AutoMax® Enhanced BASIC Language

Instruction Manual J-3675-6

RELIANCE ELECTRIC

The information in this user's manual is subject to change without notice.

DANGER

ONLY QUALIFIED PERSONNEL FAMILIAR WITH THE CONSTRUCTION AND OPERATION OF THIS EQUIPMENT AND THE HAZARDS INVOLVED SHOULD INSTALL, ADJUST, OPERATE, OR SERVICE THIS EQUIPMENT. READ AND UNDERSTAND THIS MANUAL AND OTHER APPLICABLE MANUALS IN THEIR ENTIRETY BEFORE PROCEEDING. FAILURE TO OBSERVE THIS PRECAUTION COULD RESULT IN SEVERE BODILY INJURY OR LOSS OF LIFE.

WARNING

PROGRAMS INSERTED INTO THE PRODUCT SHOULD BE REVIEWED BY QUALIFIED PERSONNEL WHO ARE FAMILIAR WITH THE CONSTRUCTION AND OPERATION OF THE SYSTEM AND THE POTENTIAL HAZARDS INVOLVED. FAILURE TO OBSERVE THIS PRECAUTION COULD RESULT IN BODILY INJURY OR DAMAGE TO EQUIPMENT.

WARNING

THE USER MUST PROVIDE AN EXTERNAL, HARDWIRED EMERGENCY STOP CIRCUIT OUTSIDE THE CONTROLLER CIRCUITRY. THIS CIRCUIT MUST DISABLE THE SYSTEM IN CASE OF IMPROPER OPERATION. UNCONTROLLED MACHINE OPERATION MAY RESULT IF THIS PROCEDURE IS NOT FOLLOWED. FAILURE TO OBSERVE THIS PRECAUTION COULD RESULT IN BODILY INJURY.

WARNING

ONLY QUALIFIED PERSONNEL WHO HAVE READ AND UNDERSTOOD ALL APPLICABLE AUTOMAX INSTRUCTION MANUALS AND ARE THOROUGHLY FAMILIAR WITH THE PARTICULAR APPLICATION MAY UTILIZE THE ON-LINE PROGRAMMING OPTION PROVIDED IN THE AUTOMAX PROGRAMMING SOFTWARE. FAILURE TO OBSERVE THIS PRECAUTION COULD RESULT IN BODILY INJURY OR DAMAGE TO EQUIPMENT.

Norton® is a registered trademark of Peter Norton Computing, Inc.
MODBUS® is a registered trademark of Gould, Inc.
Data Highway™ is a trademark of Rockwell International.
VT100™ is a trademark of Digital Equipment Corporation (DEC).
IBM-XT™ and AT™ are trademarks of International Business Machines.
Toshiba is a trademark of Toshiba America.
Microsoft®, Windows®, and MS-DOS® are trademarks of Microsoft.
Reliance™, AutoMax™, and AutoMate™ are registered trademarks of Rockwell International.
Shark™, ReSource™, and R-Net™ are trademarks of Reliance Electric Company or its subsidiaries.

Table of Contents

Appendices

List of Tables

1.0 INTRODUCTION

The products described in this instruction manual are manufactured by Reliance Electric Industrial Company.

The AutoMax Programming Executive software includes the software used to create and compile Enhanced BASIC programs. This instruction manual describes AutoMax Enhanced BASIC language for Version 2.0 and later AutoMax Programming Executive software.

Features that are either new or different from those in the previous version of the AutoMax Programming Executive software are so noted in the text. Appendix F lists the differences between versions of the software.

This instruction manual is organized as follows:

1.1 Compatibility with Earlier Versions

Version 2.0 of the AutoMax Programming Executive requires AutoMax Processor M/N 57C430A or 57C431; Version 3.0 and later require AutoMax Processor M/N 57C430A, 57C431, or 57C435. M/N 57C430 cannot co-exist in the same rack with M/N 57C430A, 57C431, or 57C435. Refer to Appendix E for a listing of the AutoMax Processors that are compatible with Version 2 and later of the AutoMax Programming Executive software.

The thick black bar shown at the right-hand margin of this page will be used throughout this instruction manual to signify new or revised text or figures.

1.2 Additional Information

You should be familiar with the instruction manuals which describe your system configuration. This may include, but is not limited to, the following:

- J-3618 NORTON EDITOR REFERENCE MANUAL
- J-3649 AutoMax CONFIGURATION TASK INSTRUCTION MANUAL
- J-3650 AutoMax PROCESSOR INSTRUCTION MANUAL
- J-3676 AutoMax CONTROL BLOCK LANGUAGE INSTRUCTION MANUAL
- J-3677 AutoMax LADDER LOGIC INSTRUCTION MANUAL
- J2-3018 AutoMax Remote I/O Shark Interface Instruction Manual
- J2-3093 AutoMax Ladder Language Editor
- J2-3094 AutoMax Enhanced Ladder Language
- Your ReSource AutoMax PROGRAMMING EXECUTIVE INSTRUCTION MANUAL
- Your personal computer, DOS, and Windows instruction manuals
- IEEE 518 GUIDE FOR THE INSTALLATION OF ELECTRICAL EQUIPMENT TO MINIMIZE ELECTRICAL NOISE INPUTS TO CONTROLLERS

1.3 Related Hardware and Software

The AutoMax Programming Executive software is used with the following hardware and software, which is sold separately.

1. M/N 57C430A, M/N 57C431, or M/N 57C435 AutoMax Processor.
2. IBM-compatible 80386-based personal computer running DOS Version 3.1 or later. Version 4.0 and later Executive software requires an 80486-based personal computer (or higher) running Windows 95.
3. M/N 61C127 RS-232C ReSource Interface Cable. This cable is used to connect the personal computer to the Processor module.
4. M/N 57C404A (and later) Network Communications module. This module is used to connect racks together as a network and supports communication with all racks on the network that contain 57C404A modules through a single Processor module. M/N 57C404 can be used to connect racks on a network; however, you cannot communicate over the network to the racks that contain M/N 57C404 Network modules. You must instead connect directly to the Processors in those racks.
5. M/N 57C440 Ethernet Network Interface module. This module is used to connect AutoMate Processors to TCP/IP Ethernet local area networks.

6. M/N 57C413 or 57C423 Common Memory module. This module is used when there is more than one Processor module in the rack.
7. M/N 57C492 Battery Back-Up. This unit is used when there is a M/N 57C413 Common Memory module in the rack.
8. M/N 57C384 Battery Back-Up Cable. This cable is used with the Battery Back-Up unit.
9. M/N 57C554 AutoMax Remote I/O Shark Interface Module. This module is used to connect a Shark remote rack to the AutoMax Remote I/O network.
10. B/M 57552 or B/M 57652 Universal Drive Control module. This module is used for drive control applications.
11. M/N 57C560 AutoMax PC3000 Processor/Scanner module. This module is a full-size ISA module that mounts in the personal computer.
12. M/N 57C565 AutoMax PC3000 Serial Interface module. This module is a full-size ISA module that mounts in the personal computer.
13. M/N 57C570 Industrial AutoMax PC300. This unit consists of a panel-mount, industrial grade enclosure containing an AutoMax PC3000 Processor/Scanner module, an AutoMax PC3000 Serial Interface Module, and a power supply.

2.0 PROGRAMMING FOR AutoMax SYSTEMS

In AutoMax systems, application programs, also referred to as tasks, can be written in Ladder Logic/PC language, Control Block language, and Enhanced BASIC language. Enhanced BASIC language is modeled after standard BASIC. It consists of simple statements, functions, and math notation to perform operations. Refer to J- 3676, J-3677, and J2-3094 for more information about Control Block and Ladder Logic/PC programming.

