be transferred to line number 100. If Q was equal to 1, control would be transferred to line number 200. If Q was equal to 2, GOTO line 300, etc. All comments that apply to GOTO and GOSUB apply to the ON statement. If Q is less than ZERO a BAD ARGUMENT ERROR will be generated. If Q is greater than the line number list following the GOTO or GOSUB statement, a BAD SYNTAX ERROR will be generated. The ON statement provides "conditional branching" options within the constructs of a 1771-DB program.

4.3.14 Statements: IF - THEN - ELSE

Mode: RUN

Type: CONTROL

The IF statement sets up a conditional test. The generalized form of the IF - THEN - ELSE statement is as follows:

[ln num] IF [rel expr] THEN valid statement ELSE valid statement

A specific example is as follows:

```
>10 IF A=100 THEN A=0 ELSE A=A+1
```

Upon execution of line 10 IF A is equal to 100, THEN A would be assigned a value of 0. IF A does not equal 100, A would be assigned a value of A+1. If it is desired to transfer control to different line numbers using the IF statement, The GOTO statement may be omitted. The following examples would yield the same results:

```
>20 IF INT(A)< 10 THEN GOTO 100 ELSE GOTO 200  and

>20 IF INT(A)< 10 THEN 100 ELSE 200
```

Additionally, the THEN statement can be replaced by any valid 1771-DB statement, as shown below:

```
>30 IF A<>10 THEN PRINT A ELSE 10

>30 IF A<>10 PRINT A ELSE 10
```

Multiple statements may be executed following the "THEN" or "ELSE" provided that colons are used to separate them.

Example:

```
>30 If A<>10 THEN PRINT A:GOTO 150 ELSE 10

>30 IF A<>10 PRINT A:,GOTO 150 ELSE 10
```

In these examples, if A does not equal 10 then both PRINT A and GOTO 150 would be executed. If A=10, then control will pass to 10.

The ELSE statement may be omitted. If it is control will pass to the next statement.

Example:

```
>20 IF A=10 THEN 40

>30 PRINT A
```

In this example, if A equals 10 then control would be passed to line number 40. If A does not equal 10 line number 30 would be executed.

4.3.15 Statements: INPUT

Mode: RUN

Type: INPUT/OUTPUT

The INPUT statement allows users to enter data from the console during program execution. One or more variables may be assigned data with a single input statement. The variables must be separated by a comma.

Example: INPUT A,B

Would cause the printing of a question mark (?) on the console device as a prompt to the operator to input two numbers separated by a comma. If the operator does not enter enough data, then the 1771-DB responds by outputting the message TRY AGAIN to the console device.

Example:

```
>10 INPUT A,B
>20 PRINT A,B
>RUN

?1

 TRY AGAIN

 ?1,2
  1  2

 READY
```

The INPUT statement may be written so that a descriptive prompt is printed to tell the user what to type. The message to be printed is placed in quotes after the INPUT statement. If a comma appears before the first variable on the input list, the question mark prompt character will not be displayed.

Examples:

```
>10 INPUT"ENTER A NUMBER"A

>20 PRINT SQR(A)
>30 END
>RUN

ENTER A NUMBER
?100
 10
```

```
>10 INPUT"ENTER A NUMBER-",A

>20 PRINT SQR(A)
>30 END
>RUN

ENTER A NUMBER-100
 10
```

Strings can also be assigned with an INPUT statement. Strings are always terminated with a carriage return (cr). So, if more than one string input is requested with a single INPUT statement, the 1771-DB will prompt you with a question mark.

EXAMPLES:

```
>10 STRING 110,10
>20 INPUT "NAME: ",$(1)

>30 PRINT "HI ",$(1)
>40 END
>RUN

NAME: SUSAN
HI SUSAN

READY
```

```
>10 STRING 110,10
>20 INPUT "NAMES: ",$(1),$(2)

>30 PRINT "HI ",$(1)," AND ",$(2)
>40 END
>RUN

NAMES: BILL
?ANN
HI BILL AND ANN

READY
```

Additionally, strings and variables can be assigned with a single INPUT statement.

Example:

```
>10 STRING 100,10
>20 INPUT"NAME(CR), AGE - ",$(1),A
>30 PRINT "HELLO ",$(1),", YOU ARE ",A,"YEARS OLD"
>40 END
>RUN


NAME(CR), AGE - FRED
?15
HELLO FRED, YOU ARE 15 YEARS OLD

READY
>
```

4.3.16 Statement: LET

Mode: COMMAND AND/OR RUN

Type: ASSIGNMENT

The LET statement is used to assign a variable to the value of an expression. The generalized form of LET is:

```
LET [var] = [expr]
```

Examples:

```
LET A = 10*SIN(B)/100 or

LET A = A + 1
```

Note that the = sign used in the LET statement is not equality operator, but rather a "replacement" operator and that the statement should be read A is replaced by A plus one. The word LET is always optional, i.e.

```
LET A = 2 is the same as A = 2
```

When LET is omitted the LET statement is called an IMPLIED LET. This document will use the word LET to refer to both the LET statement and the IMPLIED LET statement.

The LET statement is also used to assign the string variables, i.e:

```
LET $(1)="THIS IS A STRING" or

LET $(2)=$(1)
```

Before strings can be assigned the STRING [expr],[expr] statement must be executed, or else a MEMORY ALLOCATION ERROR will occur (see section 4.3.27).

4.3.17 Statement: ONERR[ln num]

Mode: RUN

Type: CONTROL

The ONERR[ln num] statement lets the programmer handle arithmetic errors, should they occur, during program execution. Only ARITH. OVERFLOW, ARITH. UNDERFLOW, DIVIDE BY ZERO, and BAD ARGUMENT errors can be "trapped" by the ONERR statement, all other errors are not. If an arithmetic error occurs after the ONERR statement is executed, the 1771-DB interpreter will pass control to the line number following the ONERR[ln num] statement. The programmer can handle the error condition in any manner suitable to the particular application. Typically, the ONERR[ln num] statement should be viewed as an easy way to handle errors that occur when the user provides inappropriate data to an INPUT statement.

With the ONERR[ln num] statement, the programmer has the option of determining what type of error occurred. This is done by examining external memory location 257 (101H) after the error condition is trapped.

The error codes are as follows:

ERROR CODE = 10 - DIVIDE BY ZERO

ERROR CODE = 20 - ARITH. OVERFLOW

ERROR CODE = 30 - ARITH. UNDERFLOW

ERROR CODE = 40 - BAD ARGUMENT

This location may be examined by using an XBY (257) statement.

4.3.18 Statement: ONTIME [expr],[ln num]

Mode: RUN

Type: CONTROL

Since the 1771-DB processes a line in the millisecond time frame and the timer/counters on the processor operate in the microsecond time frame, there is an inherent incompatibility between the timer/counters on the processor and the 1771-DB. To help solve this situation the ONTIME [expr],[ln num] statement was devised. What ONTIME does is generate an interrupt every time the special function operator, TIME, is equal to or greater than the expression following the ONTIME statement. Actually, only the integer portion of TIME is compared to the integer portion of the expression. The interrupt forces a GOSUB to the line number ([ln num]) following the expression ([expr]) in the ONTIME statement.

Since the ONTIME statement uses the special function operator, TIME, the CLOCK1 statement must be executed in order for ONTIME to operate. If CLOCK1 is not executed the special function operator, TIME, will never increment and not much will happen.

Since The ONTIME statement generates an interrupt when TIME is greater than or equal to the expression following the ONTIME statement, how can periodic interrupts be generated? That's easy, the ONTIME statement must be executed again in the interrupt routine:

Example:
```
>10 TIME=0 : CLOCK1 : ONTIME 2,100 : DO
>20 WHILE TIME<10 : END
>100 PRINT "TIMER INTERRUPT AT -",TIME,"SECONDS"
>110 ONTIME TIME+2,100 : RETI
>RUN

TIMER INTERRUPT AT - 2.045 SECONDS
TIMER INTERRUPT AT - 4.045 SECONDS
TIMER INTERRUPT AT - 6.045 SECONDS
TIMER INTERRUPT AT - 8.045 SECONDS
TIMER INTERRUPT AT - 10.045 SECONDS

READY
```

The terminal used in this example was running at 4800 baud and it takes about 45 milliseconds to print the message TIMER INTERRUPT AT -" ". The result is the TIME that was printed out was 45 milliseconds greater than the time the interrupt was supposed to be generated.

If the programmer does not want this delay, a variable should be assigned to the special function operator, TIME, at the beginning of the interrupt routine.

Example:
```
>10 TIME=0 : CLOCK1 : ONTIME 2,100: DO
>20 WHILE TIME<10 : END
>100 A=TIME
>110 PRINT "TIMER INTERRUPT AT -",A,"SECONDS"
>120 ONTIME A+2,100 : RETI
>RUN

TIMER INTERRUPT AT - 2 SECONDS
TIMER INTERRUPT AT - 4 SECONDS
TIMER INTERRUPT AT - 6 SECONDS
TIMER INTERRUPT AT - 8 SECONDS
TIMER INTERRUPT AT - 10 SECONDS

READY
```

The ONTIME interrupt routine must be exited with a RETI statement. Failure to do this will "lock-out" all future interrupts.

The ONTIME statement in the 1771-DB is unique, relative to most BASICS. This powerful statements eliminates the need for the user to "test" the value of the TIME operator periodically throughout the BASIC program.

4.3.19 Statement: PRINT or P.

Mode: COMMAND and/or RUN

Type: INPUT/OUTPUT

The PRINT statement directs the 1771-DB to output to the console device. The value of expressions, strings, literal values, variables or text strings may be printed out. The various forms may be combined in the print list by separating them with commas. If the list is terminated with a comma, the carriage return/line feed will be suppressed. P. is a "shorthand" notation for PRINT.

Examples:

```
>PRINT 10*10,3*3      >PRINT "MCS-51"        >PRINT 5,1E3

 100  9               MCS-51                  5  1000
```

Values are printed next to one another with two intervening blanks. A PRINT statement with no arguments causes a carriage return/line feed sequence to be sent to the console device.

Special Print Formatting Statements

TAB([expr])

Use the TAB([expr]) function in the PRINT statement to cause data to be printed out in exact locations on the output device. TAB([expr]) tells the 1771-DB which position to begin printing the next value in the print list. If the printhead or cursor is on or beyond the specified TAB position, the 1771-DB will ignore the TAB function.

Example:

```
>PRINT TAB(5),"X",TAB(10),"Y"
     X    Y
```

SPC([expr])

Use the SPC([expr]) function in the PRINT statement to cause the 1771-DB to output the number of spaces in the SPC argument.

Example:

```
>PRINT A,SPC(5),B
```

may be used to place an additional 5 spaces between the A and B over and above the two that would normally be printed.

CR

When you use CR in a PRINT statement, it will force a carriage return, but no line feed. This can be used to create one line on a CRT device that is repeatedly updated.

Example:

```
>10 FOR I=1 TO 1000
>20 PRINT I,CR,
>30 NEXT I
```

will cause the output to remain only on one line. No line feed will ever be sent to the console device.

USING(special characters)

The USING function is used to tell the 1771-DB what format to display the values that are printed. The 1771-DB "stores" the desired format after the USING statement is executed. So, all outputs following a USING statement will be in the format evoked by the last USING statement executed. The USING statement need not be executed within every PRINT statement unless the programmer wants to change the format. U. is a "shorthand" notation for USING.

Note: The USING statement applies to numbers following it until another USING statement is encountered.

The options for USING are as follows:

USING(Fx) - This will force the 1771-DB to output all numbers using the floating point format. The value of x determines how many significant digits will be printed. If x equals 0, the 1771-DB will not output any trailing zeros, so the number of digits will vary depending upon the number. The 1771-DB will always output at least 3 significant digits even if x is 1 or 2. The maximum value for x is 8.

Example:

```
>10 PRINT USING(F3),1,2,3
>20 PRINT USING(F4),1,2,3
>30 PRINT USING(F5),1,2,3
>40 FOR I=10 TO 40 STEP 10
>50 PRINT I
>60 NEXT I
>RUN

 1.00 E 0  2.00 E 0  3.00 E 0
 1.000 E 0  2.000 E 0  3.000 E 0
 1.0000 E 0  2.0000 E 0  3.0000 E 0
 1.0000 E+1
 2.0000 E+1
 3.0000 E+1
 4.0000 E+1
 READY
```

USING(#.#)- This will force the 1771-DB to output all numbers using an integer and/or fraction format. The number of "#"'s before the decimal point represents the number of significant integer digits that will be printed and the number of "#"'s after the decimal point represents the number of digits that will be printed in the fraction. The decimal point may be omitted, in which case only integers will be printed. USING may be abbreviated U. USING(###.###), USING(######) and USING(######.##) are all valid in the 1771-DB. The maximum number of "#" characters is 8. If the 1771-DB cannot output the value in the desired format (usually because the value is too large) a question mark (?) will be printed to console device, then BASIC will output the number in the FREE FORMAT described below.

Example:

```
>10 PRINT USING(##.##),1,2,3
>20 FOR I=1 TO 120 STEP 20
>30 PRINT I
>40 NEXT I
>50 END
>RUN

  1.00    2.00    3.00
  1.00
 21.00
 41.00
 61.00
 81.00
? 101

READY
```

NOTE: The USING(Fx) and the USING(#.#) formats will always "align" the decimal points when printing a number. This feature makes displayed columns of numbers easy to read.

USING(0) - This argument lets the 1771-DB determine what format to use. The rules are simple, if the number is between +/-99999999 and +/-.1, BASIC will display integers and fractions. If it is out of this range, BASIC will use the USING(F0) format. Leading and trailing zeros will always be suppressed. After reset, the 1771-DB is placed in the USING(0) format.

4.3.20 Statement: PRINT# or P.#

Mode: COMMAND and/or RUN

Type: INPUT/OUTPUT

The PRINT# or P.# statement does the same thing as the PRINT or P. statement except that the output is directed to the list device instead of the console device. The baud rate must be selected by a configuration plug and must match your device (see section 3.2.4). The peripheral port parameters must also be configured to match your device (see section 4.7.1). All comments that apply to the PRINT or P. statement apply to the PRINT# or P.# statement. P.# is a "shorthand" notation for PRINT#.

4.3.21 Statements: PH0., PH1., PH0.#, PH1.#

Mode: COMMAND and/or RUN

Type: INPUT/OUTPUT

The PH0. and PH1. statements do the same thing as the PRINT statement except that the values are printed out in a hexadecimal format. The PH0. statement suppresses two leading zeros if the number to be printed is less than 255 (0FFH). The PH1. statement always prints out four hexadecimal digits. The character "H" is always printed after the number when PH0. or PH1. is used to direct an output. The values printed are always truncated integers. If the number to be printed is not within the range of valid integer (i.e. between 0 and 65535 (0FFFFH) inclusive), the 1771-DB will default to the normal mode of print. If this happens no "H" will be printed out after the value. Since integers can be entered in either decimal or hexadecimal form the statements PRINT, PH0., and PH1. can be used to perform decimal to hexadecimal and hexadecimal to decimal conversion. All comments that apply to the PRINT statement apply to the PH0. and PH1. statements. PH0.# and PH1.# do the same thing as PH0. and PH1. respectively, except that the output is directed to the list device instead of the console device. The baud rate and peripheral port parameters must match your device (see sections 3.2.4 and 4.7.1 respectively).

Examples:

```
>PH0. 2*2       >PH1. 2*2       >PRINT 99H      >PH0. 100

 04H             0004H           153             64H

>PH0. 1000      >PH1. 1000      >P. 3E8H        >PH0. PI

 3E8H            03E8H           1000            03H
```

4.3.22 Statement: PUSH[expr]

Mode: COMMAND AND / OR RUN

Type: ASSIGNMENT

The arithmetic expression, or expressions following the PUSH statement are sequentially placed on the 1771-DB's ARGUMENT STACK. This statement, in conjunction with the POP statement provide a simple means of passing parameters to the CALL routines. In addition, The PUSH and POP statements can be used to pass parameters to BASIC subroutines and to "SWAP" variables. The last value PUSHED onto the ARGUMENT STACK will be the first value POPPED off the ARGUMENT STACK.

Variations:

More than one expression can be pushed onto the ARGUMENT stack with a single PUSH statement. The expressions are simply followed by a comma: PUSH[expr],[expr],........[expr]). The last value PUSHED onto the ARGUMENT STACK will be the last expression [expr] encountered in the PUSH STATEMENT.

Examples:

```
SWAPPING                CALL
VARIABLES               ROUTINE

>10 A=10                >10 PUSH 3
>20 B=20                >20 CALL 10
>30 PRINT A,B           >30 POP W
>40 PUSH A,B            >40 REM PUSH VALUE ON STACK, CALL
>50 POP A,B             >50 REM CONVERSION, POP RESULT IN W

>60 PRINT A,B           >60 END
>70 END
>RUN

10  20
20  10
READY
<
```

4.3.23 Statement: POP[var]

Mode: COMMAND AND / OR RUN

Type: ASSIGNMENT

The top of the ARGUMENT STACK is assigned to the variable following the POP statement and the ARGUMENT STACK is "POPPED" (i.e. incremented by 6). Values can be placed on the stack by either the PUSH statement or by assembly language CALLS. NOTE - If a POP statement is executed and no number is on the ARGUMENT STACK, an A-STACK ERROR will occur.

Variations:

More than one variable can be popped off the ARGUMENT stack with a single POP statement. The variables are simply followed by a comma (i.e. POP [var],[var],........[var]).

Examples:

See PUSH statement.

Comment:

The PUSH and POP statements are unique to the 1771-DB. These powerful statements can be used to "get around" the GLOBAL variable problems so often encountered in BASIC programs. This problem arises because in BASIC the "main" program and all subroutines used by the main program

are required to use the same variable names (i.e. GLOBAL VARIABLES). It is not always convenient to use the same variables in a subroutine as in the main program and you often see programs re-assign a number of variables (i.e. A=Q) before a GOSUB statement is executed. If you reserve some variable names just for subroutines (i.e. S1, S2) and passes variables on the stack as shown in the previous example, you will avoid any GLOBAL variable problems in the 1771-DB.

The PUSH and POP statements will accept dimensioned variables i.e., A(4), and SI(12), as well as scalar variables. This feature can be particularly useful when large amounts of data must be PUSHED or POPPED when using CALL routines.

Example:

```
40  FOR I=1 TO 64
50  PUSH I
60  CALL 10
70  POP A(I)
80  NEXT I
```

4.3.24 Statement: REM

Mode: RUN

Type: CONTROL - Performs no operation

REM is short for REMark. It does nothing, but allows you to add comments to a program. Comments are usually needed to make a program a little easier to understand. Once a REM statement appears on a line the entire line is assumed to be a remark, so a REM statement may not be terminated by a colon (:), however, it may be placed after a colon. This allows you to place a comment on each line.

Examples:

```
>10 REM INPUT ONE VARIABLE
>20 INPUT A
>30 REM INPUT ANOTHER VARIABLE
>40 INPUT B
>50 REM MULTIPLY THE TWO
>60 Z=A*B
>70 REM PRINT THE ANSWER
>80 PRINT Z
>90 END
>10 INPUT A : REM INPUT ONE VARIABLE
>20 INPUT B : REM INPUT ANOTHER VARIABLE
>30 Z=A*B : REM MULTIPLY THE TWO
>40 PRINT Z :REM PRINT THE ANSWER
>50 END
```

The following will NOT work because the entire line would be interpreted as a REMark, so the PRINT statement would not be executed:

```
>10 REM PRINT THE NUMBER : PRINT A
```

4.3.25 Statement: RETI

Mode: RUN

Type: CONTROL

Use the RETI statement to exit from the ONTIME interrupts that are handled by a 1771-DB program. The RETI statement does the same thing as the RETURN statement except that it also clears a software interrupt flags so interrupts can again be acknowledged. If you fail to execute the RETI statement in the interrupt procedure, all future interrupts will be ignored.

4.3.26 Statement: STOP

Mode: RUN

Type: CONTROL

The STOP statement allows you to break program execution at specific points in a program. After a program is STOPped variables can be displayed and/or modified. Program execution may be resumed with a CONTinue command. The purpose of the STOP statement is to allow for easy program "debugging". More details of the STOP-CONT sequence are covered in the DESCRIPTION OF COMMANDS - CONT in section 4.4.2 of this manual.

Example:

```
>10 FOR I=1 TO 100
>20 PRINT I
>30 STOP
>40 NEXT I
>RUN

 1
 STOP - IN LINE 40

 READY
>CONT

 2
```

Note that the line number printed out after the STOP statement is executed is the line number following the STOP statement, not the line number that contains the STOP statement.

4.3.27 Statement: STRING

Mode: COMMAND and/or RUN

Type: CONTROL

The STRING [expr],[expr] statement allocates memory for strings. Initially, no memory is allocated for strings. If you attempt to define a string with a statement such as LET $(1)="HELLO" before memory has been allocated for strings, a MEMORY ALLOCATION ERROR will be generated. The first expression in the STRING [expr],[expr] statement is the total number of bytes you wish to allocate for string

storage. The second expression denotes the maximum number of bytes that are in each string. These two numbers determine the total number of defined string variables.

The 1771-DB requires one additional byte for each string, plus one additional byte overall. The additional character for each string is allocated for the carriage return character that terminates the string. This means that the statement STRING 100,10 would allocate enough memory for 9 string variables, ranging from $(0) to $(8) and all of the 100 allocated bytes would be used. Note that $(0) is a valid string in the 1771-DB. Refer to section 4.9 for further discussion of strings and example programs for string memory allocation.

After memory is allocated for string storage, neither commands, such as NEW nor statements, such as CLEAR, will "de-allocate" this memory. The only way memory can be de-allocated is to execute a STRING 0,0 statement. STRING 0,0 will allocate no memory to string variables.

Note: Every time the STRING [expr],[expr] statement is executed, the 1771-DB basic executes the equivalent of a CLEAR statement. This is a necessity because string variables and numeric variables occupy the same external memory space. So, after the STRING statement is executed, all variables are "wiped-out". Because of this, string memory allocation should be performed early in a program (like the first statement or so) and string memory should never be "re-allocated" unless the programmer is willing to destroy all defined variables.

4.4 Description of Arithmetic/Logical Operators and Expressions

4.4.1 Dual Operand (DYADIC) Operators

The 1771-DB contains a complete set of arithmetical and logical operators. Operators are divided into two groups, dual operand or dyadic operators and single operand or unary operators. The generalized form of all dual operand instructions is as follows:

(expr) OP (expr), where OP is one of the following operators:

\+ Addition Operator

Example:
```
PRINT 3+2
5
```

/ Division Operator

Example:
```
PRINT 100/5
20
```

** Exponentation Operator

Raises the first expression to the power of the second expression. The power any number can be raised to is limited to 255.

Example:
```
PRINT 2**3
8
```

* Multiplication Operator

Example:
```
PRINT 3*3
9
```

- Subtraction Operator

Example:
```
PRINT 9-6

3
```

.AND. Logical AND Operator

Example:
```
PRINT 3.AND.2

2
```

.OR. Logical OR Operator

Example:
```
PRINT 1.OR.4
5.
```

.XOR. Logical EXCLUSIVE OR operator

Example:
```
PRINT 7.XOR.6
1
```

Comments on logical operators .AND.,.OR., and .XOR.

These operators perform a BIT-WISE logical function on valid INTEGERS. That means both arguments for these operators must be between 0 and 65535 (OFFFFH) inclusive. If they are not, the 1771-DB will generate a BAD ARGUMENT ERROR. All non-integer values are truncated, not rounded.

4.4.2 Unary Operators

4.4.2.1 General Purpose

ABS([expr])

Returns the absolute value of the expression.

Examples:	PRINT ABS(5)	PRINT ABS(-5)
	5	5

NOT([expr])

Returns a 16 bit one's complement of the expression. The expression must be a valid integer (i.e.between 0 and 65535 (0FFFFH) inclusive). Non-integers will be truncated, not rounded.

Examples:	PRINT NOT(65000)	PRINT NOT(0)
	535	65535

INT([expr])

Returns the integer portion of the expression.

Examples:	PRINT INT(3.7)	PRINT INT(100.876)
	3	100

SGN([expr])

Will return a value of +1 if the argument is greater than zero, zero if the argument is equal to zero, and -1 if the argument is less than zero.

Examples:	PRINT SGN(52)	PRINT SGN(0)	PRINT SGN(-8)
	1	0	-1

SQR([expr])

Returns the square root of the argument. The argument may not be less than zero. The result returned will be accurate to within +/- a value of 5 on the least significant digit.

Examples:	PRINT SQR(9)	PRINT SQR(45)	PRINT SQR(100)
	3	6.7082035	10

RND

Returns a pseudo-random number in the range between 0 and 1 inclusive. The RND operator uses a 16-bit binary seed and generates 65536 pseudo-random numbers before repeating the sequence. The numbers generated are specifically between 0/65535 and 65535/65535 inclusive. Unlike most BASICS, the RND operator in the 1771-DB does not require an argument or a dummy argument. In fact, if an argument is placed after the RND operator, a BAD SYNTAX error will occur.

```
Examples:   PRINT RND
            .30278477
```

PI

PI is not really an operator, it is a stored constant. In the 1771-DB, PI is stored as 3.1415926.

4.4.2.2 Log Functions

LOG([expr])

Returns the natural logarithm of the argument. The argument must be greater than 0. This calculation is carried out to 7 significant digits.

```
Examples:   PRINT LOG(12)     PRINT LOG(EXP(1))
            2.484906          1
```

EXP([expr])

This function raises the number "e" (2.7182818) to the power of the argument.

```
Examples:   PRINT EXP(1)      PRINT EXP(LOG(2))
            2.7182818         2
```

4.4.2.3 Trig Functions

SIN([expr]

Returns the sine of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between + 200000.

```
Examples:   PRINT SIN(PI/4)   PRINT SIN(0)
            .7071067          0
```

COS([expr])

Returns the cosine of the argument. The argument is expressed in radians. Calculations are carried out to 7 significant digits. The argument must be between + 200000.

```
Examples:   PRINT COS(PI/4)    PRINT COS(0)
            .7071067           1
```

TAN([expr])

Returns the tangent of the argument. The argument is expressed in radians. The argument must be between + 200000.

```
Examples:   PRINT TAN(PI/4)    PRINT TAN(0)
            1                  0
```

ATN([expr])

Returns the arctangent of the argument. The result is in radians. Calculations are carried out to 7 significant digits. The ATN operator returns a result between -PI/2 (3.1415926/2) and PI/2.

```
Examples:   PRINT ATN(PI)      PRINT ATN(1)
            1.2626272          .78539804
```

Comments on Trig Functions

The SIN, COS, and TAN operators use a Taylor series to calculate the function. These operators first reduce the argument to a value that is between 0 and PI/2. This reduction is accomplished by the following equation:

reduced argument = (user arg/PI - INT(user arg/PI)) * PI

The reduced argument, from the above equation, will be between 0 and PI. The reduced argument is then tested to see if it is greater than PI/2. If it is, then it is subtracted from PI to yield the final value. If it is not, then the reduced argument is the final value.

Although this method of angle reduction provides a simple and economical means of generating the appropriate arguments for a Taylor series. There is an accuracy problem associated with this technique. The accuracy problem is noticed when the user argument is large (i.e., greater than 1000). That is because significant digits, in the decimal (fraction) portion of reduced argument are lost in the (user arg/PI - INT(user arg/PI)) expression. As a general rule, try to keep the arguments for the trigonometric functions as small as possible.

4.4.3 Understanding Precedence of Operators

The hierarchy of mathematics dictates that some operations are carried out before others. If you understand the hierarchy of mathematics, it is possible to write complex expressions using a minimum amount of parenthesis. It's easy to illustrate what precedence is all about, examine the following equation:

    4+3*2 = ?

Should you add (4+3) then multiply seven by 2, or should you multiply (3*2) then add 4? Well, the hierarchy of mathematics says that multiplication has precedence over addition, so you would multiply (3*2) first then add 4. So,

    4+3*2 = 10

The rules for the hierarchy of math are simple. When an expression is scanned from left to right an operation is not performed until an operator of lower or equal precedence is encountered. In the example addition could not be performed because multiplication has higher precedence. The precedence of operators from highest to lowest in the 1771-DB is as follows:

- Operators that use parenthesis ( )
- Exponentation (**)
- Negation (-)
- Multiplication (*) and division (/)
- Addition (+) and subtraction (-)
- Relational expressions (=, <>, >, >=, <, <=)
- Logical and (.AND.)
- Logical or (.OR.)
- Logical XOR (.XOR.)

Relative to operator precedence, the rule of thumb should always be; when in doubt, use parenthesis.

4.4.4 How Relational Expressions Work

Relational expressions involve the operators =, <>, >, >=, <, and <=. These operators are typically used to "test" a condition. In the 1771-DB relational operators return a result of 65535 (OFFFFH) if the relational expression is true, and a result of 0, if the relation expression is false. The result is returned to the argument stack. Because of this, it's possible to actually display the result of a relational expression.

Examples:	PRINT 1=0	PRINT 1>0	PRINT A<>A	PRINT A=A
	0	65535	0	65535

It may seem strange to have a relational expression actually return a result, but it offers a unique benefit in that relational expressions can actually be "chained" together using the logical operators .AND., .OR., and .XOR.. This makes it possible to test a rather complex condition with ONE statement.

```
Example:  >10 IF A<B.AND.A>C.OR.A>D THEN.........
```

Additionally, the NOT([expr]) operator can be used.

```
Example:  >10 IF NOT(A>B).AND.A<C THEN..........
```

By "chaining" together relational expressions with logical operators, it is possible to test very particular conditions with one statement. When using logical operators to link together relational expressions, it is very important that the programmer pay careful attention to the precedence of operators. The logical operators were assigned lower precedence, relative to relational expressions, just to make the linking of relational expressions possible without using parenthesis.

4.5 Special Operators

4.5.1 Special Function Operators

Special function operators directly manipulate the I/O hardware and the memory addresses on the processor.

GET

The GET operator only produces a meaningful result when used in the RUN mode. It will always return a result of zero in the command mode. What GET does is read the console input device. Actually, it takes a "snapshot" of the console input device. If a character is available from the console device, the value of the character will be assigned to GET. After GET is read in the program, GET will be assigned the value of zero until another character is sent from the console device. The following example will print the decimal representation of any character sent from the console:

```
Example:    >10 A=GET
            >20 IF A<>0 THEN PRINT A
            >30 GOTO 10
            >RUN

             65     (TYPE "A" ON CONSOLE)
             49     (TYPE "1" ON CONSOLE)
             24     (TYPE "CONTROL-X" ON CONSOLE)
             50     (TYPE "2" ON CONSOLE)
```

The reason the GET operator can be read only once before it is assigned a value of zero is that this implementation guarantees that the first character entered will always be read, independent of where the GET operator is placed in the program.

TIME

Use the TIME operator to retrieve and/or assign a value to the real time clock resident in the 1771-DB. After reset, time is equal to 0. The CLOCK1 statement enables the real time clock. When the real time clock is enabled, the special function operator, TIME, will increment once every 5 milliseconds. The TIME operator uses TIMER0 and the interrupts associated with the TIMER0 on the processor. The units of time are in seconds.

When TIME is assigned a value with a LET statement (i.e. TIME = 100), only the integer portion of TIME will be changed.

```
Example:    >CLOCK1        (enable REAL TIME CLOCK)

            >CLOCK0        (disable REAL TIME CLOCK)

            >PRINT TIME    (display TIME)
             3.315

            >TIME = 0      (set TIME = 0)

            >PRINT TIME    (display TIME)
             .315          (only the integer is changed)
```

The "fraction" portion of TIME can be changed by manipulating the contents of internal memory location 71 (47H). This is accomplished by a DBY(71) statement. Note that each count in internal memory location 71 (47H) represents 5 milliseconds of TIME.
Continuing with the example;

```
>DBY(71) = 0    (fraction of TIME = 0)

>PRINT TIME
 0

>DBY(71) = 3    (fraction of TIME = 3, 15 ms)


>PRINT TIME
 1.5 E-2
```

The reason only the integer portion of TIME is changed when assigned a value is that it allows you to generate accurate time intervals. For instance, let's say you want to create an accurate 12 hour clock. There are 43200 seconds in a 12 hour period, so an ONTIME 43200, (ln num) statement is used. Now, when the TIME interrupt occurs the statement TIME = 0 is executed, but the millisecond counter is not re-assigned a value so if interrupt latency happens to exceed 5 milliseconds, the clock will still remain accurate.

4.5.2 System Control Values

The system control values determine or reveal how memory is allocated by the 1771-DB.

MTOP

After reset, the 1771-DB sizes the external memory and assigns the last valid memory address to the system control value, MTOP. The 1771-DB will not use any external RAM memory beyond the value assigned to MTOP.

```
Examples:  >PRINT MTOP   or   PHO. MTOP
           14335              37FFH
```

LEN

The system control value, LEN, tells you how many bytes of memory the current selected program occupies. Obviously, LEN cannot be assigned a value, it can only be read. A NULL program (i.e. no program) will return a LEN of 1. The 1 represents the end of program file character.