In addition to multi-processing, AutoMax systems incorporate multi-tasking. This means that each AutoMax Processor (up to four) in a rack allows real-time concurrent operation of multiple application tasks.

Multi-tasking features allow the programmer's overall control scheme to be separated into individual tasks, each written in the programming language best suited to the task. This simplifies writing, check-out, and maintenance of programs; reduces overall execution time; and provides faster execution for critical tasks.

Programming in AutoMax systems consists of configuration, or defining the hardware, system-wide variables, and application tasks in that system, as well as application programming.

2.1 Configuration

Version 3.0 and Later Systems

If you are using AutoMax Version 3.0 or later, you configure the system within the AutoMax Programming Executive. See the AutoMax Programming Executive for information about configuration if you are using V3.0 or later.

The information that follows is applicable only if you are using AutoMax Version 2.1 or earlier. If you are using AutoMax Version 3.0 or later, you can skip over the remainder of this section and continue with 2.2.

Version 2.1 and Earlier Systems

AutoMax Version 2.1 and earlier requires a configuration task in order to define the following:

1. All tasks that will reside on the Processors in a rack.
2. All variables that equate to physical I/O in the system.
3. All other variables that must be accessible to all Processors in the rack.

One configuration task is required for each rack that contains at least one Processor. The configuration task must be loaded onto the Processor(s) in the rack before any application task can be executed because it contains information about the physical organization of the entire system.

The configuration task does not actually execute or run; it serves as a central storage location for system-wide information. Note that local variables, those variables that do not need to be accessible to more than one task, do not need to be defined in the configuration task. Refer to J-3649 for more information about configuration tasks.

2.2 AutoMax Application Tasks

AutoMax Processors allow real-time concurrent operation of multiple programs, or application tasks, on the same Processor module. The tasks are executed on a priority basis and share all defined system data. Application tasks on different Processor modules in the rack are run asynchronously.

Each task operates on its own variables. The same variable names may be used in different tasks, but each variable is only recognized within the confines of its task unless it is specifically designated a COMMON variable. Changing local variable ABC% (designated LOCAL) in one task has no effect on variable ABC% in any other task.

Multi-tasking in a control application can be compared to driving a car. The programmer can think of the different functions required as separate tasks, each with its own priority.

In driving a car, the operator must monitor the speedometer, constantly adjust the pressure of his foot on the gas pedal, check the rearview mirror for other traffic, stay within the boundaries of his lane, etc., all while maintaining a true course to his destination. All of these functions have an importance or priority attached to them, with keeping the car on the road being the highest priority. Some tasks, like monitoring the gasoline gauge, require attention at infrequent intervals. Other tasks require constant monitoring and immediate action, such as avoiding obstacles on the road.

In a control application the Processor needs to be able to perform calculations necessary for executing a control scan loop, monitor an operator's console, log error messages to the console screen, etc. Of these tasks, executing the main control loop is obviously the most important, while logging error messages is the least important. Multi-tasking allows the control application to be broken down into such tasks, with their execution being dependent upon specified "events," such as an interrupt, operator input, or the expiration of a time interval.

The following table is a representation of typical tasks found in a control application and the kind of event that might trigger each.

Task	Triggering Event
Execute main control loop	Expiration of a hardware timer that indicates the interval at which to begin a new scan
Respond to external I/O input	Generation of a hardware interrupt by an input module
Read operator data	Input to an operator panel
Log information	Expiration of a software timer

Each of these tasks would be assigned a priority level (either in the specific configuration task for the rack, or in later versions of the Programming Executive software, through the configuration option). The priority determines which task should run at any particular instant. The more important the task, the higher the task priority.

2.3 Universal Drive Controller Application Tasks

Universal Drive Controller (UDC) modules can be used in an AutoMax Version 3.3 (or later) system for drive control applications. Only UDC Control Block tasks can be run on a UDC module; BASIC tasks cannot be run.

UDC Control Block tasks can use some of the statements and functions in the AutoMax Enhanced BASIC language. See Appendix D for a list of the BASIC language statements and functions that are allowed in UDC Control Block tasks.

3.0 STRUCTURE OF AN AutoMax ENHANCED BASIC PROGRAM

BASIC programs, or tasks, are created using a text editor.

Note the following naming convention. Application task names in AutoMax are limited to 8 characters. The initial character must always be a letter. Following the initial character can be letters (A-Z), underscores (_), and numbers (0-9). Spaces and other characters are not permitted. The file extension is used to identify the task. Extension .CNF identifies configuration tasks. Extension .BAS is used for BASIC tasks. AutoMax Control Block tasks use extension .BLK. UDC Control Block tasks also use extension .BLK. PC/Ladder Logic tasks have a .PC extension.

An AutoMax Enhanced BASIC program consists of a set of statements using certain language elements and syntax (rules for the form of each statement). Each line begins with a number that identifies the line as a statement and indicates the order of statement execution. Each statement starts with a word specifying the type of operation to be performed, such as PRINT, GOTO, and READ. For a BASIC program to compile correctly, all text except print list items delimited by quotation marks must be in upper case.

The following symbols have special meaning for the duration of this manual:

<CR>	=	Carriage return, sometimes marked “RETURN” or “ENTER” on keyboards. You should assume that all BASIC statements end with a <CR> unless otherwise noted. Some statements used in examples may explicitly use the <CR> notation at the end of a statement to make the example easier to understand.
_	=	Underscore character used to make variable names more readable (for example, MOTOR_SPEED, LINE_REFERENCE).

Note that the underscore is not a dash or minus character, which appears on the same key as the underscore on most standard keyboards.

3.1 Line Format

The format of a statement in a BASIC program is as follows:

```
  line      statement      statement       line
 number     keyword           body       terminator

   10          LET    SPEED%=(GAIN%+3)     <CR>
```

The line number is a label that distinguishes one line from another within a program. Consequently, each line number must be unique. The line number must be a positive integer within the range of 1 to 32767 inclusive.

Line numbers are required for the following reasons:

1. To determine the order in which to execute the program.

2. To provide a reference for conditional and unconditional transfers of control (GOTO,GOSUB, etc).

Line numbers can be consecutive numbers:

```
1  LET M%=23
2  LET Z%=11
3  LET K%=Z%+M%
4  END
```

However, writing line numbers in increments of 10 allows for inserting additional statements between existing lines:

```
10  LET M%=23
20  LET Z%=11
30  LET K%=Z%+M%
40  END
```

3.2 Multi-Statement Lines

In BASIC, one line can be either one statement or several statements, always terminated by pressing the RETURN <CR> key.

A single statement line consists of:

1. A line number from 1 to 32767	10
2. A statement keyword	PRINT
3. The body of the statement	A%+B%
4. A line terminator	<CR>

Example of a single statement line:

```
10 LET SPEED%=(GAIN%+3)/12                    <CR>
```

A multi-statement line (more than one statement on a single line) requires a backslash (\) or a colon (:) to separate each complete statement. The backslash or colon statement separator must be typed after every statement except the last one. For example, the following line contains three complete PRINT statements:

```
10 PRINT A$\PRINT B$\PRINT C$                 <CR>
```

or

```
10 PRINT A$:PRINT B$:PRINT C$                 <CR>
```

There is only one line number for a multi-statement line. You should take this into consideration if you plan to transfer control to a particular statement within a program. For instance, in the above example, you cannot execute just the statement PRINT B$ without executing PRINT A$ and PRINT C$ as well.