Note: Unlike some BASICS, the 1771-DB does not require any "dummy" arguments for the system control values.

4.6 Data Transfer Support Routines

The 1771-DB BASIC module communicates to the host programmable controller via the block transfer mechanism. The host controller sends variable length blocks of data to the 1771-DB. A maximum of 64 word in and 64 words out can be transferred per scan. In turn, the 1771-DB responds with the requested block length of data. All 64 words are entirely user defined and interpreted. The 1771-DB has an auxiliary processor which is dedicated to detailed servicing of the block transfers with the host programmable controller and these support routines provide the communications link to this second processor with the BASIC processor.

The 1771-DB auxiliary processor maintains a block transfer write buffer that has the values of the last transfer written by the host. The data is transferred to the BASIC processors block transfer write buffer by calls to the routines that wait on block transfer write or explicit requests to update the block transfer write buffer. This buffer is frozen and will not change when the wait on block transfer call is completed. This double buffering and freezing is required to ensure the data does not change during processing by the 1771-DB module. The block transfer write buffer remains unchanged and may be accessed repeatedly by any of the data access routines provided.

The 1771-DB also maintains a block transfer read buffer that is the value of the next block to be read by the host. This buffer is initialized by the BASIC program using these routines and is actually transferred to the auxiliary processor (for subsequent transfer to the host) when block transfer read buffer transmit routine or the wait on block transfer read routine are called. The user should complete the building of the read buffer before initiating its transfer.

The auxiliary processor receives commands from the host that allow it to halt the BASIC processor and to thus disable the 1771-DB.

4.6.1 Update Block Transfer Read Buffer (timed) CALL 2

This routine transfers the block transfer read buffer to the auxiliary processor for use in the next block transfer read request from the host. If no data transfer occurs within 2 seconds the routine returns to BASIC without transfering data. This routine has no input arguments and one output argument-the status of the transfer. A non-zero returned means that no transfer occurred and the CALL timed out. A zero returned means a successful transfer.

Ex:

```
10 CALL 2
20 POP X
30 IF X <> 0 PRINT "TRANSFER UNSUCCESSFUL"
```

4.6.2 Update Block Transfer Write Buffer (timed) CALL 3

This routine transfers the block transfer write buffer of the auxiliary processor to the BASIC block transfer write buffer. If no data transfer occurs within 2 seconds the routine returns to BASIC with no new data. This routine has no input arguments and one output argument- the status of the transfer. A non-zero returned means that no transfer occurred and the CALL timed out. A zero returned means a successful transfer.

Ex:
```
10 CALL 3
20 POP X
30 IF X <> 0 PRINT "TRANSFER UNSUCCESSFUL"
```

4.6.3 Set Block Transfer Write Length CALL 4

This routine sets the number of words (1-64) to transfer between the 1771-DB and the PLC. The PLC program block transfer length must match the value set here. Only one argument is input, the number of words to BTW, and none returned.

Ex:
```
10 PUSH 10:
20 CALL 4
```

4.6.4 Set Block Transfer Read Length CALL 5

This routine sets the number of words (1-64) to transfer between the 1771-DB and the PLC. The PLC program block transfer length must match the value set here. Only one argument is input, the number of words to BTR, and none returned.

Ex:
```
10 PUSH 10
20 CALL 5
```

4.6.5 Update Block Transfer Write Buffer CALL 6

This routine transfers the block transfer write buffer of the auxiliary processor to the BASIC block transfer write buffer. This routine will wait until a block transfer occurs.

4.6.6 Update Block Transfer Read Buffer CALL 7

This routine transfers the block transfer read buffer to the auxiliary processor for use in the next block transfer read request from the host. This routine will wait until a block transfer occurs.

4.6.7 Disable Interrupts CALL 8

This routine disables system interrupts. It is mandatory for PROM programming. The wall clock cannot be accessed and the peripheral port is disabled.

4.6.8 Enable Interrupts CALL 9

This routine enables system interrupts. It is mandatory for PROM programming. The wall clock is accessible and the peripheral port is enabled.

INPUT CALL CONVERSION ROUTINES

All of the input call conversion routines require the same sequence of commands. These are:

PUSH - the word position of the PLC data transfer file to be converted (1-64)

CALL - the appropriate input conversion

POP - the input argument into a variable

The data format of the output arguments from each of the input conversion routines is described in more detail in section 6.

4.6.9 3-Digit Signed, Fixed Decimal BCD to Internal F.P. xXXX CALL 10

The input argument is the number (1 to 64) of the word in the write data transfer buffer that is to be converted from 3-digit BCD to internal format. The output argument is the converted value in internal floating point format.

4.6.10 16-Bit Binary to Internal F.P. CALL 11

The input argument is the number (1 to 64) of the word in the write data transfer buffer to be converted from 16-bit binary to internal format. The output argument is the converted value in internal floating point format. There are no sign or error bits decoded, just a one for one conversion of the binary data.

4.6.11 4-Digit Signed Octal to Internal F.P. xXXXX CALL 12

The input argument is the number (1 to 64) of the word in the PLC data transfer buffer to be converted from 4-digit signed octal to internal format. This 12-bit format has a maximum value of ±7777 octal. The output argument is the converted value in internal floating point format.

4.6.12 6-Digit, Signed, Fixed Decimal BCD to Internal F.P. xXXXXXX CALL 13

The input argument is the number (1-64) of the first word (6-digit BCD is sent to the BASIC module in two PLC words) of the write data transfer buffer to be converted from 6-digit, signed, fixed decimal BCD to internal format. The maximum values allowed are ±999999. The output argument is the converted value in internal floating point format.

4.6.13 4-Digit BCD to Internal F.P. XXXX CALL 14

The input argument is the number (1-64) of the word in the write data transfer buffer to be converted from 4-digit BCD to internal format. The maximum value allowed is 0-9999. The output argument is the converted value in internal floating point format.

Sample input conversions:

```
20    PUSH  3     :REM CONVERT 3rd WORD OF PLC DATA
30    CALL  10    :REM DO 3 DIGIT BCD TO F.P. CONVERSION
40    POP   W     :REM GET CONVERTED VALUE - STORE IN VARIABLE W


20    PUSH  9     :REM CONVERT 9th WORD OF PLC DATA
30    CALL  13    :REM DO 6 DIGIT BCD TO F.P. CONVERSION
40    POP   L(1) :REM GET CONVERTED VALUE - STORE IN ARRAY L(1)
```

OUTPUT CALL CONVERSION ROUTINES

All of the output call conversion routines require the same sequence of commands. These are:

PUSH - The value to be converted
PUSH - the word position of the PLC data transfer file to be converted (1-64)
CALL - the appropriate output conversion

The data format of the converted value of each of the output conversion routines is described in more detail in chapter 6.

4.6.14 Internal FP to 3-Digit, Signed, Fixed Decimal BCD $xXXX. CALL 20

This routine has two input arguments and no output arguments. The first argument is the variable with a value in the range of -999 to +999 that will be converted to a signed 3-digit binary coded decimal format for use by the PLC. The second input argument is the number of the word to receive the value in the read block transfer buffer.

EX.

```
20 PUSH W  : REM DATA TO BE CONVERTED
30 PUSH 6  : REM WORD LOCATION TO GET DATA
40 CALL 20
```

4.6.15 Internal FP to 16-Bit Binary CALL 21

This routine takes a value between 0 and 65535 and converts it to its binary representative and stores this in the read block transfer buffer in one word. Two arguments are PUSHed, none POPed. The first value PUSHed is the data or variable. This is followed by the number of the word to receive the value in the read block transfer buffer.

EX.

```
50 PUSH T : REM THE VALUE TO BE CONVERTED TO 16 BINARY
60 PUSH 3 : REM WORD 3 IN THE BTR BUFFER GETS THE VALUE T
70 CALL 21 : REM DO THE CONVERSION
```

4.6.16 Internal FP to 4-Digit, Signed Octal ±xXXXX CALL 22

This routine converts a value from internal format to a four digit signed octal value. Two arguments are PUSHed, none POPed. The first value PUSHed is the data ($\pm 7777_8$) or variable. This is followed by the number of the word to receive the value in the read block transfer buffer.

```
EX.     50 PUSH H : REM THE VALUE TO BE CONVERTED TO 4-DIGIT SIGNED OCTAL
        60 PUSH 3 : REM WORD 3 IN THE BTR BUFFER GETS THE VALUE H
        70 CALL 22 : REM DO THE CONVERSION
```

4.6.17 Internal FP to 6-Digit, Signed, Fixed Decimal BCD ±xXXXXXX. CALL 23

This routine converts an internal 6-digit, signed, integer to a 2 word format and places the converted value in the read block transfer buffer. Two arguments are pushed, none poped. The first value pushed is the data or variable. This is followed by the number of the word to receive the value in the read block transfer buffer.

```
EX.     20 PUSH 654321. OR 20 PUSH B(I)
        30 PUSH 3          30 PUSH 3
        40 CALL 23         40 CALL 23
```

4.6.18 Internal FP to 3.3-digit, Signed, Fixed Decimal BCD ±xXXX.XXX CALL 26

This routine converts a variable in internal format to a signed, 6-digit, fixed decimal point number stored in 2 words in the read block transfer buffer. Two arguments are PUSHed, non POPed. The first value PUSHed is the data (±999.999) or variable. This is followed by the number of the word to receive the value in the read block transfer buffer.

```
EX.     50 PUSH S    :REM THE VALUE TO BE CONVERTED TO 3.3-DIGIT FIXED POINT
        60 PUSH 3    :REM WORD 3 IN THE BTR BUFFER GETS THE VALUE S
        70 CALL 26   :REM DO THE CONVERSION
```

4.6.19 Internal FP to 4-digit BCD ±XXXX CALL 27

This routine converts a value in internal floating point format to a 4-digit, unsigned BCD value and places it in the read block transfer buffer. Two arguments are PUSHed, none POPed. The first value PUSHed is the data (0-9999) or variable. This is followed by the number of the word to receive the value in the read block transfer buffer.

```
EX.     20 PUSH B    :REM THE VALUE TO BE CONVERTED TO 4-DIGIT BCD
        30 PUSH 7    :REM WORD 7 IN THE BTR BUFFER GETS THE VALUE B
        40 CALL 27   :REM DO THE CONVERSION
```

4.7 Peripheral Port Support

The multipurpose peripheral port is a vital part of the 1771-DB and it is supported in a variety of ways. The user may exchange data with an external device using a user written protocol that meets the needs of the application at hand. Further, this port is used by the LIST# statement of the language to list programs. Finally, this port is used to load/save programs via the 1771-SA/SB data recoders. In order for these functions to operate properly, parameter setup routines are also provided.

In order to make best use of the port, especially in the communications to another device mode, a 256 character buffer is provided on both input and output. This means that up to 256 characters may be received without error from a device before the user must service the data. Often an entire response to a command can be stored in the input buffer transparently to the user; this is especially useful when the data arrives in a high speed burst that is much too fast for a BASIC program to handle character by character.
Similarly, a 256 character output buffer is provided that allows an entire message to be assembled with the user's program free to perform other operations while the message is being sent character by character by the device driver.

The communications port is driven by two different parts of the 1771-DB module for different functions and the two groups of functions cannot be active at the same time. The first use of this port is provided by the LIST#, PRINT#, PHO.#, and PH1.# statements. These are used to list the program, output variables, strings, or control characters. Entirely separate are the routines described here.

4.7.1 Peripheral Port Support - Parameter Set — CALL 30

This routine sets up the parameters for the peripheral port. The parameters to be set are the number of bits/word, parity enable or disable, even or odd parity, number of stop bits, software handshaking, hardware handshaking. The five values that are PUSHed on the stack before executing the CALL are in the following order:

Parameter	Selections
Number of bits/word	5,6,7,8
Parity Enable	0=None 2=Even 1=Odd
Number of Stop Bits	1=1 Stop Bits, 2=2 Stop Bits, 3=1.5 Stop Bits
Software Handshaking	0=None, 1=XON-XOF,
Hardware Handshaking	0=Disable DCD 1=Enable DCD

EX.

```
100 PUSH 8  : REM 8 BITS/WORD
120 PUSH 1  : REM ODD PARITY
140 PUSH 2  : REM 2 STOP BITS
160 PUSH 0  : REM NO SOFTWARE HANDSHAKING
180 PUSH 1  : REM ENABLE DCD
200 CALL 30 : REM SET UP PERIPHERAL PORT
```

4.7.2 Peripheral Port Support - Display Peripheral Port Parameters CALL 31

This routine displays the current peripheral port configuration on the terminal. No argument is PUSHed or POPed.

Enter CALL 31 [return] from the command mode.

EX. CALL 31 [RETURN]

```
DCD ON
2 STOP BITS
ODD PARITY
8 BITS/CHAR
```

4.7.3 Save Program to Data Recorder CALL 32

This routine will save the current RAM program onto a 1770-SA/SB recorder. The program in RAM is not modified.

To use this routine, insert the cassette/cartridge into the recorder and rewind it. After the re-winding has stopped, enter CALL 32. The terminal will respond with "POSITION TAPE AND PRESS WRITE/RECORD ON TAPE".

Shortly after pressing record, the tape will begin to move and an initial asterisk(*) will be displayed on the terminal indicating that the save has begun. As each line of the program is sent to the recorder another asterisk will be displayed on the terminal. During this time the "DATA IN" LED on the data recorder and the XMIT LED on the front of the 1771-DB will illuminate. When the last line of program is sent to the recorder, a final asterisk will be displayed on the terminal indicating that the save operation is complete. Finally, tape motion will cease and the BASIC prompt (>) will be displayed.

Note: If there is no program in RAM then no data will be written to the recorder.

4.7.4 Verify Program with Data Recorder CALL 33

This routine is used to verify the current RAM program with a previously stored program on the data recorder.

To use this routine, the cassette/recorder should be inserted into the recorder and re-wound. Enter CALL 33 and the terminal will respond with "POSITION TAPE AND PRESS READ FROM TAPE". Shortly after pressing PLAY, tape movement will begin and an initial asterisk will be displayed on the terminal indicating that the verification proccess has begun. As each line of the program is verified, another asterisk will be displayed. During this time the "DATA OUT" LED on the recorder and the "RECV" LED on the 1771-DB will illuminate. When the last line of program has been verified a final asterisk will be displayed on the terminal followed by the "VERIFICATION COMPLETE" message and the BASIC (>) prompt. If any differences between programs are encountered, the routine will display a "ERROR--VERIFICATION FAILURE" message and the BASIC (>) prompt will immediately be displayed.

4.7.5 Load Program from Data Recorder CALL 34

This routine will load a program stored on a 1770-SA/SB recorder into user RAM, destroying the previous contents of RAM.

To use this routine, enter CALL 34 from the terminal. The terminal will respond with "POSITION TAPE AND PRESS READ FROM TAPE". After pressing play, tape movement will begin and the routine will now search for the beginning of the next program. When the program is found an asterisk will be printed for each line read from the recorder. During this time the "DATA OUT" LED of the recorder and the "RECV" LED on the front of the 1771-DB will illuminate. When the last line of program is sent by the recorder, a final asterisk will be displayed on the terminal indicating that the load operation is complete. Finally, tape motion will cease and the BASIC prompt (>) will be displayed.

Note: This routine will load the first program encountered and not differentiate between labelled programs. (See Section 4.7.8 for explanation of labelled programs.)

4.7.6 Get Numeric Input Character from Peripheral Port CALL 35

This routine will get the current character in the input buffer and return it as its output argument. If there is no character there, then output argument will be a 0 (null). If there was a character there, the output argument will be the ASCII representation of that character. There is no input argument for this routine.

Example:

```
10  CALL 35
20  POP X
30  IF X = 0 THEN GOTO 10
40  POINT CHR(X)
```

4.7.7 Save Labeled Program to Data Recorder (1770-SB only) CALL 38

This routine performs identically to the CALL 32 routine except the program is assigned an ID number (0-255) before storing it on tape. This enables the user to search for a specific program on a tape that contains many different programs. In order to accomplish this, PUSH the ID number, then enter CALL 38. From this point on operation is identical to the CALL 32 routine. (See section 4.7.3 for explanation) This routine has one input argument an no output arguments. If no ID number is pushed prior to CALL 38, an error will occur and the routine will abort without saving.

Note: An unlabeled save (CALL 32) has an ID of 0.