3.3 Multi-Line Statements

In BASIC, a statement can continue onto another line. When a statement is to be continued, the line is terminated with an ampersand (&) followed by a <CR>. After the ampersand, only spaces or tabs are allowed. Other characters will cause compiler errors. The following is an example of a multi-line statement.

```
20    LET MOTOR_REF%=MOTOR_REF%+           & <CR>
      SYSTEM_GAIN% - OLDGAIN%/2+FACTOR%   <CR>
```

The ampersand tells the compiler that the statement is continuing on the next line. <CR> without the ampersand before it signifies that the entire statement is complete. When a statement is continued on a second line, that line should begin with a tab to provide maximum readability. The statement below is confusing because the 30 looks like a line number instead of part of an equation.

```
20 LET MOTOR_REF%=MOTOR_REF%+OLDGAIN%+ & <CR>

30+GAINFACTOR%+VALUE%                    <CR>
```

4.0 VARIABLES AND CONSTANTS

All operations performed in BASIC use constants or variables. Constants are quantities with fixed value represented in numeric format. Variables are names that represent stored values or physical I/O. These values may change during program execution. BASIC always uses the current value of a common (i.e., system-wide) variable in performing calculations.

Note that Control Block and PC/Ladder Logic tasks capture (latch) the values of all common simple double integer, integer, and boolean variables at the beginning of the task scan. Strings, reals, and array variables of any type are not latched. This means that Control Block and PC/Ladder Logic tasks do not see the most current state of common simple double integer, integer, and boolean variables; instead, they see the state of these variables at the beginning of the scan. Any changes made to these variable values by Control Block or PC/Ladder Logic tasks are written to the variable locations at the end of the scan of the particular task. See section 4.1.3 for more information about common variables.

4.1 Variables

The following sections describe the use of variables in AutoMax Enhanced BASIC.

4.1.1 Simple Variables

Variable names in AutoMax tasks must meet the following conditions:

1. They must ALWAYS start with a letter or an underscore.
2. Following the letter/underscore can be letters, digits, or an underscore.
3. They must not include spaces.
4. The maximum length for any variable in BASIC or Control Block tasks is 16 characters (letters, underscore, or digits), not including the type character attached at the end (%,!,@,$). Note that PC/Ladder Logic tasks variables are limited to 14 characters (16 in V4.0 and later). This is important if variables that are used in BASIC or Control Block tasks must also be used in Ladder Logic tasks, i.e., if the variables are common.

This variable length (16) permits meaningful and understandable names. Avoid cryptic variable names.

Meaningful Variable Names	Unintelligible Variable Names
MOTOR_SPEED%	MSPD%
GAIN%	G%
CURRENT_GAIN%	CGN%
DROP_1_REFERENCE%	D1RF%

AutoMax Enhanced BASIC has variable "types" just as standard BASIC does. The variable type indicates the kind of information the variable is representing (numeric data, characters, etc.). The variable type is specified by a terminator or ending character.

BASIC uses five types of variables:

1. Single integer variables (values −32768 to +32767)
2. Double integer variables (values −2147483648 to +2147483647)
3. Real variables (values 9.2233717E+18 to −9.2233717E+18). Note that the "E+(n)" is read as an exponent in BASIC.
4. Boolean variables [values TRUE (ON) or FALSE (OFF)]
5. String variables.

4.1.1.1 Single Integer Variables

A single integer variable is a named location in which an integer value can be stored. It is called a "single" integer because it requires a single 16-bit word to represent its current value in the range +32767 to −32768 (a 16-bit signed number). It is named using the rules listed in section 4.1.1 are terminated with a percent sign (%).

If you include an integer variable in a program, its value can be an integer (no fractional part) or a real (decimal) number. If you assign a decimal number to an integer variable, the fractional part will be truncated or ignored. For example, if the statement attempts A% = 3.6574, the value 3 will be assigned to A%.

If an attempt is made to assign a value larger than the range +32767 to −32768 to a single integer variable, BASIC will log this condition into the error log and will load the largest possible single integer value into the variable. For example, if the statement attempts A% = 43987, BASIC will log this as an error and set a A% = 32767; if the statement attempts A% = −53667, Basic will log an error and A% will be set to = −32768.

The following are valid single integer variables:

```
MOTOR_SPEED%
FREQUENCY%
ROLL_WIDTH%
VOLTAGE_REF%
```

The following are invalid single integer variables and the reasons that they are invalid:

GAIN *(Variable not terminated with %)*
GAIN%2 *(Variable not terminated with%)*
55SPEED% *(Variable starts with a digit rather than a letter or an underscore.)*

All internal integer calculations are in double precision, or 32 bits.

4.1.1.2 Double Integer Variables (Long Integers)

A double integer variable is a named location in which an integer value can be stored. It is called a “double” integer because it requires two 16-bit words, or 32 bits, to represent its value in the range +2147483647 to −2147483648 (a 32-bit signed number). It is named using the rules listed in section 4.1.1 and terminated with an exclamation point (!). If you include an integer variable in a program, its value can be an integer (no fractional part) or a real (decimal) number. If you assign a decimal number to a double integer variable, the fractional part will be truncated or ignored. For example, if the statement attempts A! = 3.6574, the value 3 will be assigned to A!.

If an attempt is made to assign a value larger than the range −214783648 to +214783647 to a double integer variable, BASIC will log this condition into the error log and will load the largest possible double integer value into the variable. For example, if the statement attempts A! = +2157483647, BASIC will log this as an error and set a A! = +2147483647; if the statement attempts A! = −214783649, Basic will log an error and A! will be set to = −214783648.

The following are valid double integer variables:

```
RESOLVER_ADDRESS!
LARGE_COUNTER!
FOREIGN_CARD_ADR!
```

All internal integer calculations are in double precision, or 32 bits.

4.1.1.3 Real Variables

A real variable is a named location in which a decimal value can be stored. It is named using the rules listed in section 4.1.1. Unlike the other variable data types, a real variable has no terminating character, such as % or !.

A real variable can have the following values:

9.2233717×10^{18} > positive value > $5.4210107 \times 10^{-20}$

$-9.2233717 \times 10^{18}$ > negative value > $-2.7105054 \times 10^{-20}$

Note: When entering real variable values in your program, use scientific notation. See section 4.2.3 for more information on real constant formats.

The following are examples of valid real variables:

```
ROLL_RATIO
GAIN_ADJUST
WINDUP_FRACTION
```

Only eight digits of significance are used when entering a real number, thus 9.4481365 and 9.4481365200178 would be treated the same way. The 200178 at the end of the second number would be ignored. Real or decimal numbers require more time to process while BASIC is running due to the increased accuracy and additional internal calculations required.

It is legal to assign an integer to a real variable (REAL=45). However, if the integer is greater than 2^{24} (16777216), the real value into which it is converted will be imprecise because of the format in which the real numbers are manipulated (24-bit mantissa).

4.1.1.4 Boolean Variables

A boolean variable is a named location which represents a TRUE/FALSE or ON/OFF value. It is named using the rules listed in section 4.1.1 and terminated with an “at” symbol (@).

The following are valid boolean variable names:

```
REEL_EMPTY@
OVER_TEMP@
TURRET_ENGAGED@
```

The following are invalid boolean variable names and the reasons that they are invalid:

SYSTEM_READY (*Variable not terminated with @*)
WEB_FULL@@ (*Two @s back-to-back are illegal*)
4TH_READY@ (*Variable starts with a digit rather than a letter or an underscore*)

As with integer and real variables, boolean variables form expressions. With boolean variables you use the boolean operators NOT, AND, OR, and XOR and boolean constants TRUE, FALSE, ON, and OFF in forming these expressions:

```
LINE_DOWN@ = NOT(POWER@  AND RUN@)
RUN _REQUEST@ = TRUE
SECTION_ POWER@ = FALSE
CRT _REFRESH@ = OFF
IF RUN@ OR (STOPPED@ AND FAULT@) THEN 1350
```

Refer to section 5.3 for more information about boolean expressions.

4.1.1.5 String Variables

String variables are used to store any alphanumeric sequence of printable characters, including spaces, tabs, and special characters. The terminating character is $.

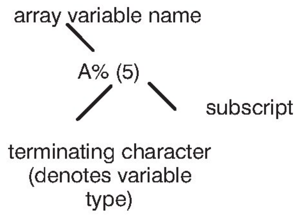
The sequence in a string variable cannot include a line terminator (<CR>). When defined, the sequence must be enclosed either in single or double quotes. If one type of quotes is used in the sequence itself, the other type must be used to enclose the sequence.

Version 1.0 Executive software allowed a fixed maximum length of 31 characters for string variables. Version 2.0 and later allows string variables of variable length, from 1 to 255 characters. To specify the maximum size of a string variable, add a colon and a number (1-255) immediately after the $ character. For example, defining A$:50 as a local variable in an application task will reserve space for 50 characters. Note that if no length is specified, the default length is 31.

4.1.2 Subscripted Variables (Arrays)

Array variables are used to store a collection of data all of the same data type. Arrays are permitted for all data types. Arrays are limited to four dimensions, or subscripts. The number of elements in each dimension is limited to 65535. This size is further limited by available memory. The term array is used to denote the entire collection of data. Each item in the array is known as an element.

Array variables are specified by adding a subscript(s) after the variable name, which includes the appropriate terminating character to denote the type of data stored in the array. The terminating character is followed by a left parenthesis (or bracket), the subscript(s), and a right parenthesis (or bracket). Multiple subscripts are separated by commas. Note that subscripts can be integer constants as well as arithmetic expressions that result in integer values.



An array with one dimension, i.e., one subscript, is said to be one-dimensional. An array with two subscripts is said to be two-dimensional, etc. The first element in each dimension of the array is always element 0. Therefore, the total number of elements in each dimension of the array is always one more than the largest subscript. For example, array A%(10) is a one-dimensional array containing eleven integer values.

Example 1 - One-dimensional array

	0	1	2	3	4	5
A%	185	2	53	79	99	122

value of A

Example 2 – Two-dimensional array

B% (6, 3)

		0	1	2	3	4	5	6
B%	0	185	2	53	79	99	122	40
	1	70	36	46	31	34	85	6
	2	77	73	21	365	476	51	47
	3	18	23	53	342	39	224	107

In the case of string arrays, Version 1.0 Executive software always allocated the maximum amount of memory for each element in the array, regardless of whether the string stored in that element was of the maximum length, 31 characters. Version 2.0 (and later) Executive software allows the programmer to specify the maximum size of elements in the array, from 1 to 255 characters.

To specify the maximum size of string variables in an array, add a colon and a number (1-255) immediately after the $ character when declaring the variable in an application task or defining it during configuration. For example, defining A$:10(20) as a local variable in an application task allocates space for 21 string values of 10 characters each. Note that if no length is specified in the initial array reference, the default maximum is 31.

To define an array that will be common, i.e., accessible to all tasks in the rack, you need to first define the variable. If you are using AutoMax Version 2.1 or earlier, this is done with a MEMDEF or NVMEMDEF statement in the configuration task for the rack. If you are using AutoMax Version 3.0 or later, common variables are defined within the Programming Executive. For example, ARRAY1@(10) will allocate space for 11 boolean variables. Then, in an application task for the rack, you declare the array a COMMON variable as follows:
COMMON ARRAY1@(10). Each element of the array that will be used in the task can be defined with LET statements as follows:
LET ARRAY1@(0) = TRUE (boolean values can only be TRUE/FALSE or ON/OFF). Other application tasks in the rack can access the value in variable ARRAY1@(0) simply by declaring it a COMMON variable.

4.1.3 Variable Control Types

The control type of a variable refers to the way the variable is declared or defined in the configuration and application tasks. There are two control variable types in AutoMax systems, local and common.

1. **Local**

   Local variables are variables that are not defined in the rack configuration and are therefore accessible only to the application task in which they are defined. BASIC and Control Block tasks must define the variables with a BASIC LOCAL statement. For Ladder Logic/PC tasks, the editor prompts for whether the variable is local or common when the task is being created.

   In BASIC and Control Block tasks, local variables can be defined as tunable. Tunables are variables whose value can be tuned, i.e., changed within limits, by the operator through the On-Line menu of the Executive software. The value of tunable variables can also be changed by application tasks by using the BASIC languange WRITE_TUNE function. BASIC and Control Block tasks must define tunable variables with a variation of the BASIC LOCAL statement that includes the tuning parameters. Ladder Logic/PC tasks cannot use tunable variables.

   The value of local variables at the time of initial tasks installation is always 0. The effect of a Stop All or a power failure on variable values in the rack depends on the variable type. Local tunable variable values in both AutoMax and UDC application tasks is always retained. Local variable values are retained for AutoMax tasks, but not for UDC tasks.

   AutoMax Processors will retain the last values of all local variables. UDC modules will retain the variable values for the following: parameter configuration data, UDC test switch information, and D/A setup configuration. The variable values of the following input data will also be retained: feedback registers, UDC-PMI communication status registers, and UDC task error log information. UDC modules will NOT retain local variable values and data found in the following registers, which are considered outputs: command registers, application registers, the ISCR (interrupt status and control register), scans per interrupt register, and scans per interrupt counter register. See the AutoMax Programming Executive for more information on the STOP ALL and system re-initialization conditions.

2. **Common**

Common variables are variables that are defined in the rack configuration and are therefore accessible to all application tasks in the rack. There are two types of common variables, those that refer to memory locations, and those that refer to actual physical I/O locations. The two types are defined differently in the configuration for the rack.

Common memory variables can be of any data type. They may be read to or written from. Common I/O variables are long integer, integer, or boolean variables that represent actual physical I/O locations. Common I/O variables that represent inputs may be read but not written to. I/O variables that represent outputs may be read or written to.

All BASIC and Control Block tasks that need to access common variables can do so by using the BASIC statement COMMON (or GLOBAL). For Ladder Logic/PC tasks, the editor prompts for whether the variable is local or common when the task is being created. At least one task in the rack should also initialize common memory variables, i.e., assign values to them, if they need to be at a known state other than 0.

The value of common variables at the time of initial task installation depends upon whether the variable references memory or physical I/O locations. Common memory variables are always 0 at task installation. Common I/O variables that represent outputs are always 0. Common I/O variables that represent inputs are always at their actual state.

After a STOP ALL condition or a power failure followed by a system-restart, common memory variables that are defined as volatile memory statements in the configuration are 0. Common memory variables that are defined as non-volatile memory in the configuration retain their last value. Common variables that represent I/O locations are at 0 for outputs and at their actual state for inputs. Note that the UDC dual port memory is treated like I/O variables. See the AutoMax Programming Executive for more information on the STOP ALL and system-restart conditions.

4.1.4 Pre-defined Common Memory Variables

The following common memory variables are pre-defined for every rack. However, they do not appear on the form for common memory variables. You must enter these variable names on the form if you want to use these variables in application tasks.

AUTORUNSTATUS@ -	True when AUTO RUN is enabled for the rack; false if AUTO RUN is not enabled
FORCINGSTATUS@ -	True when a variable is forced in the rack; false when no variables are forced in the rack
BATTERYSTATUS0@ -	True when the on-board battery of the Processor module or Common Memory module in slot 0 is OK
BATTERYSTATUS1@ -	" " " " " " " " " " " " 1 " " "
BATTERYSTATUS2@ -	" " " " " " " " " " " " 2 " " "
BATTERYSTATUS3@ -	" " " " " " " " " " " " 3 " " "
BATTERYSTATUS4@ -	" " " " " " " " " " " " 4 " " "

4.2 Constants

A constant, also known as a literal, is a fixed value that is not associated with a variable name. Listed below are the five types of constants that can be used in AutoMax, along with their size limitations.