4.7.8 Load Labeled Program from Data Recorder (1770-SB only) CALL 39

This routine performs identically to the CALL 34 routine except that labeled programs are differentiated according to an ID number (0-255) that was assigned to the stored program via a CALL 38. To use this routine, PUSH the ID number, then enter CALL 39. From this point on, operation is identical to the CALL 34 routine. (See section 4.7.4 for explanation) This routine has one input argument an no output arguments. If no ID number is pushed prior to CALL 38, an error will occur and the routine will abort without saving.

Note: If the tape is positioned in the middle of the program or a labled load encounters a program with a wrong label, the tape will stop motion at the end of the encountered program. To continue search, press the load button twice.

Note: An unlabeled save (CALL 32) has an ID of 0.

Note: Maximum baud rate = 2400 bps for all data recorder CALL routines.The UART is set up in these routines and cannot be set in BASIC. The parameters of the UART are: 2 stop bits, no parity, CHAR = 8 bit ASCII, and hardware handshaking as described in the 1771-SB/SA manual.

4.8 Wall Clock Support Calls

The battery backed wall time clock provides year, month, day of month, hours, minutes, and seconds. The clock supports 24 hour military time format only. The support routines allow the setting of the clock and retrieval of clock values in numeric form.

The wall clock support routines use the argument stack to pass data between the BASIC program and the routines. Data is passed in both directions and consists of the actual clock data.

The wall clock or time of day clock is totally separate from the real time clock also provided on the 1771-DB. The real time clock is accessed by CLOCK1, CLOCK0, ONTIME, and other statements, and has a resolution of 5 milliseconds. It should be used for all short time interval measurements as it has greater resolution and results in more accurate timing. The two clocks are not synchronized and comparison of times from the two may not always give exactly the expected answer. Also, the real time clock is not battery backed.

4.8.1 Setting the Wall Clock Time (Hour, Minute, Second) CALL 40

This routine is used to set the wall clock time functions: hours, minutes, and seconds. Follow this order: hours (0-23), minutes(0-59), seconds(0-59), then call this routine.

EX.

The time is 13:35 (1:35 pm) exactly on a 24 hour clock. The wall clock must be programmed in 24 hour military format.

```
10 H=13:  M=35: S=00    :REM HOURS=13; MINUTES=35; SECONDS=00
20 PUSH H,M,S           :REM PUSH HOURS, MINUTES, SECONDS
30 CALL 40              :REM CALL THE ROUTINE TO SET THE WALL CLOCK TIME
```

4.8.2 Setting the Wall Clock Date (Day, Month, Year) CALL 41

You use this routine to set the wall clock functions: (day, month, year)

Three values are PUSHed, none POPed. Follow this order: day of the month, month,year, then call this routine.

EX.

The date is the 16th day of June, 1985

```
10 D=16: M=06: Y=85    :REM DAY OF MONTH = 16, MONTH = 6, YEAR =85
20 PUSH D,M,Y          :REM PUSH DAY OF MONTH, MONTH, YEAR
30 CALL 41             :REM CALL THE ROUTINE TO SET THE WALL CLOCK DATE
```

4.8.3 Date Retrieve Numeric (Day, Month, Year)   CALL 44

The current date is returned on the argument stack as three numbers. There is no input argument to this routine and three variables are returned. The date is POPed in day, month, and year order.

```
10 REM DATE RETRIEVE - NUMERIC EXAMPLE
20 CALL 44    : REM INVOKE THE UTILITY ROUTINE
30 POP D,M1,Y   : REM GET THE DATA FROM THE STACK
40 PRINT "CURRENT DATE IS", Y,M1,D
500 END

RUN

CURRENT DATE IS 84 12 25

READY
```

4.8.4 Time Retrieve Numeric   CALL 46

The time of day is available in numeric form by executing a CALL 46 and POPing the three variables off of the argument stack upon return. There are no input arguments. The time is POPed in hour, minute, and second order.

```
10 REM TIME IN VARIABLES EXAMPLE        : REM GET THE WALL CLOCK TIME
20 CALL 46
30 POP H,M,S
40 PRINT "CURRENT TIME IS", H,M,S
50 END

RUN

CURRENT TIME IS 13 44 54

READY
```

4.9 Description of String Operators

4.9.1 What Are Strings?

A string is a character or a group of characters that are stored in memory. Usually, the characters stored in a string make up a word or a sentence. Strings are handy because they allow the programmer to deal with words instead of numbers. This is useful because it allows one to write "friendly" programs, where individuals can be referred to by their names instead of a number.

The 1771-DB contains one dimensioned string variable, $([expr]). The dimension of the string variable (the [expr] value) range from 0 to 254. This means that 255 different strings can be defined and manipulated in the 1771-DB. Initially, no memory is allocated for strings. Memory must be allocated by the STRING [expr], [expr] STATEMENT. The details of this statement are covered in sections 4.3.27 and 4.9.

In the 1771-DB, strings can be defined in two ways: with the LET statement and with the INPUT statement.

Example:

```
>10 STRING 100,20
>20 $(1)="THIS IS A STRING, "
>30 INPUT "WHAT'S YOUR NAME? - ",$(2)
>40 PRINT $(1),$(2)
>50 END
>RUN

WHAT'S YOUR NAME? - FRED

THIS IS A STRING, FRED
```

Strings can also be assigned to each other with a LET statement.

Example: LET $(2)=$(1)

Would assign the string value in $(1) to the STRING $(2).

4.9.2 The ASC Operator

In the 1771-DB, two operators manipulate STRINGS. These operators are ASC( ) and CHR( ). The string operators available in the 1771-DB are as follows:

ASC( )

The ASC( ) operator returns the integer value of the ASCII character placed in the parenthesis.

```
Example:        >PRINT ASC(A)
                 65
```

The decimal representation for the ASCII character "A" is 65. In addition, individual characters in a predefined ASCII string can be evaluated with the ASC( ) operator.

Example:

```
>5  STRING 1000,40
>10 $(1)="THIS IS A STRING"
>20 PRINT $(1)
>30 PRINT ASC($(1),1)
>40 END
>RUN

 THIS IS A STRING
 84
```

When the ASC( ) operator is used in the manner shown above, the $([expr]) denotes what string is being accessed and the expression after the comma "picks out" an individual character in the string. In the above example, the first character in the string was picked out and 84 is the decimal representation for the ASCII character "T".

Example:

```
>5  STRING 1000, 40
>10 $(1)="ABCDEFGHIJKL"
>20 FOR X=1 TO 12
>30 PRINT ASC($(1),X),
>40 NEXT X
>50 END
>RUN

 65   66   67   68   69   70   71   72   73   74   75   76
```

The numbers printed in the previous example are the values that represent the ASCII characters A,B,C,...L.

Additionally, the ASC( ) operator can be used to change individual characters in a defined string.

Example:

```
>5  STRING 1000, 40
>10 $(1)="ABCDEFGHIJKL"
>20 PRINT $(1)
>30 ASC($(1),1)=75   REM: DECIMAL EQUIVALENT OF K
>40 PRINT $(1)
>50 ASC($(1),2)=ASC($(1),3)
>60 PRINT $(1)
>RUN

ABCDEFGHIJKL
KBCDEFGHIJKL
KCCDEFGHIJKL
```

In general, the ASC( ) operator lets the programmer manipulate individual characters in a string. A simple program can determine if two strings are identical.

Example:

```
>5  STRING 1000,40
>10 $(1)="SECRET" : REM SECRET IS THE PASSWORD

>20 INPUT "WHAT'S THE PASSWORD - ",$(2)
>30 FOR I=1 TO 6
>40 IF ASC($(1),I)=ASC($(2),I) THEN NEXT I ELSE 70
>50 PRINT "YOU GUESSED IT!"
>60 END
>70 PRINT "WRONG, TRY AGAIN" : GOTO 20
>RUN

WHAT'S THE PASSWORD - SECURE
WRONG, TRY AGAIN
WHAT'S THE PASSWORD - SECRET
YOU GUESSED IT
```

4.9.3 The CHR Operator

CHR( )

The CHR( ) operator is the converse of the ASC( ) operator. It converts a numeric expression to an ASCII character.

Example:

```
>PRINT CHR(65)
A
```

Like the ASC( ) operator, the CHR( ) operator can also "pick out" individual characters in a defined ASCII string.

Example:

```
>5 STRING 1000,40
>10 $(1)="The 1771-DB"
>20 FOR I=1 TO 11 : PRINT CHR($(1),I), : NEXT I

>30 PRINT : FOR I=11 TO 1 STEP -1
>40 PRINT CHR($(1),I), : NEXT I
>RUN

The 1771-DB
BD-1771 ehT
```

In the above example, the expressions contained within the parenthesis, following the CHR operator have the same meaning as the expressions in the ASC( ) operator.

Unlike the ASC( ) operator, the CHR( ) operator CANNOT be assigned a value. A statement such as CHR($(1),1) = H, is INVALID and will generate a BAD SYNTAX ERROR. Use the ASC( ) operator to change a value in a string, or use the string support call routine--replace string in a string. See section 4.9.4 for explainations of string support calls.

The CHR( ) function cannot be used in the expression portion of an IF-THEN statement like the ASC( ) function can.

4.9.4 String Support Calls

The intrinsic BASIC in the 1771-DB is enhanced by adding several routines for string manipulation that facilitate programming.

Strings in 1771-DB BASIC are declared by the STRING statement which must be executed before any strings are accessed or any of these string routines are called. The STRING statement has two arguments or numbers that follow it and they are first, the total amount of memory to allocate to string storage and second, the maximum size in characters of each string. Since strings are terminated by a carriage return character, each string is given an extra byte of storage for its carriage return terminator. Thus, the number of strings allowed can be determined by taking the first number and dividing it by one plus the second number. Note that the strings must be consecutively used starting with string 1 through the allowed number of strings and that all strings are allocated the maximum number of characters regardless of the actual number used.

All of these routines use or modify strings as part of their operation. The mechanism for passing a string to the support routine is to PUSH its number or subscript onto the stack. The support routine can then find the string in the 1771-DB's argument stack.

Many of these routines also require the length of a string as an input. This number must normally be inclusively between zero and the second number used in the last STRING statement which specifies the maximum size of a string. However, in all cases, if a string length argument is given of minus one (-1), it will be interpreted to be the maximum allowable string length.

It is important to note that since the carriage return character is the string terminator, it cannot be used within a string as one of its characters. Further, since all strings are initialized to carriage returns, the first character that has not been changed will be the last character of the string, even if characters later in the string have been defined. However, if the most significant bit is set in a carriage return character (use 141 instead of 13 as the decimal value of the carriage return character), the 1771-DB will not recognize it as the string terminator and will pass it to the output device. Most, but not all, devices use a seven bit ASCII code and will ignore this top bit and treat the 141 as a normal carriage return.

The following three programs allow you to determine how much string space to allocate knowing two of three variables associated with the strings: number of characters in the longest string, the number of string variables, and the amount of memory to allocate for strings.

Example 1

```
>LIST
10   REM STRING ALLOCATION COMPUTATION KNOWING:
20   REM 1) # CHARACTERS IN LONGEST STRING 2) # OF STRING VARIABLES
30   PRINT : PRINT
40   INPUT "HOW MANY CHARACTERS IN YOUR LONGEST STRING"C
50   INPUT "HOW MANY STRING VARIABLES WILL YOU NEED"V
60   PRINT
70   N=(((C+1)*V)+1) : REM COMPUTE THE # OF BYTES OF MEMORY NEEDED
80   PRINT : PRINT
90   PRINT "YOU NEED TO ALLOCATE",N,"BYTES OF MEMORY FOR",V,"VARIABLES
100  PRINT "CONTAINING",C,"CHARACTERS EACH"
110  PRINT : PRINT
120  PRINT"             STRING",N",",C
130  END


>RUN

HOW MANY CHARACTERS IN YOUR LONGEST STRING
?21
HOW MANY VARIABLES WILL YOU NEED
?15

YOU NEED TO ALLOCATE 331 BYTES OF MEMORY FOR 15 VARIABLES
CONTAINING 21 CHARACTERS EACH

        STRING 331 , 21
READY
>
```

EXAMPLE 2

```
>LIST
10   REM STRING ALLOCATION COMPUTATION KNOWING:
20   REM 1) # CHARACERS IN LONGEST STRING 2) AMOUNT OF STRING MEMORY
30   PRINT : PRINT
40   INPUT "HOW MANY CHARACTERS IN YOUR LONGEST STRING"C
50   INPUT "# OF BYTES OF MEMORY CAN YOU ALLOCATE FOR STRINGS"N
60   PRINT
70   V=INT((N-1)/(C+1)) : REM COMPUTE THE # OF POSSIBLE VARIABLES
80   N=(V*(C+1))+1 : REM COMPUTE HOW MUCH MEMORY IS ACTUALLY NEEDED
90   PRINT : PRINT
100  PRINT "YOU NEED TO ALLOCATE",N,"BYTES OF MEMORY FOR",V,"VARIABLES
110  PRINT "CONTAINING",C,"CHARACTERS EACH
120  PRINT : PRINT
130  PRINT "          STRING",N,",",C
140  END

>RUN
```

```
HOW MANY CHARACTERS IN YOUR LONGEST STRING
?20
# OF BYTES OF MEMORY CAN YOU ALLOCATE FOR STRINGS
?500

YOU NEED TO ALLOCATE 485 BYTES OF MEMORY FOR 22 VARIABLES
CONTAINING 21 CHARACTERS EACH

            STRING 485 , 21
READY
>
```

Example 3

```
10    REM STRING ALLOCATION COMPUTATION KNOWING:
20    REM 1) AMOUNT OF STRING MEMORY 2) #OF STRING VARIABLES
30    PRINT : PRINT
40    INPUT "ENTER # OF BYTES OF MEMORY YOU CAN ALLOCATE FOR STRINGS"N
50    INPUT "HOW MANY STRING VARIABLES WILL YOU NEED"V
60    PRINT
70    C=INT(((N-1)/V)-1) : REM COMPUTE THE # OF CHARACTERS/STRING
80    N=(V*(C+1))+1 : REM COMPUTE THE # OF BYTES OF MEMORY NEEDED
90    PRINT : PRINT
100   PRINT "YOU NEED TO ALLOCATE",N,"BYTES OF MEMORY FOR",V,"VARIABLES"
110   PRINT "CONTAINING","C,CHARACTERS EACH"
120   PRINT : PRINT
130   PRINT "        STRING",N,",",C
140   END


>RUN

ENTER # OF BYTES OF MEMORY YOU CAN ALLOCATE FOR STRINGS
500
HOW MANY STRING VARIABLES WILL YOU NEED
?15

YOU NEED TO ALLOCATE 496 BYTES OF MEMORY FOR 15 VARIABLES
CONTAINING 32 CHARACTERS EACH

                    STRING 496 , 32

READY
>
```

4.9.4.1 STRING REPEAT (CALL 60)

This routine allows a character to be repeated a number of times and placed into a string. This is a useful feature in designing output formats. First PUSH the number of times to repeat the character, then PUSH the number of the string containing the character to be repeated. No arguments are POPed. You cannot repeat more characters than the string's maximum length.

```
10 REM STRING REPEAT EXAMPLE PROGRAM
20 STRING 1000,50
30 $(1)="*"
40 PUSH 40  :REM THE NUMBER OF TIMES TO REPEAT CHARACTER
50 PUSH 1   :REM WHICH STRING CONTAINS CHARACTER
60 CALL 60
70 PRINT $(1)
80 END
RUN
****************************************
READY
```

4.9.4.2 STRING APPEND (CONCATENATION) (CALL 61)

This routine supports appending one string to the end of another string. The CALL expects two string arguments. The first is the string number of the string to be appended and the second is the string number of the base string. If the resulting string is longer than the maximum string length, the extra characters will be lost. There are no output arguments. This is essentially a string concatenation assignment. (Ex. $(1)=$(1)+$(2)).

```
10  STRING 200,20
20  $(1)="How are "
30  $(2)="you?"
40  PRINT "BEFORE:",
50  PRINT "$1=",$(1),"      $2=",$(2)
60  PUSH 2                  :REM STRING NUMBER OF STRING TO BE
              APPENDED
70  PUSH 1                  :REM BASE STRING NUMBER
80  CALL 61                 :REM INVOKE STRING CONCATENATION
                            ROUTINE
90  PRINT "AFTER: ",
100 PRINT "$1=",$(1),"      $2=",$(2)
110 END

RUN

BEFORE:   $1=How are       $2=you?
AFTER:    $1=How are you?  $2=you?

READY
```

4.9.4.3 Find a String in a String (CALL 64)

This routine will find a string within another string. It locates the first occurrence (position) of this string to be found. This call expects two input arguments. The first is the string to be found, the second is the string to be searched for a match. One return argument is required. If the number is not zero then a match was located at the position indicated by the value of the return argument. This routine is similar to the BASIC INSTR$(findstr$,str$). (Ex. L=INSTR$($(1),$(2))

```
10  REM SAMPLE FIND STRING IN STRING ROUTINE
20  STRING 1000,20
30  $(1)="456"
40  $(2)="12345678"
50  PUSH 1          :REM STRING NUMBER OF STRING TO BE FOUND
60  PUSH 2 :REM BASE STRING NUMBER
70  CALL 64 :REM GET LOCATION OF FIRST CHARACTER
80  POP L
90  IF L=0 THEN PRINT "NOT FOUND"
100 IF L>0 THEN PRINT "FOUND AT LOCATION ",L
110 END

RUN

FOUND AT LOCATION 4

READY
```

4.9.4.4 Replace a String in a String (CALL 65)

This routine will replace a string within another string. Three arguments are expected. The first argument is the string number of the string which will replace the string identified by the second argument string number. The third argument is the base string's string number. There are no return arguments.

```
10  REM Sample of replace string in string
20  STRING 1000,20
30  $(0)="RED-LINES"
40  $(1)="RED"
50  $(2)="BLUE"
60  PRINT "BEFORE:  $0=",$(0)
70  PUSH 2                 :REM STRING NUMBER OF THE STRING TO REPLACE WITH
80  PUSH 1                 :REM STRING NUMBER OF THE STRING TO BE REPLACED
90  PUSH 0   :REM BASE STRING NUMBER
100 CALL 65                :REM INVOKE REPLACE STRING IN STRING ROUTINE
110 PRINT "AFTER:  $0=",$(0)
120 END

RUN


BEFORE:  $0=RED-LINES
AFTER:  $0=BLUE-LINES
READY
```

4.9.4.5 Insert String in a String (CALL 66)

This routine will insert a string within another string. The call expects three arguments. The first argument is the position at which to begin the insert. The second argument is the string number of the string to be inserted into the base string. The third argument is the string number of the base string. This routine has no return arguments. It is similar to the MID$ in some BASICs.

```
10  REM SAMPLE ROUTINE TO INSERT A STRING IN A STRING
20  STRING 500,15
30  $(0)="1234590"
40  $(1)="67890  PRINT "BEFORE: $0=",$(0)
60  PUSH 6                   :REM POSITION TO START THE INSERT
70  PUSH 1                   :REM STRING NUMBER OF THE STRING TO BE INSERTED
80  PUSH 0                   :REM BASE STRING NUMBER
90  CALL 66                  :REM INVOKE INSERT A STRING IN A STRING
91  REM                      :REM ROUTINE
100 PRINT "AFTER: $0=",$(0)
110 END

RUN

BEFORE:  $0=1234590
AFTER:  $0=123456789090
READY
```

4.9.4.6 Delete String from a String (CALL 67)

This routine will delete a string from within another string. The call expects two arguments. The first argument is the base string number. The second is the string number of the string to be deleted from the base string. This routine has no return arguments.

Note: This routine will only delete the first occurrence of the string to be deleted.

```
10  REM ROUTINE TO DELETE A STRING IN A STRING
20  STRING 200,14
30  $(1)="123456789012"
40  $(2)="12"
50  PRINT "BEFORE:  $1=",$(1)
60  PUSH 1                   :REM BASE STRING NUMBER
70  PUSH 2                   :REM STRING NUMBER OF THE STRING TO BE DELETED
80  CALL 67                  :REM INVOKE STRING DELETE ROUTINE
90  PRINT "AFTER:  $1=",$(1)
100 END

RUN

BEFORE:  $1=123456789012
AFTER:   $1=3456789012

READY
```

4.9.4.7 Determine Length of a String (CALL 68)

This routine will determine the length of a string. One input argument is expected. This is the string number upon which this routine acts. One output argument is required. It is the actual number of non-carriage return (CR) characters in this string. This is similar to the BASIC command LEN(str$). (Ex. L=LEN($1))

```
10 REM Sample of string length
20 STRING 100,10
30 $(1)="1234567"
40 PUSH 1                    :REM BASE STRING
50 CALL 68                   :REM INVOKE STRING LENGTH ROUTINE
60 POP L                     :REM GET LENGTH OF BASE STRING
70 PRINT "THE LENGTH OF ",$(1)," IS ",L
80 END

RUN

THE LENGTH OF 1234567 IS 7

READY
```

4.10 Memory Support Calls

4.10.1 ROM to RAM Program Transfer CALL 70

This routine will shift program execution from a running ROM program to the beginning of the RAM program. Variables are shared. No arguments PUSHed or POPed.

IMPORTANT: The first line of the RAM program will not be executed. We recommend that you make it a remark.

Example:

```
ROM #5

10 REM SAMPLE ROM PROG FOR CALL 70
20 PRINT "NOW EXECUTING ROM #5"
30 CALL 70   :REM GO EXECUTE RAM
40 END

RAM

10 REM SAMPLE RAM PROGRAM FOR CALL 70
20 PRINT "NOW EXECUTING RAM"
30 END

>RUN

NOW EXECUTING ROM #5
NOW EXECUTING RAM
```

4.10.2 ROM/RAM to ROM Program Transfer CALL 71

This routine will transfer from a running ROM or RAM program to the beginning of any available ROM program. One argument is PUSHed - which ROM program. None are POPed. An invalid program error will be displayed and the command mode will be entered if the ROM # does not exist.

IMPORTANT: The first line of the ROM program will not be executed. We recommend that you make it a remark.

Example:

```
10 REM THIS ROUTINE WILL CALL AND EXECUTE A ROM ROUTINE
20 INPUT "ENTER ROM ROUTINE TO EXECUTE ", N
30 PUSH N
40 CALL 71
50       END
```

```
>RUN
```

```
ENTER ROM ROUTINE TO EXECUTE 4
```

The user is now executing ROM 4 if it exists. If the ROM routine requested did not exist the result would be:

```
PROGRAM NOT FOUND.
READY
>
```

4.10.3 RAM/ROM Return CALL 72

This routine will allow you to return to the ROM or RAM routine that called this ROM or RAM routine. Execution will begin on the line following the line that CALLed the routine. No arguments are PUSHed or POPed. This routine will only work one layer deep; you may only go back to the last CALLing program's next line.

Note: There must be a next line for the program to return to, otherwise unpredictable events could occur which may destroy the contents of RAM. For this reason always be sure that at least one END statement exists following a Call 70 or 71.

Example:

ROM #1

```
10 REM SAMPLE PROG FOR CALL 72
20 PRINT "NOW EXECUTING ROM #1
30 PUSH 3
40 CALL 71  :REM EXECUTE ROM #3 THEN RETURN
50 PRINT "EXECUTING ROM #1 AGAIN"
60 END
```

ROM #3

```
10 REM THIS LINE WONT BE EXECUTED
20 PRINT "NOW EXECUTING ROM #3"
30 CALL 72
40 END
```

With ROM #1 selected:

```
RUN

NOW EXECUTING ROM #1
NOW EXECUTING ROM #3
EXECUTING ROM #1 AGAIN

READY
>
```

4.10.4 Battery-backed RAM Disable CALL 73

This disables the battery-backed RAM, allowing a purging reset.

4.10.5 Battery-backed RAM Enable CALL 74

This enables the battery-backed RAM. It is disabled on power-up, so you must enable the battery-backed RAM in order for it to work. There are no PUSHes or POPs. It will remain enabled until you execute a CALL 73 or until the battery goes dead.

Note: In addition to executing a CALL 74, you must enter RUN mode prior to powering the module down in order to enable the battery-backed RAM.

5.1 Objectives

In this chapter you will learn how to interface the 1771-DB module to your PLC. This chapter will outline the BASIC programming and ladder logic required for this interface. Example BASIC programs demonstrating the 1771-DB's capability, will also be shown. Be sure that the module has been installed and initialized as explained in sections 3.2.5 and 3.2.6.

5.2 Concept Behind BASIC Module Block Transfer with a PLC

Your BASIC module communicates with any processor that has block transfer capability. Your ladder logic program and BASIC program must work together to achieve proper communications between the module and processor.

The following description explains how the BASIC module reacts to block transfer read and write requests from the PLC. As explained in chapter 4, you must do a CALL in the 1771-DB to complete the block transfer requested in the PLC ladder logic. The balance of the 1771-DB's response to block transfer requests is transparent to the user but is provided to allow a better understanding of the BASIC modules operation.

The DB module essentially consists of two buffers . These are the BASIC side buffer and the block transfer (BT) side buffer. The two buffers on the module are linked together via the data transfer routines, CALL 2 and CALL 6 for a write or CALL 3 and CALL 7 for a read. These routines transfer data between the two buffers located in the module. Refer to figure 5.1.

The sequence of events to complete a block transfer read (BTR) with the processor is as follows: When a CALL 2 or 7 is executed, the BT processor on the module is informed that there is a data transfer pending. Following the next block transfer read (BTR) or block transfer write (BTW) attempt by the PLC, the BT processor will move the data from the BASIC read buffer to the BTR buffer. At this time the CALL 2 or 7 routine is satisfied and the BASIC program will continue. Note that the data transferred by the CALL 2 or 7 has not yet been sent to the PLC and that until a request is received from the PLC the BASIC line execution has stopped until another request is received or the CALL times out. The next BTR request from the PLC will transfer the data from the BTR buffer in the 1771-DB to the PLC. Hence, it requires two block transfers to get data to the PLC.

It should be noted that if a BTR is requested by the PLC and the BASIC read buffer was never loaded, -- such as following powerup -- then no data transfer will occur. This situation would only occur prior to the first CALL 2 or 7. Once data has been entered into the buffer, BTR's will always be allowed.

The BTW sequence is somewhat different than the BTR. Assume that a BTW has been done with the BASIC module and that valid data exists in the BTW buffer on the 1771-DB. When a CALL 3 or 6 is executed, the BT processor is informed that a data transfer is pending. Following the next BTR or BTW attempt, the BT processor will move the data from the BTW buffer to the BASIC buffer. This will satisfy the CALL 3 or 6 routine and the BASIC program will continue. Conversely, if there was no data previously written to the module, then no data transfer will occur. In this case, the module will wait until a BTW occurs before the CALL 3 or 6 routine will be satisfied and the BASIC program will

continue. Once a BTW occurs between the PLC and the BASIC module, any further BTW's with it are locked out until that data is transferred to the BASIC write buffer via a CALL 3 or 6 routine. This will prevent the PLC from writing data to the BASIC module when the module is not ready for it.

There will be instances where doing a BTW with the BASIC module has satisfied a CALL 2 or 7 read transfer and a BTR has satisfied a CALL 3 or 6 write transfer. It can be seen from the above explanations, that this situation is normal operation. This is due to the fact that data is transferred within the module at the end of a block transfer attempt. Therefore, if a block transfer read or write is attempted (it doesn't have to be completed successfully) the data transfer sequence will be executed.

5.3 Block Transfer Programming

The BASIC module is a bi-directional block transfer module. Bi-directional block transfer is the performance of both read and write operations. Although the BASIC module can perform both read and write operations, the module does not allow the enable bit of both the read and write instructions to be set at the same time. Your program must toggle request for the read and write instructions as shown in the sample program.

5.3.1 PLC-2 Family Processors

You transfer data from your module to the processor data table with a block transfer read instruction and from the processor data table with a block transfer write instruction. In addition to your ladder logic programming, you must program your BASIC module to receive and transmit data via block transfer with various CALL routines.

5.3.1.1 Sample Programming--Single Data Set

This sample program assumes that you can fit all needed data into one block transfer read file and one block transfer write file.

A sample ladder logic segment and corresponding BASIC program are shown in figures 5.2, and 5.3 and described in the following paragraphs.

Sample ladder logic action

Rungs 1 and 2

The first two rungs of the sample program toggle the requests for write block transfers (BTW) and read block transfers (BTR). The BTW instruction in rung 1 sends data from the PLC to the BASIC modules BTW buffer (refer to figure 5.1). The BTR instruction in rung 2, sends data from the BASIC modules BTR buffer to the PLC (refer to figure 5.1).

Rung 3

When a read block transfer has been successfully completed, it's done bit is set, enabling the file-to-file move instruction. The read block transfer data file is then moved into a storage data file. This prevents the PLC from receiving invalid data should a block transfer communication fault occur.

In the sample program shown in fig 5.2, we made the following assumptions for demonstration purposes. Program your block instructions to meet your system requirements.

Module location: rack 1, group 3, slot 1

First available counter address: 030

Write data file length: 10 (i.e. 10 values sent to the DB)

Read data file length: 2 (i.e. 2 values sent from the DB)

Write data file: 300-311

Read data file: 400-401

Storage file (read data): 1000-1001

Sample BASIC Program Action

Lines 10 and 20 set up the BTW and BTR lengths respectively. The value PUSHed represents the number of words in the block transfer file.

Note: The block length in the block transfer instruction must match the value PUSHed in the BASIC program. If you use the default length (00), the BASIC module will expect a 64 word block transfer.

Line 40 is the data transfer write routine. This routine will wait for a BTW to occur. Lines 50 through 59 will PUSH the BASIC write buffer position, CALL a conversion routine (in this case CALL 10: AB 3-digit signed BCD to internal floating point) and POP the converted values into variables A through J. At this point the data sent from the PLC is now ready to be manipulated and/or formatted for a report, etc. by your BASIC program. For demonstration purposes, line 100 will calculate the sum and the average of the ten numbers sent to the BASIC module. Lines 110 and 120 will load the BASIC read buffer by PUSHing the variable, PUSHing the buffer word position, and CALLing a conversion routine (in this case CALL 20: Internal floating point to AB 3-digit signed BCD). Line 130 will display the values returned on a single line of a terminal connected to the program port. Line 140 performs a data transfer read with the BT circuitry of the BASIC module. Line 150 will repeat the procedure. Control-C is required to exit this program.

5.3.1.2 Sample Programming-Multiple Data Sets

The BASIC module will be able to accommodate multiple data sets (i.e. several screens for operator interfaces). The use of a data or block ID is required to determine which PLC file was sent to the 1771-DB and where a returned file should be stored in the PLC. A sample program, set up for three sets of data, is shown in figure 5.4 and explained in the following paragraphs.

Sample Ladder Logic Action

The first two rungs of the sample program toggle the requests for write block transfers (BTW) and read block transfers (BTR). The write block transfer instruction in rung one sends data from the PLC to the BASIC modules BTW buffer. The read block transfer instruction in rung two sends data from the BASIC modules BTR buffer to the PLC (refer to figure 5.1).

Rungs 3 through 7 sequence the write data to the module. Based on the count of the CTU (rung 3) a different file is sent to the BASIC module (rungs 5 through 7). The CTU's accumulated value is incremented by the BTW done bit. The preset value of the CTU should equal the number of data sets to be sent plus one. The first word of each of these files should be a block ID that is echoed back in the first read file word (400 in our example).

Rungs 8 through 10 sequence the read data from the BASIC module into proper PLC files. The block ID is used for this purpose. In our example, the block ID is the number of the data transfer (i.e. 1 through 3). By examining word 400, we move the data to the proper file.

When you use this method, remember the following points. All the data blocks utilize the same BTR and BTW instructions. Make the block lengths equal to the largest files to be transmitted and received. While this will leave you with unused data table words, it will keep the ladder logic relatively simple. We recommend placing zero's in all unused data table words.

The BASIC program will be as shown in figure 5.5. The program operates as the single data set program does with the following additions.

Line 50 PUSH 1: CALL 10: POP S

In this line the BASIC program assigns the block ID to the variable S.

Line 100 ON S GOTO 40,200,300,400

The BASIC program examines S and goes to the appropriate subroutine for the data set sent to the BASIC module.