1. Single and double integer constants (whole numbers)
2. Hexadecimal constants (whole numbers in base 16 or "hex" format)
3. Real (decimal) constants
4. String constants (alphanumeric and/or special characters)
5. Boolean constants

4.2.1 Integer Constants

An integer constant is a whole number with no fractional part. For example, the following numbers are all integer constants:

29	−8
3432	1
12345	205

The following are not integer constants:

1.6	.08
754.2	5.2041E+06
$34^1/_2$	95.3

Recall that BASIC integer constants must fall in the range −32768 to +32767 when used as single 16-bit integer variables (ending in %), or in the range −2147483648 to +2147483647 when used as double (32-bit) integer variables (ending with !). If you specify a number outside the appropriate range, BASIC prints a compiler error message telling you to replace the number with one within the proper limits.

4.2.2 Hexadecimal Constants

A hexadecimal constant also specifies an integer value in base 16 or "hex" (hexadecimal) format. A hexadecimal number has three parts:

0NNNNNNNNH

where:

1. A leading zero (0) is required if the first digit of the hexadecimal number is an alphabetical character (A through F) so that BASIC can distinguish it as a number and not a variable name. A leading zero may also be used in front of a numeric character in the hexadecimal number just as in a normal integer constant (0987H = 987H)
2. The eight Ns represent the 8 hexadecimal (hex) digits in the range 0 through F.
3. The trailing character "H" indicates that the number is hexadecimal and is always required.

The following are correct hexadecimal numbers:

098FCE2H	0FFEEC1H
0BEEFC2H	400B3C2H
99987H	

The following are invalid hexadecimal numbers and the reasons that they are invalid:

FEC002H *(Does not start with zero in front of the alpha hex character F.)*
9800BE *Does not end with H.)*
3FFFFE342H *(Larger than maximum double integer.)*

BASIC hexadecimal numbers must fall in the range from zero to 0FFFFFFFFH. Hexadecimal constants are stored by BASIC exactly as they are specified, with leading zeros filling in any of the eight hex digits not specified for a double word or 32-bit format. This means the numbers must be specified as 2's complement signed numbers.

For example, BASIC will load the hex constant 0F371H as 0000F371H. It will not sign-extend the number to 0FFFFF371H. If the number 0FFFFF37H is desired, the entire 8 hex digits must be specified. If you specify a number outside the appropriate range (0FFFFFFFFH), the compiler will print an error on the screen.

4.2.3 Real Constants

A real constant is a number with a decimal point. For large numbers, use scientific notation in the following general format: a sign, digits, a decimal point, digits, an "E," a sign, and digits. Take, for example, the following real constant example:

−1234.5678E+11

The "E" is a real constant which means "times ten to the" followed by the "power" or exponent. In the real constant 34.99876E+07, the "E" means times ten to the 7th. There must always be a number in front of the "E" when the "E" is used (E+13 by itself is illegal.). Only 8 digits of significance are used to store the number. The total number of digits on the left and right side of the decimal point must be less than or equal to 8.

As with the real variable, the real constant can have a value in the range:

$9.2233717 \times 10^{18} >$ positive value $> 5.4210107 \times 10^{-20}$
$-9.2233717 \times 10^{18} <$ negative value $< -2.7105054 \times 10^{-20}$

All of the values listed below represent the number one hundred twenty:

120	0000120.00	+.120E+03
120.	1200000.00E-0004	0.000120E6
+120.	120E3	1200000E-4

4.2.4 String Constants

String constants are sequences of alphanumeric and other printable characters. Line terminators (<CR>) are not allowed. String constants must be enclosed either in single or double quotes. If one type of quotes is used in the sequence itself, the other type must be used to enclose the sequence. String constants may be up to 132 characters long.

A BASIC string prints every character between quotation marks exactly as you type it into the source program. This includes the following:

1. Letters (A–Z)
2. Leading, trailing, and embedded spaces
3. Tabs
4. Special characters (\, ?, !, etc.)

Note, however, that the actual BASIC string does not contain the delimiting quotation marks.

The following are valid string constants:

"THIS IS A STRING CONSTANT."

"SO IS THIS."

"THIS IS A MESSAGE!*/??"

"HERE IS A 'QUOTE' FROM SOMEONE'" *(Note the embedded single quote.)*

'HE SAID "GOODBYE" TODAY' *(Note the embedded double quotes.)*

The following are invalid string constants and the reasons that they are invalid:

“WRONG TERMINATOR’ *(Surrounding quotes must be of same type.)*

‘SAME HERE” *(Surrounding quotes must be of same type.)*

“NO TERMINATOR *(No closing double quote.)*

The following are examples of valid string constants assigned to string variables and string constants used as intermediate values in expressions:

MESSAGE$ = “GEAR BOX FAULT”

MESSAGE$ = “SECTION 12 AIR PRESSURE SAFETY VALVE FAULT”

PART_MESSAGE$ = LEFT$(“SECTION 12 AIR PRESSURE SAFETY VALVE FAULT”, 12)

(Valid string constant not assigned to a variable directly but used an an intermediate value to a function.)

PRINT “SECTION 12 AIR PRESSURE SAFETY VALVE FAULT”

(Valid string constant used in a print statement.)

4.2.5 Boolean Constants

With integers and strings, a constant is used in initializing or assigning a value to a variable. Boolean variables must be assigned the values of true or false. The two boolean constants are “TRUE” or “ON” and “FALSE” or “OFF.”

The following are valid boolean constants:

```
SYSTEM_READY@ = TRUE
OVER_TEMP@ = OFF
```

5.0 EXPRESSIONS

An expression is a symbol or a group of symbols that BASIC can evaluate. These symbols can be numbers, strings, constants, variables, functions, array references, or any combination of these. The following are the different types of operations which can be performed:

1. Arithmetic expressions/operators
2. String expressions/operators
3. Boolean expressions/operators
4. Relational expressions/operators

5.1 Arithmetic Expressions

BASIC allows you to perform addition, subtraction, multiplication, division, and exponentiation with the following operators:

**	Exponentiation
*	Multiplication
/	Division
+	Addition, Unary+
−	Subtraction, Unary−

The unary plus and minus are different from the other operators because they operate on only one operand, not two. The standard operators (binary operators) require two operands.

The following expressions use binary operators:

```
MOTOR_SPEED% + JOG_SPEED%
GAIN% - GAIN_CHANGE%
GAIN% * GAIN_FACTOR%
```

The following expressions use unary operators:

GAIN − SPEED (*Legal but results in a positive value*)
−MOTOR_SPEED%
−(GAIN%+GAIN FACTOR%)
+MOTOR_SPEED% (*This form is not typically used because it is assumed that the absence of an operator in front of a variable means plus or positive*)

Unary minus makes a positive expression negative. Unary plus does not make a negative expression positive. A unary minus applied to a variable already having a negative value will, of course, make the variable (or expression) positive.

The symbols for unary plus and minus (+ and −)are the same as the binary plus and minus, but the operation is different. For example, A% − B% means subtract B% from A%, whereas −(A%) means negative the value of A%.

Performing an operation on two arithmetic expressions of the same data type yields a result of that same data type. For example, A% + B% yields an integer, and K$ + M$ yields a string.

When a real value (constant or variable) is used in an expression with any other numeric data type (single integer, double integer, or real) the result is always real. When a boolean value is used in an expression with a single or double integer variable, the result is always integer.

Table 5.1 lists the arithmetic operators and their meanings. In general, you cannot place two arithmetic operators consecutively in the same expression. The exception is the unary minus and plus and the exponentiation symbol **. For example, A* −B is valid, and A/(−B) is valid, but A+*B is not valid.

Table 5.1 - Arithmetic Operators

Operator	Example	Meaning
+	A + B	Add B to A
−	A − B	Subtract B from A
*	A *B	Multiply A by B
/	A/B	Divide A by B
**	A**B	Calculate A to the power B

BASIC evaluates expressions according to arithmetic operator precedence or priority. Each arithmetic operator has a predetermined position in the hierarchy or importance of operators. This priority tells BASIC when to evaluate the operator in relation to the other operators in the same expression. Refer to table 5.2.

Table 5.2 - Relative Precedence of Arithmetic Operators

Symbol	Operation	Relative Precedence
()	Parentheses	1 (Highest, evaluated first)
−	Unary minus	2
+	Unary plus	
**	Exponentiation	3
*	Multiply	4
/	Divide	
+	Add	5 (Lowest, evaluated last)
−	Subtract	

Operators shown on the same line have equal precedence. BASIC evaluates operators of the same precedence level from left to right. Note that BASIC evaluates A**B**C as (A**B) **C.

In the case of nested parentheses (one set of parentheses within another), BASIC evaluates the innermost expression first, then the one immediately outside it, and so on. The evaluation proceeds from the inside out until all parenthetical expressions have been evaluated. For example, in the expression B = (25+(16*(9**2))), (9**2) is the innermost parenthetical expression and BASIC evaluates it first. Then it calculates (16*81), and finally (25+1296).

BASIC evaluates expressions enclosed in parentheses before the operator immediately outside the parentheses, even when the operator enclosed in parentheses is on a lower precedence level than the operator outside the parentheses. In the statement A = B*(C – D), BASIC evaluates the (C – D) first, and then multiplies B by the result of (C – D).

BASIC will still evaluate other expressions before those in parentheses if the other expressions come first in the statement and have a higher precedence. In the statement below, however, the parenthetical expression occurs later in the overall expression. The exponentiation operation is performed first (before the parentheses) because it is encountered first in the left-to-right evaluation and, at the time it is encountered, is a higher precedence than any operator before it.

BASIC evaluates the expression A = B – C**5 + (X*(Z – 17)), in the following sequence:

C^5	Exponentiation
$B - C^5$	Subtraction with first term
Z – 17	Innermost parenthetical expression
X * (Z – 17)	Next level of parentheses
$[B - C^5] + [X*(Z - 17)]$	Combination of the two expressions

Arithmetic mixing of both single and double precision integers along with real variables and constants is permitted in a BASIC statement. The rules regarding truncation and the maximum size integer-to-real conversion still apply.

The following are valid arithmetic mixing examples:

```
20 GAIN = MOTOR_SPEED%*(OLDGAIN!*13.8876)
```

or

```
20 GAIN! = GAIN! + REFERENCE%
```

The following example could cause an overflow if the resultant value is larger than 16 bits of precision. In such a case, the largest possible single positive or negative integer would be loaded into the variable GAIN%. The program would continue to run, and an error would be logged to notify the user of the problem.

```
20 GAIN% = REFERENCE! + GAIN%
```

5.2 String Expressions

BASIC provides three operations for use with string expressions. These are the assignment operation (=), the concatenation (addition of strings) operation (+), and the equality/inequality comparison operations (=, < >, or > <).

By using the assignment operator, you can equate or assign one string variable or constant to another string variable. In the statement below, the character sequence “THIS IS A MESSAGE” is assigned to the string variable C$:

```
C$ = “THIS IS A MESSAGE”
```

The concatenation operator (+) combines string variables and constants to form a new string expression:

```
C$ = “HI” + B$ + D$ + “STRING”
```

The relational operators“=”and “< >” or “> <” are used when there is a relational or comparison expression, such as that found in an “IF”statement. The statement below tests the value of the boolean which is the result of the string comparison. If the result is true and the two strings are not equal (< >), the program transfers control to line 250. The relational operator “=” is used in the same way to test strings:

```
20 IF C$ < > “MESSAGE” THEN GOTO 250
```

When strings are concatenated (added) and the result is stored in a string variable (A$ = B$ + “TEXT”), the resultant length of the computed string expression still must not exceed the maximum length for a string (255 characters if specified in the variable definition, 31 as a default if no size is specified). If it is longer than the maximum, it is truncated to the maximum and loaded.

If the string expression is used in a relational expression or PRINT statement where it is not assigned to a variable but only exists as a temporary entity, the string expression may be as large as 132 characters.

The following are strings used as intermediate values. In both cases, the string expression enclosed within the parentheses may be as large as 132 characters. If it exceeds that length, it is truncated to 132 characters. This means that BASIC will allow string expressions to be as large as 132 characters while the expression is being computed; however, at the time it is to be assigned to a string variable, it must be able to fit the string into the allocated variable space (31 default, 1-255 if specified in the variable definition):

```
IF (A$ + C$) < > (B$ + “TEXT”) THEN 200
PRINT (“HI”+ A$ + B$ + C$),N%,XYZ
```

5.3 Boolean Expressions

A boolean expression, just like a boolean variable, evaluates to a true/false value. Refer to table 5.3 for the truth table. Boolean expressions use the following boolean operators:

AND	(performs logical “AND” function)
OR	(performs logical “OR” function)
XOR	(performs logical “exclusive-OR” function)
NOT	(unary boolean operator performs a boolean complement)

The following are boolean expressions:

OVER_TEMP@ (Simple boolean variable, TRUE or FALSE)

OVER_TEMP@ AND SHUTDOWN_READY@ (ANDs two boolean results together)

NOT ((A@OR B@ AND C@) *(complex boolean expression)*

Table 5.3 - Truth Table for Boolean Operators

A	B	A AND B	A OR B	A XOR B	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

1= TRUE; 0 = FALSE

The AND boolean operator has a higher precedence than the OR or XOR operators, which have equal precedence. Thus, in a boolean expression, the AND operator will be evaluated before the OR operator. The NOT operator is always applied immediately to the expression (which is the same as the unary minus operator). The following examples show a boolean expression and the order of evaluation of the operators:

A@ = B@ OR C@ AND D@

1. C@ AND D@
2. B@ OR [C@ AND D@]

A@=B@ AND NOT C@ OR D@ AND A@

1. NOT C@
2. B@ AND [NOT C@]
3. D@ AND A@
4. [B@ AND (NOT C@)] OR [D@ AND A@]

A@ = NOT(A@ OR B@AND C@) OR C@ AND NOT D@ OR B@

1. B@ AND C@
2. A@ OR [B@ AND C@]
3. NOT [A@ OR (B@ AND C@)]
4. NOT D@
5. C@ AND [NOT D@]
6. [NOT (A@ OR (B@ AND C @))] OR [C@ AND (NOT D@)]
7. [(NOT (A@ OR (B@ AND C@))) OR (C@ AND (NOT D@))] OR B@

Boolean values can be combined with integer values by using either boolean or arithmetic operators. In combining the two data types, note the following guidelines:

- When a boolean value is combined with an integer value using boolean operators or arithmetic operators, the result is always in integer.

- When a boolean value is combined with an integer value, the boolean is always treated as the value zero (0) if it is FALSE and one (1) if it is TRUE.

The following statement illustrates these rules:

```
A% = SYSTEM_DOWN@*(SPEED_REFERENCE – 2.3376)
```

The value of this statement will be 0 if SYSTEM_DOWN@ is FALSE or (SPEED_REFERENCE – 2.3376) if SYSTEM_DOWN@ is TRUE.

When a boolean operator is used to combine the two different data types, it performs a bit-wise or bit-for-bit operation on the two values, treating the boolean as either an integer one (1) or zero (0):

```
LOWER_BYTE% = ANALOG_IN% AND OFFH
```

This operation will "mask" off the upper 8 bits of the value ANALOG_IN%, which can be useful when manipulating integers as binary data.