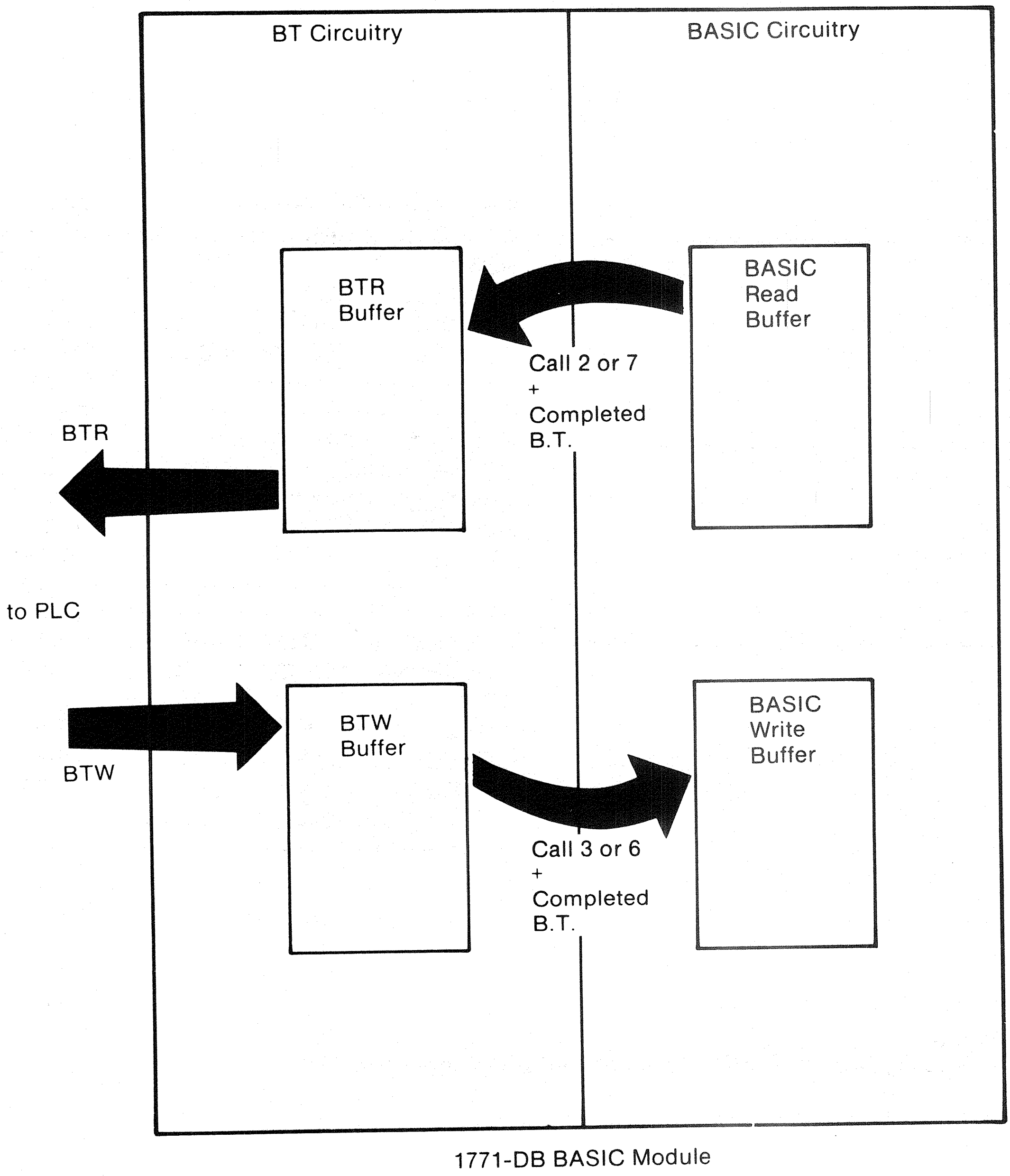


Figure 5-1
Simplified Structure of the BASIC Module W/Buffer Locations and Transfer Information

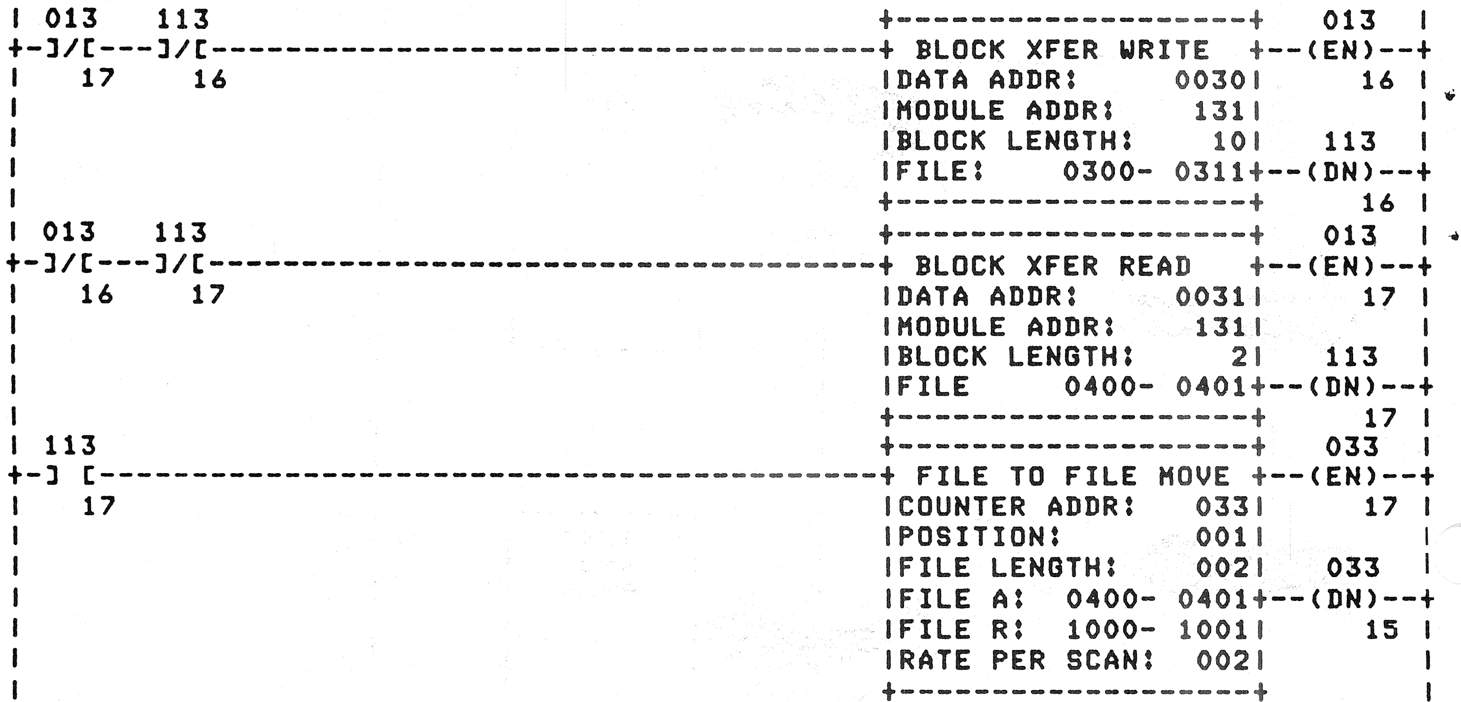


Figure 5.2
PLC-2 Family Sample Ladder Logic Structure - Single Data Set

```
10   PUSH10 : CALL 4
20   PUSH 2 : CALL 5
30   REM SET UP COMPLETE. GET THE WRITE DATA
40   CALL 6 : REM DO WRITE DATA TRANSFER
50   PUSH 1 : CALL 10 : POP A
51   PUSH 2 : CALL 10 : POP B
52   PUSH 3 : CALL 10 : POP C
53   PUSH 4 : CALL 10 : POP D
54   PUSH 5 : CALL 10 : POP E
55   PUSH 6 : CALL 10 : POP F
56   PUSH 7 : CALL 10 : POP G
57   PUSH 8 : CALL 10 : POP H
58   PUSH 9 : CALL 10 : POP I
59   PUSH 10 : CALL 10 : POP J
90   REM ALL DATA IS IN. CALCULATE RESULTS AND SEND TO THE PLC
100 T=A+B+C+D+E+F+G+H+I+J : V=T/10
110 PUSH T : PUSH 1 : CALL 20
120 PUSH V : PUSH 2 : CALL 20
130 PRINT "SENDING BACK DATA: SUM =",T,"AVE=",V,"    ",CR,
140 CALL 7
150 GOTO 40
```

Figure 5.3
Sample BASIC Program for Block Transfer - Single Data Set

LADDER DIAGRAM DUMP

```
                                         START
|                                                                   |
| 013   113                   1                              +------------------+   013  |
+-]/[---]/[--------------------------------------------------+ BLOCK XFER WRITE +--(EN)--+
|   17     16                                                |DATA ADDR:     0030|    16  |
|                                                            |MODULE ADDR:    131|        |
|                                                            |BLOCK LENGTH:    10|   113  |
|                                                            |FILE:    0300- 0311+--(DN)--+
|                                                            +------------------+    16  |
| 013   113                   2                              +------------------+   013  |
+-]/[---]/[--------------------------------------------------+ BLOCK XFER READ  +--(EN)--+
|   16     17                                                |DATA ADDR:     0031|    17  |
|                                                            |MODULE ADDR:    131|        |
|                                                            |BLOCK LENGTH:    03|   113  |
| WRT                                                        |FILE:    0400- 0411+--(DN)--+
|DN BIT                                                      +------------------+    17  |
| 113                         3                                                    0066  |
+-] [-----------------------------------------------------------------------------(CTU)-+
|   16                                                                                   |
|                                                                                PR 004  |
|                                                                                AC 003  |
|                                                                                        |
| 046                         4                                                    0046  |
+-] [-----------------------------------------------------------------------------(CTR)-+
|   15                                                                                   |
|                                                                                PR 004  |
| CTU                                                                            AC 000  |
| ALL   STORE1                                                                           |
| 0046  1300                  5                              +------------------+  0047  |
+-[G]---[=]--------------------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 001    001                                                 |COUNTER ADDR:  0047|    17  |
|                                                            |POSITION:       010|        |
|                                                            |FILE LENGTH:    010|  0047  |
|                                                            |FILE A:  1000- 1011+--(DN)--+
|                                                            |FILE R:  0300- 0311|    15  |
| CTU                                                        |RATE PER SCAN:  010|        |
| ALL  STORE2                                                +------------------+        |
| 0046  1301                  6                              +------------------+  0050  |
+-[G]---[=]--------------------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 002    002                                                 |COUNTER ADDR:  0050|    17  |
|                                                            |POSITION:       001|        |
|                                                            |FILE LENGTH:    010|  0050  |
|                                                            |FILE A:  1100- 1111+--(DN)--+
|                                                            |FILE R:  0300- 0311|    15  |
|                                                            |RATE PER SCAN:  010|        |
|                                                            +------------------+        |
```

Figure 5.4
PLC-2 Family Sample Ladder Logic Structure —Multiple Data Set

```
| CTU                                                                          |
| ALL   STORE3                                                                 |
| 0046  1302                  7                +------------------+   0051 |
+-[G]---[=]------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 003   003                                    |COUNTER ADDR:  0051|      17 |
|                                              |POSITION:       001|         |
|                                              |FILE LENGTH:    010|   0051 |
|                                              |FILE A:  1200- 1211+--(DN)--+
|                                              |FILE R:  0300- 0311|      15 |
| DYN                                          |RATE PER SCAN:  010|         |
|BLOCK STORE1                                  +------------------+         |
| 0400  1300                  8                +------------------+   0052 |
+-[G]---[=]------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 001   001                                    |COUNTER ADDR:  0052|      17 |
|                                              |POSITION:       001|         |
|                                              |FILE LENGTH:    003|   0052 |
|                                              |FILE A:  0400- 0402+--(DN)--+
|                                              |FILE R:  1400- 1402|      15 |
| DYN                                          |RATE PER SCAN:  003|         |
|BLOCK STORE2                                  +------------------+         |
| 0400  1301                  9                +------------------+   0053 |
+-[G]---[=]------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 002   002                                    |COUNTER ADDR:  0053|      17 |
|                                              |POSITION:       001|         |
|                                              |FILE LENGTH:    003|   0053 |
|                                              |FILE A:  0400- 0402+--(DN)--+
|                                              |FILE R:   1500-1502|      15 |
| DYN                                          |RATE PER SCAN:  010|         |
|BLOCK STORE3                                  +------------------+         |
| 0400  1302                  10               +------------------+   0054 |
+-[G]---[=]------------------------------------| FILE TO FILE MOVE +--(EN)--+
| 003   003                                    |COUNTER ADDR:  0054|      17 |
|                                              |POSITION:       001|         |
|                                              |FILE LENGTH:    003|   0054 |
|                                              |FILE A:  0400- 0402+--(DN)--+
|                                              |FILE R:  1600- 1602|      15 |
|                                              |RATE PER SCAN:  010|         |
|                                              +------------------+         |
```

Figure 5.4 (cont.)
PLC-2 Family Sample Ladder Logic Structure —Multiple Data Set

```
10  PUSH 10 : CALL 4 : REM SET THE BTW LENGTH
20  PUSH 3 : CALL 5 : REM SET THE BTR LENGTH
30  REM SET-UP COMPLETE. GET THE WRITE DATA
40  CALL 6 : DO BLOCK TRANSFER WRITE
50  PUSH 1 : CALL 10 : POP S : REM LOAD DATA, COVERT, ASSIGN VARIABLE
51  PUSH 2 : CALL 10 : POP A
52  PUSH 3 : CALL 10 : POP B
53  PUSH 4 : CALL 10 : POP C
54  PUSH 5 : CALL 10 : POP D
55  PUSH 6 : CALL 10 : POP E
56  PUSH 7 : CALL 10 : POP F
57  PUSH 8 : CALL 10 : POP G
58  PUSH 9 : CALL 10 : POP H
59  PUSH 10 : CALL 10 : POP I
90  REM ALL DATA IS IN. CALCULATE RESULTS AND SEND TO THE PRINTER
100 ON S GOTO, 40, 200, 300, 400 : REM GOTO A SUBROUTINE BASED ON ID VALUE
200 T=A+B+C+D+E+F+G+H+I : V=T/10 : REM DO CALCULATIONS
210 PUSH S : PUSH 1 : CALL 20 : REM CONVERT ID AND LOAD IN BTR WORD 1
220 PUSH T : PUSH 2 : CALL 30 : REM CONVERT SUM AND LOAD IN BTR WORD 2
230 PUSH V : PUSH 3 : CALL 20 : REM CONVERT AVE. AND LOAD IN BTR WORD 3
240 PRINT : 'SENDING BACK DATA : SUM =',T,'AVE=',V,'DATA ID=',S,' ',CR,
250 GOTO 600
300
 |
 |                        USER WRITTEN SUB-ROUTINE
 v
350 GOTO 600
400
 |
 |                        USER WRITTEN SUB-ROUTINE
 v
450 GOTO 600
600 CALL 7 : REM DO A BLOCK TRANSFER READ
610 GOTO 40 : REM REPEAT THE PROCEDURE
620 END
```

Figure 5.5
Sample BASIC Program for Block Transfer - Multiple Data Set

Chapter 6 Data Types

6.1 Objectives

In this chapter the data types and formats used by the 1771-DB Data Conversion CALL routines are explained. After reading this chapter you should be able to interpret and manipulate the data values generated by the 1771-DB module.

6.2 Output Data Types

You can output the following data types from the 1771-DB module:

- 16-bit binary (XXXX)
- 3-digit, signed, fixed decimal BCD (± xXXX.)
- 4-digit, unsigned, fixed decimal BCD (XXXX.)
- 4-digit, signed, octal (± xXXXX)
- 6-digit, signed, fixed decimal BCD (± xXXXXXX.)
- 3.3-digit, signed, fixed decimal BCD (± xXXX.XXX)

x = sign bit or don't care

6.2.1 16-bit Binary (4 Hex Digits)

This value requires one word of the PC data table. The data is represented by 16 straight binary bits (figure 6-1). The value ranges from 0 to 65,535. No sign, overflow or underflow bits are affected or decoded. An attempt to use a value larger than 65,535 or a negative number will yield a BAD ARGUMENT error.

6.2.2 3-digit, Signed, Fixed Decimal BCD

This value requires one word of the PC data table. The data is represented by a 3-digit binary coded decimal integer (figure 6-2). Overflow, underflow, and sign are also indicated. An underflow or overflow condition sets the appropriate bit and a value of 000 is returned. The value ranges from -999 to +999. Fractional portions of any number used with the routine will be truncated.

6.2.3 4-digit, Unsigned, Fixed Decimal BCD

This value requires one word of the PC data table. The data is represented by a 4-digit BCD integer (figure 6-3). The value ranges from 0 -9999. There is no indication of sign, underflow, or overflow. However, if a value of greater than 9999 is converted, the value reported is 0000. Fractional portions of any number used with the routine will be truncated.

6.2.4 4-digit, Signed, Octal

This value requires one word of the PC data table. The data is represented by a 4-digit octal integer (figure 6-4). The value ranges from $\pm 7777_8$ (±4095). Overflow, underflow, and sign are also indicated. If an overflow or underflow condition exists, the appropriate bit is set and the value of 0000 is reported. Fractional portions of any number used in this routine will be truncated.

6.2.5 6-digit, Signed, Fixed Decimal BCD

This value requires two words of the PC data table. The first word contains overflow, underflow, and sign data and the three most significant digits of the 6-digit BCD integer. The second word contains the lower three digits of the value (figure 6-5). The value ranges from -999999 to +999999. If an overflow or underflow condition exists, the appropriate bit is set and a value of 000000 is reported. Fractional portions of any number used with this routine will be truncated.

6.2.6 3.3-digit, Signed, Fixed Decimal BCD

This value requires two words of the PC data table. The first word contains the overflow, underflow, sign data, and the three most significant digits of the value. The second word contains the lower three digits of the value (figure 6-6). The value ranges from -999.999 to +999.999. If an overflow or underflow condition exists, a value of 000.000 is reported and the appropriate bit is set. Any digits more than 3 places to the right of the decimal point will be truncated.

6.3 Input Data Types

The 1771-DB interfaces with the PLC-2 and PLC-3 family processors. You can send the following data types to the 1771-DB module:

o 16-bit binary (4 Hex digits) (XXXX)
o 3-digit, signed, fixed decimal BCD (± xXXX.)
o 4-digit, unsigned, fixed decimal BCD (XXXX.)
o 4-digit, signed, octal (± xXXXX.)
o 6-digit, signed, fixed decimal BCD (± xXXXXXX)

These data formats are the same as those described for output data types with the exception of 3.3 digit BCD which cannot be input to the 1771-DB. Refer to sections 6.2.1 through 6.2.5.

The 1771-DB module converts the data looking at the sign bit when applicable. Refer to the programming manual for your processor for the proper data formats.

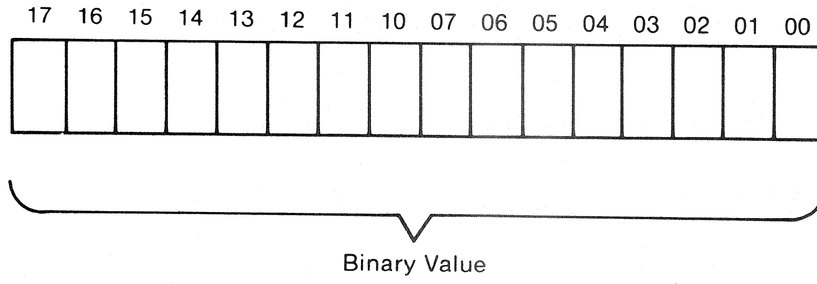


Figure 6-1
16 Bit Binary Word

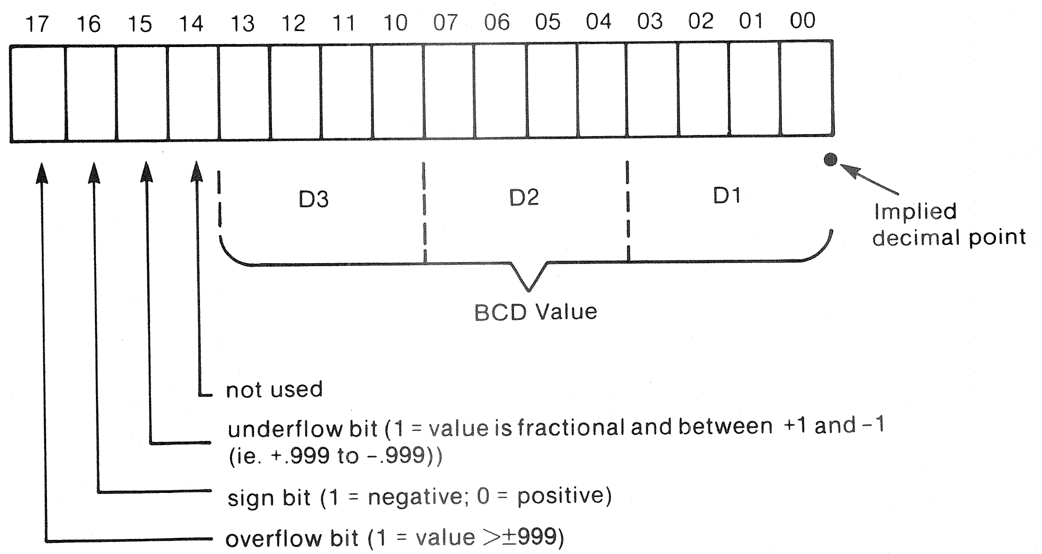


Figure 6-2
Truncated 3-Digit BCD Integer

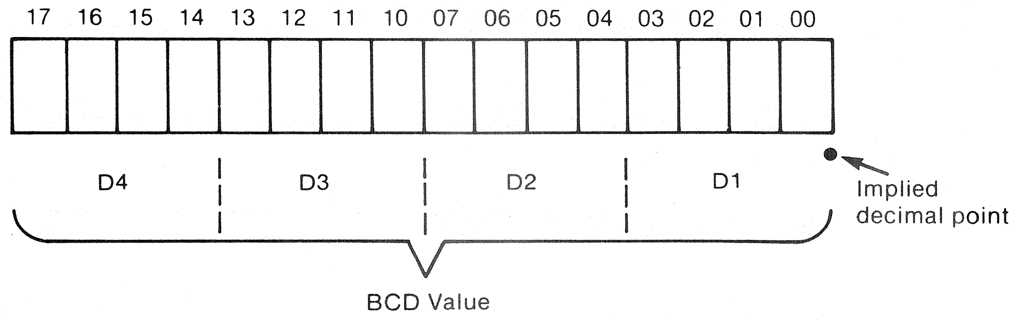


Figure 6-3
Truncated 4-Digit BCD Integer

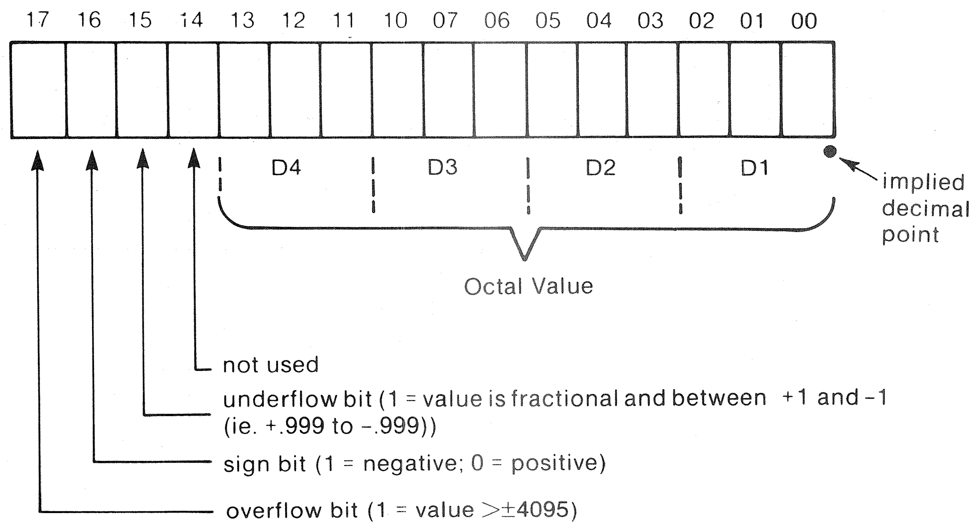


Figure 6-4
Truncated 4-Digit Octal Integer

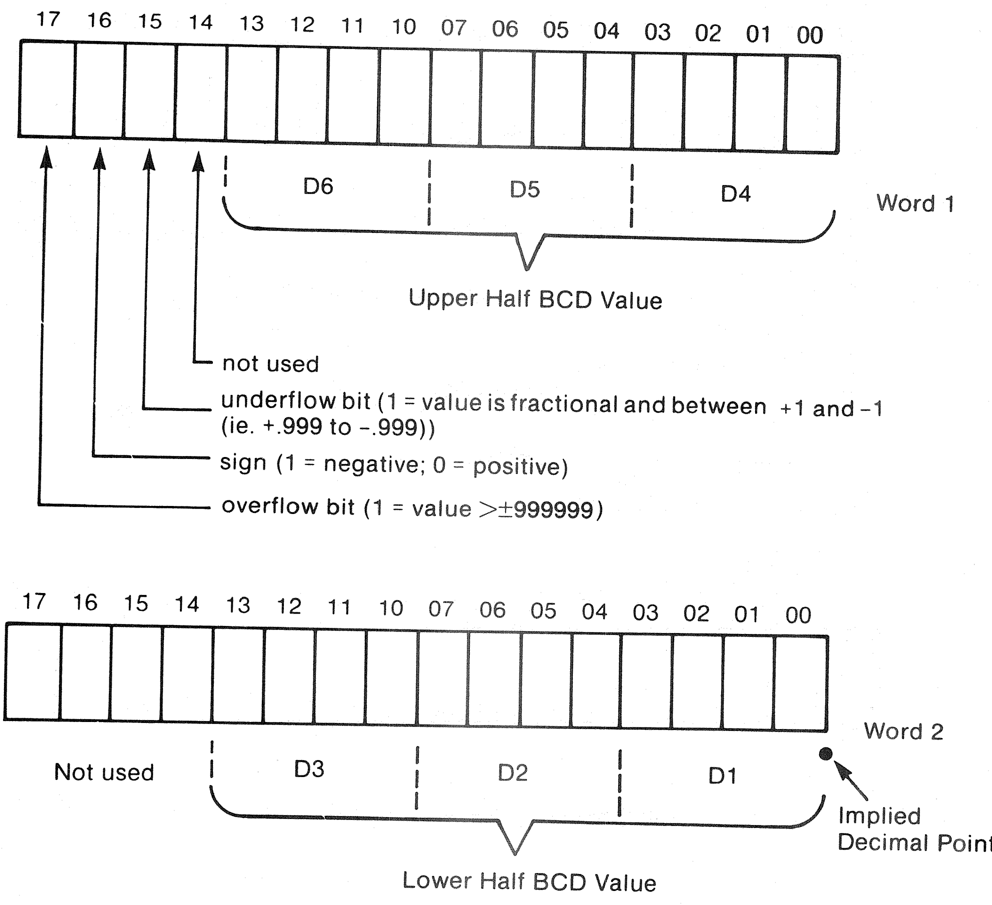


Figure 6-5
Truncated 6-Digit BCD Integer

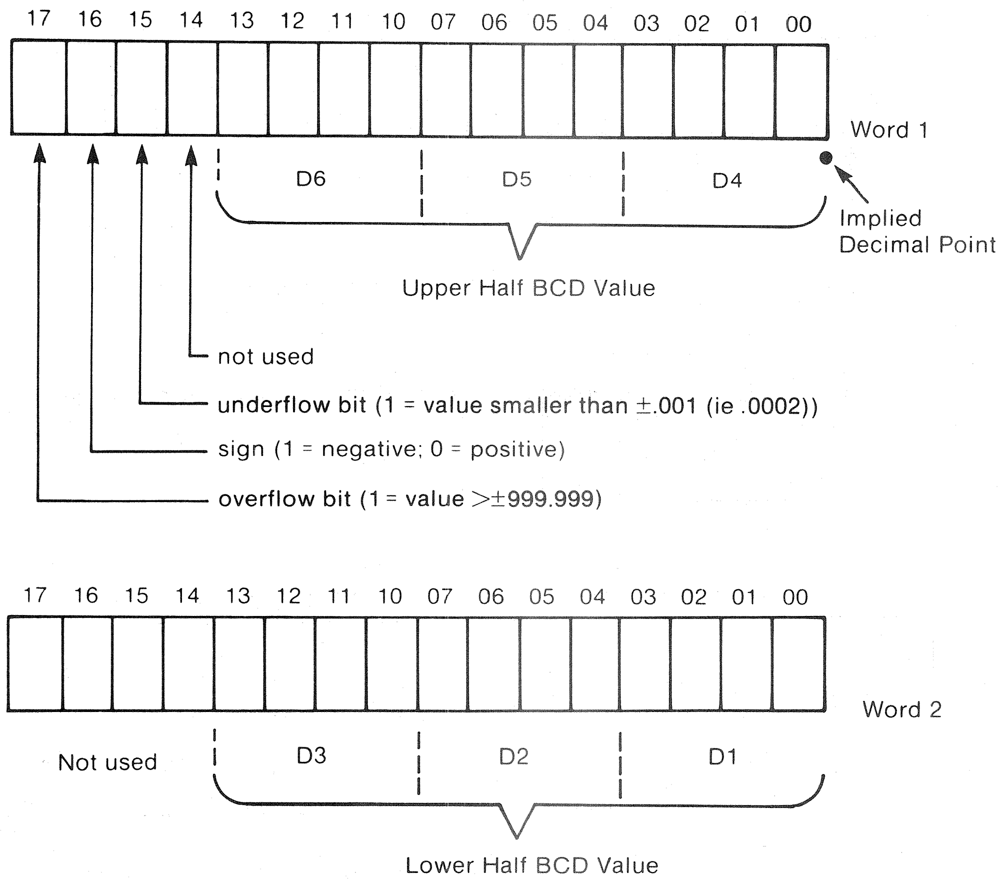


Figure 6-6
Truncated 3-Digit BCD

Chapter 7 Editing A Procedure

7.1 Objectives

After reading this chapter you should be able to make corrections or modifications to your BASIC programs using the editing features presented.

7.2 Entering the Edit Mode

To invoke the edit mode, type EDIT and the line number of the line to edit, i.e. >EDIT 10. The following features are available in the edit mode.

- o right/left cursor control
- o replace a character(s)
- o delete a character
- o retype a line

7.3 Editing Commands/Features

7.3.1 Move

The move feature provides right/left cursor control. The space bar moves the cursor one space to the right. The [RUB OUT](DELETE) key moves the cursor one space to the left.

7.3.2 Replace

The replace feature allows you to replace the character at the cursor position. Simply type the new character over the previous one.

7.3.3 Insert

The insert command is invoked by typing [CTRL]A. This command inserts text at the cursor position. You must type a second [CTRL]A to terminate the insert. Note: When the insert is used, all text to the right of the cursor will disappear until the second [CTRL]A is typed. A total line length of 72 characters still applies.

7.3.4 Delete

The delete command is invoked by typing [CTRL]D. This command deletes the character at the cursor position.

7.3.5 Retype

Retype is invoked by typing the [RETURN] key. This feature retypes the line and initializes the cursor to the first character.

7.3.6 Exits

The exit command is invoked by typing [CTRL]Q or [CTRL]C. [CTRL]Q exits the editor and replaces the old file with the edited file. [CTRL]C exits the editor with no changes made to the file.

7.3.7 Renumber

The renumber command is invoked by typing REN or REN[NUM]on the screen. Ten is the default value.This command will renumber your program, starting with the number specified or the default value, in increments of the number specified or the default value.

Note: The renumber command will update the destination of GOSUB, GOTO, and ON GOTO statements.

Example:

REN 20 will renumber a program beginning with 20 in increments of 20.

REN will renumber a program beginning with 10 in increments of 10 (the default value).

7.4 Editing A Simple Procedure

The following is an example of using the editor:

```
Screen                  Action

>LIST
10 REM
20 FOR I = 1 TO 6
30 PRINT I
40 NEXT I
READY
>
                        - Type EDIT 20 then [RETURN]
>EDIT 20

> 20 FOR I = 1 TO 6
                        - Press the space bar 18 times to get the
                          cursor on 6 (MOVE right feature)
                        - Type 9 to change 6 to 9
                          (REPLACE feature)
                        - Press [CTRL]A (INSERT command)
                        - Press space bar once, type STEP, press
                          space bar once, type 2
                        - Press [CTRL]A to exit the insert
                        - Press [CTRL]Q to exit the editor with
                          the changes accepted
READY
>
                        type LIST then [RETURN]
>LIST
10 REM
20 FOR I = 1 TO 9 STEP 2
30 PRINT I,
40 NEXT I

READY
>
```

Chapter 8 Error Messages

8.1 Objectives

In this chapter we will explain error messages, the disabling of control-C, and anomalies.

8.2 Error Messages

The 1771-DB has a relatively sophisticated ERROR processor. When BASIC is in the RUN mode the generalized form of the ERROR message is as follows:

```
ERROR: XXX - IN LINE YYY

YYY BASIC STATEMENT
-----------X
```

Where XXX is the ERROR TYPE and YYY is the line number of the program in which the error occurred. A specific example is:

```
ERROR: BAD SYNTAX - IN LINE 10

10 PRINT 34*21*
--------------X
```

The X signifies approximately where the ERROR occurred in the line number. The specific location of the X may be off by one or two characters or expressions depending on the type of error and where the error occurred in the program. If an ERROR occurs in the COMMAND MODE only the ERROR TYPE will be printed out NOT the Line number. This makes sense, because there are no line numbers in the COMMAND MODE. The ERROR TYPES are as follows:

BAD SYNTAX

A BAD SYNTAX error means that either an invalid 1771-DB command, statement, or operator was entered and BASIC cannot process the entry. The user should check and make sure that everything was typed in correctly.