5.4 Relational Expressions

It is often necessary in BASIC to compare different values and, based on the result of that comparison, perform one of several actions. These comparisons are done with relational operators and are usually used in conjunction with the IF-THEN statement to create conditional transfers or conditional executions of different parts of a program. Table 5.4 lists the valid relational or comparison operators and their meanings.

In forming relational expressions, similar data types must be compared, i.e., numeric types must be compared to other numeric types. It would be illegal to compare a boolean or integer expression to a string expression using a relational operator.

Table 5.4 - Relational or Comparison Operators

Operator	Example	Meaning
=	A = B	A is equal to B
<	A < B	A is less than B
>	A > B	A is greater than B
< =	A < = B	A is less than or equal to B
> =	A > = B	A is greater than or equal to B
< >	A < > B	A is not equal to B
> <	A > < B	A is not equal to B

The following are relational expressions:

```
A% > B%
((A%+3)/16) < > 32
(((A%/25) +13)+B%) >= SPEED%
(A% < > B%) OR (GAIN => 3.58867) OR (FAULTS% = 0)
(MESSAGE$ = "SYSTEM DOWN")
(SPEED > OLD_SPEED + 23.8876/GAIN)
```

Since the result of a relational expression is a true or false value (boolean result), several relational sub-expressions may be combined by using the boolean operators AND, OR, and XOR.

The following are statements using relational expressions:

```
10  IF (A% > B%) THEN 200

10  IF (SPEED < 32.887) AND (SECTION 5_on@) THEN
    GOSUB 12000

10  IF SYSTEM_STOPPED@ OR FAULTS@ OR (ROLL_
    WIDTH% <23) THEN 240
```

5.5 Mixed Mode (Integers and Reals) Arithmetic

In performing mixed mode arithmetic (expressions in which integers and reals are intermixed), BASIC must always convert the integer value to a real or decimal number internally to be able to operate on the two quantities.

The integer must be converted to a real to maintain the maximum amount of precision possible. (Converting the real to integer and doing all integer arithmetic obviously is not feasible because all the fractional parts would be lost; A=2.37764+4 should result in 6.37764 not 6.) This integer-to-real conversion happens only, however, at the point where the integer value or sub-expression value is combined with a real value in an operation.

When BASIC evaluates an expression. it follows certain rules which determine the order of evaluation in the expression. When using mode arithmetic, use caution to assure the desired results.

If there are integer parts to the expression, BASIC will use integer arithmetic until it encounters a real value and then convert the integer partial result to real. For example, the following expression is evaluated exactly as seen, left to right (because there are not parentheses and all the operators are of the same precedence or importance):

```
REAL3 = B% * C% * D% * REAL1 * REAL2
```

The above statement is evaluated as follows:

1. B% * C% will be calculated in integer arithmetic.
2. The intermediate value of (B% * C%) is then multiplied by D% using integer arithmetic because both quantities are still integers.
3. The intermediate value of [(B% * C%) * D%] is now multiplied by REAL1, but since one of the values is real and one is integer, the intermediate value of [(B% * C%) * D%] must be converted to a real value before the multiplication by REAL1.

4. The intermediate value of ([B% * C%) *D%] * REAL1), which is in real format, is now multiplied by REAL2, also in real format. The result is then a real value which is loaded into variable REAL3.

   Note: If the variable on the left side of the equal sign were an integer, the resultant real value would be truncated first and then loaded into the variable.

The mixing of integers and reals in the previous example does not result in a problem because, although there are intermediate integer values, multiplication operators do not intermediate integer values, multiplication operators do not cause any loss of precision as the operations are performed (4 * 5.334 is the same as 4.00 * 5.334).

Problems may, however, occur when mixed mode arithmetic involves division. Consider the following example in which the operation (A%/B%) must occur first because of the parentheses:

```
10 A%=17:B%=3:REAL=13.7889
20 REAL2=(A%/B%) * REAL
```

The partial result of the first expressions is 5 (17 ÷ 3 = 5.66666; the fractional part 0.66666 is ignored because it is integer division). The 5 is then multiplied by 13.7889, yielding 68.9445 (5 x 13.7889), not 78.1370 (5.66666 x 13.7889).

Once an intermediate result in a BASIC expression is evaluated as a real, the rest of the expression will also be done as real arithmetic. The above expression could be modified as follows to get the full precision from the division:

```
10 A% = 17:B%=3:REAL = 13.7889
20 REAL2=(1.0*A%/B%) *REAL
```

Multiplying the 1.0 (which is a real number) by the variable A% forces A% to be converted to real. The result of (1.0 * A%) is then a real value. Since (1.0 * A%) is real, B% must be converted to real to be used in the division.

The following is a comparison of the execution times for different arithmetic modes doing the same expression. Notice that the third example, combining integer and real values, is the most time consuming because of the conversion required on the integer before the addition can be performed. It is, therefore, faster to do arithmetic in either all real or all integer. If possible:

REAL = REAL + REAL1	250µsec
INT% = INT% + INT1%	210µsec
REAL = REAL + INT%	362µsec

6.0 AutoMax ENHANCED BASIC STATEMENT TYPES

As described in section 3.1, each BASIC statement begins with a line number followed by a keyword. The keyword determines what information will follow on the line. This section describes all the keywords used in AutoMax Enhanced BASIC grouped by statement type as follows:

The format of all statements is defined, along with the parameters required and the permitted variable types. Parameters that are optional are so noted.

6.1 Defining Variable Control

AutoMax Enhanced BASIC requires that all variables be defined in the task, i.e., initialized, prior to their use in the task if they must be at a known state other than 0. See section 4.1.3, Variable Control Types, for more information about variable control and the initial state of variables. Arrays must always be defined prior to their use. Variables are defined using either a LOCAL or COMMON statement. The storage area required for the variables is automatically set aside by these two statements.

6.1.1 LOCAL Statement

The LOCAL statement is used to define three kinds of variables, all of which are “local”, or accessible to, only the task in which they are defined. This means that even if the same variable name is used in another task the values of the two variables are totally independent of each other. Any operation performed within a task on the variable has no effect on the variable in the other task. The following types of variables are defined using the LOCAL statement:

1. Simple variables used only by the task. These variables can be of any data type and can be written to or read from.
2. Subscripted (array) variables used only by the task. These variables can be of any data type and can be written to or read from.

3. Tunable variables. These variables can be double integer, integer, or real type and can be read from, but cannot be written to by any means except through the Programming Executive software running on the personal computer. Note that tunable variables used to define gain parameters affected by auto-tuning on the Universal Drive Controller (UDC) module will also be written to by the operating system on the UDC. See instruction manual S-3006 for more information on UDC tasks.