BAD ARGUMENT

When the argument of an operator is not within the limits of the operator a BAD ARGUMENT error will be generated. For instance, SQR(-12) would generate a BAD ARGUMENT error because the value of the SQR operator is limited to positive numbers.

ARITH. UNDERFLOW

If the result of an arithmetic operation exceeds the lower limit of a 1771-DB floating point number, an ARITH. UNDERFLOW error will occur. The smallest floating point number in the 1771-DB is +/- 1E-127. For instance, 1E-80/1E+80 would cause an ARITH. UNDERFLOW error.

ARITH. OVERFLOW

If the result of an arithmetic operation exceeds the upper limit of a 1771-DB floating point number, an ARITH. OVERFLOW will occur. The largest floating point number in the 1771-DB is +/-.99999999E+127. For instance, 1E+70*1E+70 would cause an ARITH. OVERFLOW error.

DIVIDE BY ZERO

If you attempt to divide by zero i.e. 12/0, a DIVIDE BY ZERO error appears.

ILLEGAL DIRECT

Some statements, such as FOR-NEXT and DATA cannot be executed while the 1771-DB device is in the COMMAND mode. If you attempt to execute one of these statements the message ERROR: ILLEGAL DIRECT will be printed to the console device.

LINE TOO LONG

If you type in a line that contains more than 73 characters the message ERROR: LINE TOO LONG will be printed to the console device. The 1771-DB's input buffer can only handle up to 73 characters.

NO DATA

If a READ STATEMENT is executed and no DATA STATEMENT exists or all DATA has been read and a RESTORE instruction was not executed the message ERROR: NO DATA - IN LINE XXX will be printed to the console device.

CAN'T CONTINUE

Program execution can be halted by either typing in a control-C to the console device or by executing a STOP statement. Normally, program execution can be resumed by typing in the CONT command. However, if you edit the program after halting execution and then enter the CONT command, a CAN'T CONTINUE error will be generated. A control-C must be typed during program execution or a STOP statement must be executed before the CONT command will work.

PROGRAMMING

If an error occurs while the 1771-DB device is programming an EPROM, a PROGRAMMING error will be generated. An error encountered during programming destroys the EPROM file structure, so you cannot save any more programs on that particular EPROM once a PROGRAMMING error occurs. If the EPROM size is exceeded, the previously stored program may be partially altered. See section 4.2.13 on how to determine the amount of space that is available in RAM.

A-STACK

An A-STACK (ARGUMENT STACK) error occurs when the argument stack pointer is forced "out of bounds". This can happen if you overflow the argument stack by PUSHing too many expressions onto the stack, or by attempting to POP data off the stack when no data is present.

C-STACK

A C-STACK (CONTROL STACK) error will occur if the control stack pointer is forced "out of bounds". 158 bytes of external memory are allocated for the control stack, FOR - NEXT loops require 17 bytes of control stack DO - UNTIL, DO - WHILE, and GOSUB require 3 bytes of control stack. This means that 9 nested FOR - NEXT loops is the maximum that the 1771-DB can handle because 9 times 17 equals 153. If you attempt to use more control stack than is available in the 1771-DB, a C-STACK error will be generated. In addition, C-STACK errors will occur if a RETURN is executed before a GOSUB, a WHILE or UNTIL before a DO, or a NEXT before a FOR.

ARRAY SIZE

If an array is dimensioned by a DIM statement and then you attempt to access a variable that is outside of the dimensioned bounds, an ARRAY SIZE error will be generated.

EXAMPLE:

```
>DIM A(10)
>PRINT A(11)

ERROR: ARRAY SIZE
READY
```

MEMORY ALLOCATION

MEMORY ALLOCATION errors are generated when you attempt to access STRINGS that are "outside" the defined string limits.

8.3 - Disabling Control-C

In some applications, it may be desirable or even a requirement that program execution not accidentally be halted. Under "normal" operation the execution of any the 1771-DB program can be terminated by typing a "control-C" on the console device. However, it is possible to disable the "control-C" break function in the 1771-DB. This is accomplished by setting bit 48 (30H) to a one. Bit 48 is located in internal memory location 38 (26H). This bit may be set by executing the following statement in a 1771-DB program or from the command mode:

```
DBY(38) = DBY(38).OR.01H
```

Once this bit is set to a one, the control-C break function, for both LIST and RUN operations will be disabled.

To re-enable the control-C function, execute the following statement in a 1771-DB program or from the command mode.

```
DBY(38) = DBY(38).AND.OFEH
```

8.4 - Anomalies

Most dictionaries define an anomaly as a deviation from the normal or common order or as an irregularity. Anomalies to an extreme become "BUGS" or something that is wrong with the program. Like all programs, the 1771-DB contains some anomalies, hopefully, no bugs. The purpose of mentioning the known anomalies here is that it may save the programmer some time, should strange things happen during program execution. The known anomalies deal mainly with the way the 1771-DB compacts or tokenizes the BASIC program. The known anomalies and cautions are as follows:

1) When using the variable H after a line number, make sure you put a space between the line number and the H, or else BASIC will assume that the line number is a HEX number.

Examples:

```
>20H=10   (WRONG)         >20 H=10   (RIGHT)

>LIST                     >LIST
32    =10                 20     H=10
```

2) When using the variable I before an ELSE statement, make sure you put a space between the I and the ELSE statement, or else BASIC will assume that the IE portion of IELSE is the special function operator IE.

This error may or may not yield a BAD SYNTAX - IN LINE XXX error message, depending on the particular program it is in.

Examples:

```
>20 IF I>10 THEN PRINT IELSE 100
>LIST
20     IF I>10 THEN  PRINT IELSE 100  (WRONG)
>20 IF I>10 THEN PRINT I ELSE 100
>LIST
20     IF I>10 THEN  PRINT I ELSE 100 (RIGHT)
```

3) A Space character may not be placed inside the ASC( ) operator. In other words, a statement like PRINT ASC( ) will yield a BAD SYNTAX ERROR. Spaces may be placed in strings however, so a statement like LET $(1) = "HELLO, HOW ARE YOU" will work properly. The reason ASC( ) yields an error is because the 1771-DB eliminates all spaces when a line is processed, so ASC( ) will be stored as ASC() and the 1771-DB interprets this as an error.

Appendix A-Quick Reference Guide

Mnemonic	Page	Description	Example
ABS()	4-37	ABSOLUTE VALUE	ABS(-3)
ATN()	4-39	RETURNS ARCTANGENT OF ARGUMENT	ATN(1)
CALL	4-11	CALL APPLICATION PROGRAM	CALL 10
CLEAR	4-12	CLEARS VARIABLES, INTERRUPTS, & STRINGS	CLEAR
CLEARI	4-13	CLEAR INTERRUPTS	CLEARI
CLOCK0	4-13	DISABLE REAL TIME CLOCK	CLOCK0
CLOCK1	4-13	ENABLE REAL TIME CLOCK	CLOCK1
CONT	4-5	CONTINUE AFTER A STOP OR CONTROL-C	CONT
CONTROL C	4-7	STOP EXECUTION & RETURN TO COMMAND MODE	CTRL C
CONTROL S	4-7	INTERRUPT A LIST COMMAND	CTRL S
CONTROL Q	4-8	RESTART A LIST AFTER CONTROL S	CTRL Q
COS()	4-39	RETURNS THE COSINE OF ARGUMENT	COS(0)
DATA	4-14	DATA TO BE READ BY READ STATEMENT	DATA 100
DIM	4-15	ALLOCATE MEMORY FOR ARRAY VARIABLES	DIM A(20)
DO	4-16	SET UP LOOP FOR WHILE OR UNTIL	DO
END	4-18	TERMINATE PROGRAM EXECUTION	END
EXP()	4-38	"e" (2.7182818) TO THE X	EXP(10)
FOR-TO (STEP)	4-19	SET UP FOR NEXT LOOP	FOR A=1 TO 5
GET	4-41	READ CONSOLE	P.GET
GOSUB	4-20	EXECUTE SUBROUTINE	GOSUB 1000
GOTO	4-21	GOTO PROGRAM LINE NUMBER	GOTO 5000
IF-THEN-ELSE	4-22	CONDITIONAL TEST	IF A<B THEN A=0
INPUT	4-23	INPUT A STRING OR VARIABLE	INPUT A
INT()	4-37	INTEGER	INT(3.2)
LEN	4-43	READ THE NUMBER OF BYTES OF MEMORY IN THE CURRENT SELECTED PROGRAM	PRINT LEN
LET	4-25	ASSIGN A VARIABLE OR STRING A VALUE (LET IS OPTIONAL)	LET A=10
LIST	4-6	LIST PROGRAM TO THE CONSOLE DEVICE	LIST LIST 10-50
LIST#	4-7	LIST PROGRAM TO SERIAL PRINTER	LIST# LIST# 50
LOG()	4-38	NATURAL LOG	LOG(10)
MTOP	4-43	READ THE LAST VALID MEMORY ADDRESS	PRINT MTOP
NEW	4-7	ERASE THE PROGRAM STORED IN RAM	NEW
NEXT	4-19	TEST FOR-NEXT LOOP CONDITION	NEXT A
NOT()	4-37	ONES COMPLIMENT	NOT(0)
NULL	4-7	SET NULL COUNT AFTER CARRIAGE RETURN-LINE FEED	NULL NULL 4
ONERR	4-25	ONER OR GOTO LINE NUMBER	ONERR 1000
ON GOTO	4-22	CONDITIONAL GOTO	ON A GOTO 5,20
ON GOSUB	4-22	CONDITIONAL GOSUB	ON A GOSUB 2,6
ONTIME	4-26	GENERATE AN INTERRUPT WHEN TIME IS EQUAL TO OR GREATER THAN ONTIME ARGUMENT-LINE NUMBER IS AFTER COMMA	ONTIME 10, 1000
PHO	4-31	PRINT HEX MODE WITH ZERO SUPPRESSION	PHO.A
PHI1	4-31	PRINT HEX MODE WITH NO ZERO SUPPRESSION	PHI.A

Appendix A-Quick Reference Guide

Mnemonic	Page	Description	Example
PHO.#	4-31	PHO. TO LINE PRINTER	PHO.#A
PHI.#	4-31	PH1.# TO LINE PRINTER	PH1.#A
PI	4-38	PI - 3.1415926	PI
POP	4-32	POP ARGUMENT STACK TO VARIABLES	POP A,B,C
PRINT	4-28	PRINT VARIABLES, STRINGS OR LITERALS P, IS SHORTHAND FOR PRINT	PRINT A
PRINT#	4-30	PRINT TO SOFTWARE SERIAL PORT	PRINT# A
PRINT CR	4-28	PRINT I, CR	
PRINT SPC(5)	4-28	PRINT "A", SPC(5), "B"	A....B
PRINT TAB(#)	4-28	PRINT TAB(5), "x"	x
PRINT USING(Fx)	4-29	PRINT USING (F3), 1,2	1.00E0 2.00E0
PRINT USING(#.#)	4-29	PRINT USING(##.##), 1,2	1.00 2.00
PROG	4-9	SAVE THE CURRENT PROGRAM IN EPROM	PROG
PROG1	4-11	SAVE BAUD RATE INFORMATION IN EPROM	PROG1
PROG2	4-11	SAVE BAUD RATE INFORMATION IN EPROM AND EXECUTE PROGRAM AFTER RESET	PROG2
PUSH	4-31	PUSH EXPRESSIONS ON ARGUMENT STACK	PUSH10,A
RAM	4-8	EVOKE RAM MODE, CURRENT PROGRAM IN READ/WRITE MEMORY	RAM
READ	4-14	READ DATA IN DATA STATEMENT	READ A
REM	4-33	REMARK	REM DONE
RESTORE	4-14	RESTORE READ POINTER	RESTORE
RETI	4-34	RETURN FROM INTERRUPT	RETI
RETURN	4-20	RETURN FROM SUBROUTINE	RETURN
RND	4-37	RANDOM NUMBER	RND
ROM	4-8	EVOKE ROM MODE, CURRENT PROGRAM IN ROM/EPROM MEMORY	ROM ROM3
RUN	4-4	EXECUTE A PROGRAM	RUN
SGN	4-37	SIGN	SGN(-5)
SIN()	4-38	RETURNS THE SINE OF ARGUEMENT	SIN(3.14)
SQR()	4-37	SQUARE ROOT	SQR(100)
STOP	4-34	BREAK PROGRAM EXECUTION	STOP
STRING #,#	4-34	ALLOCATE MEMORY FOR STRINGS # BYTES TO ALLOCATE, # BYTES/STRING	STRING 50,10
TAN()	4-39	RETURNS THE TANGENT OF THE ARGUMENT	TAN(0)
TIME	4-42	RETRIEVE AND/OR ASSIGN REAL TIME CLOCK VALUE	PRINT TIME TIME=0
XFER	4-9	TRANSFER A PROGRAM FROM ROM/EPROM TO RAM	XFER
+	4-35	ADDITION	1+1
/	4-35	DIVISION	10/2
**	4-36	EXPONENTATION	2**4
*	4-36	MULTIPLICATION	4*2
-	4-36	SUBTRACTION	8-4
.AND.	4-36	LOGICAL AND	10.AND.5
.OR.	4-36	LOGICAL OR	2.OR.1
.XOR.	4-36	LOGICAL EXCLUSIVE OR	3.XOR.2

Table D.A

Decimal/Hexadecimal/Octal/ASCII Conversion Table

Column 1				Column 2				Column 3				Column 4			
DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC	DEC	HEX	OCT	ASC
00	00	000	NUL	32	20	040	SP	64	40	100	@	96	60	140	\
01	01	001	SOH	33	21	041	!	65	41	101	A	97	61	141	a
02	02	002	STX	34	22	042	"	66	42	102	B	98	62	142	b
03	03	003	ETX	35	23	043	#	67	43	103	C	99	63	143	c
04	04	004	EOT	36	24	044	$	68	44	104	D	100	64	144	d
O5	05	005	ENQ	37	25	045	%	69	45	105	E	101	65	145	e
06	06	006	ACK	38	26	046	&	70	46	106	F	102	66	146	f
07	07	007	BEL	39	27	047	'	71	47	107	G	103	67	147	g
08	08	010	BS	40	28	050	(	72	48	110	H	104	68	150	h
09	09	011	HT	41	29	051	)	73	49	111	I	105	69	151	i
10	0A	012	LF	42	2A	052	*	74	4A	112	J	106	6A	152	j
11	0B	013	VT	43	2B	053	+	75	4B	113	K	107	6B	153	k
12	0C	014	FF	44	2C	054	,	76	4C	114	L	108	6C	154	l
13	0D	015	CR	45	2D	055	-	77	4D	115	M	109	6D	155	m
14	0E	016	S0	46	2E	056	.	78	4E	116	N	110	6E	156	n
15	0F	017	SI	47	2F	057	/	79	4F	117	O	111	6F	157	o
16	10	020	DLE	48	30	060	0	80	50	120	P	112	70	160	p
17	11	021	DC1	49	31	061	1	81	51	121	Q	113	71	161	q
18	12	022	DC2	50	32	062	2	82	52	122	R	114	72	162	r
19	13	023	DC3	51	33	063	3	83	53	123	S	115	73	163	s
20	14	024	DC4	52	34	064	4	84	54	124	T	116	74	164	t
21	15	025	NAK	53	35	065	5	85	55	125	U	117	75	165	u
22	16	026	SYN	54	36	066	6	86	56	126	V	118	76	166	v
23	17	027	ETB	55	37	067	7	87	57	127	W	119	77	167	w
24	18	030	CAN	56	38	070	8	88	58	130	X	120	78	170	x
25	19	031	EM	57	39	071	9	89	59	131	Y	121	79	171	y
26	1A	032	SUB	58	3A	072	:	90	5A	132	Z	122	7A	172	z
27	1B	033	ESC	59	3B	073	;	91	5B	133	[	123	7B	173	{
28	1C	034	FS	60	3C	074	<	92	5C	134	\	124	7C	174	|
29	1D	035	GS	61	3D	075	=	93	5D	135	]	125	7D	175	}
30	1E	036	RS	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

ALLEN-BRADLEY

Industrial Computer Group — PC Division

Publication 1771-6.5.34 — September, 1985

PN 955097-59