LOCAL Simple Variable Format

LOCAL variable

where:

variable =	simple variable of any type, i.e., double integer, integer, real, string, or boolean; more than one variable can be defined with one statement by separating the variables with commas

examples:

```
10 LOCAL START_BUTTON@
20 LOCAL LIMIT%, TEMPERATURE, MESSAGE$
```

LOCAL Subscripted (Array) Variable Format

LOCAL variable(size_list)

where:

variable =	simple variable of any type, i.e., double integer, integer, real, string, or boolean
size_list =	up to five integer constants or integer variables separated by commas, each value defining the limit of the dimension of the array; expressions that result in integer values are also permitted; see section 4.1.2 for more information about memory allocation for arrays; note that the first item in every dimension is indicated in location 0, not 1

examples:

```
10 LOCAL A%(25) [reserves space for 26 integer
   elements]
20 LOCAL B@(2,5) [reserves space for 18 boolean
   elements]
```

LOCAL Tunable Variable Format

Tunable variables provide a method of adjusting values within a certain range through the Programming Executive software while the application task is running. Tunable variables can only be read by the application task itself. They cannot be written to, except in the case of tunable variables in UDC tasks, which are also written to by the operating system on the UDC. Tunable variables can never occur on the left side of a LET (assignment) statement.

```
LOCAL variable [CURRENT=val1, HIGH=val2, LOW=val3, &
                         STEP=val4]
```

where:

variable =	simple variable of double integer, integer, or real type
val1 =	a constant of integer, double integer, or real type representing the value of the variable when the task is first downloaded to the Processor; if the variable is modified when the task is running, it assumes the new value as the “CURRENT” value. If the task is reconstructed (uploaded from the Processor to the personal computer), instead of val1, the “CURRENT” value at the time will be printed following the keyword “CURRENT”.
val2 =	a constant of integer, double integer, or real type representing the highest value that the operator can assign to the variable
val3 =	a constant of integer, double integer, or real type representing the lowest value that the operator can assign to the variable
val4 =	a constant of integer, double integer, or real type representing the amount (step) by which the operator can adjust the value by decrementing or incrementing the variable

examples:

```
50 LOCAL TENSION_GAIN%[  CURRENT=25,HIGH=50, &
                              LOW=10,STEP=5]
60 LOCAL RANGE%[ CURRENT=2500,HIGH=3500, &
                      LOW=2000,STEP=50]
```

6.1.2 COMMON Statement

The COMMON or (GLOBAL) statement is used to define two kinds of variables, both of which will be, common, i.e., accessible to all tasks in the rack. The value of the variable is made accessible to all tasks by defining it in the configuration task for the rack and then declaring the same variable common in tasks that need to reference the variable. If you are using AutoMax Version 2.1 or earlier, see J-3649 for more information. If you are using AutoMax Version 3.0 or later, see the AutoMax Programming Executive for more information.

A change in the value of a common variable in one application task will be seen by all application tasks that reference that variable name as a common variable. The following variables are defined using the COMMON (or GLOBAL) statement:

1. Memory variables (variables that are assigned to memory locations) that must be accessible to all tasks in the rack. These variables can be of any data type. They can be read to or written from.
2. I/O variables (variables that that refer to actual physical I/O locations). These variables can be double integer, integer, or boolean variables. Common I/O variables that represent inputs may be read but not written to. Common I/O variables that represent outputs may be read or written to.

Recall that BASIC tasks always use the most current value of common variables when performing calculations, while Control Block and PC/Ladder Logic tasks latch the value of all simple double integer, integer, and boolean variables. See section 4.0 for more information.

Note that in the following examples, GLOBAL can be substituted for COMMON.

COMMON Simple Variable Format

COMMON variable

where:

variable =	simple variable of double integer, integer, real, string, or boolean type for common memory locations; simple variable of double integer, integer, or boolean type for common I/O locations; more than one variable can be defined with one statement by separating the variables with commas

examples:

```
10 COMMON SPEED%
20 COMMON LIMIT%, START@, PROMPT$
```

COMMON Subscripted (Array) Variable Format

COMMON variable(size_list)

where:

variable =	simple variable of any type, i.e., double integer, integer, real, string, or boolean
size_list =	up to five integer constants or integer variables separated by commas, each value defining the limit of the dimension of the array; expressions that result in integer values are also permitted; see section 4.1.2 for more information about memory allocation for arrays; note that the first item in every dimension is indicated in location 0, not 1

examples:

```
10 COMMON A%(10) [reserves space for 11 integer
   elements]
20 COMMON B$:15(2,10) [reserves space for 33 string
   elements of 15 characters each maximum]
```

6.2 Program Documentation

BASIC allows you to insert notes and comments in a task. BASIC provides two statements for this purpose, the REM and ! statements.

REM comment

OR

! comment

where:

comment = any text

The ! format is interchangeable with the REM format for a comment; however, the statements are treated differently by the compiler.

The ! format comments are downloaded with a task onto the Processor module. Consequently, when that task is uploaded to the operator's terminal at a later time, the comments can be reconstructed along with the other program statements.

When the REM format comments are compiled, they are discarded and consequently, they are not downloaded with the task. They are not reconstructed and cannot be referenced with a GOTO or GOSUB type transfer of control because they will not exist in the executable task. The REM format comments serve only to document the source file.

When programming critical applications, note that the ! format comments use memory on the Processor module. ! statements require a small amount of execution time even though they do not actually execute.

The remark (REM) statement must be the only statement on a program line. The remark (!) statement can be either the only statement on a line or it can be one of several statements in a multi-statement line as long as it is the last statement. REM and ! statements may be continued onto more than one line just like any other statement:

```
10 !      THIS IS AN EXTREMELY LONG COMMENT & <CR>
          SHOWING JUST HOW IT IS POSSIBLE TO & <CR>
          CONTINUE A REMARK ACROSS SEVERAL LINES
```

```
10 !      This is an example of a series of several consecutive
20 !      ! statements used to form a block of comments.
30 !      These comments may contain any character except
40 !      a carriage return, for example: !@$%^&*()   &
           +~][}{'".,??&?>%$
```

```
10 REM    This is an example of a series of several
20 REM    consecutive REM statements used to form a
30 REM    block of comments. These comments may
40 REM    contain any character except a carriage
50 REM    return, for example: !@$%&*()
60 REM    +~][}{'",:/.,?\/%$@%%#'
```

The following is a valid remark statement because the remark is at the end:

```
10 A% = B% −23: PRINT A%:! A REMARK
```

The following is an invalid remark statement because the remark is not at the end of the statement:

```
10 REAL = 3.57: ! NEW REMARK: PRINT
```

The line number of a ! remark statement can be used in a reference from another statement, such as a GO TO statement. BASIC only displays the remarks on the personal computer when you edit the program. See the GOTO (GO TO) statement in section 6.4.

6.3 Variable Assignment (LET/SET_MAGNITUDE)

There are two formats for assigning a value to a variable: the LET statement and the SET_MAGNITUDE statement. The "LET" in an assignment is optional. In actuality tasks are more readable without "LET" in front of the assignment. Its use is left to the discretion of the programmer.

The following is the LET statement format:

LET variable = expression

where:

variable =	simple (A%) or subscripted [A%(5)]; variable of any data type.
expression =	can be as simple as a constant or as involved as a complex arithmetic expression. (See section 5.0).

The following are valid LET statements:

```
10   LET SPEED%=25
15   LET MESSAGE$="SYSTEM FAILURE"
95   WINDER_EMPTY@=TRUE
25   GAIN_CHANGE!=GAIN%*13
10   A% = B% + C% – (D% / 234) + 15 –F%
22   TENSION = TENSION – ((19.7765 *  &
     GAIN%)/78 – 12.3) + OLD_REFERENCE
```

The following are invalid LET statements:

10 LET =	A% *(no variable specified on the left side of the equal sign)*
10 LET A% =	"THIS IS A MESSAGE" *(a string value cannot be assigned to an integer variable)*
10 LET A% =	6+22–(B%–34)+(missing term) *(Invalid expression)*

The purpose of the SET_MAGNITUDE statement is to allow the programmer to enter 16-bit hexadecimal values without having to worry about sign extending the numbers into a 32-bit form. The following is the SET_MAGNITUDE statement format:

SET_MAGNITUDE (variable, value)

where:

variable =	a numeric simple variable (integer, double integer) (not an array)
value =	a numeric constant or expression of same type as variable

If the variable specified in the statement is a single integer variable, the value loaded will be only the lower 16-bits of the value field with no sign extension:

10 SET_MAGNITUDE(A%,0FFFFH)
(A% is replaced by 0FFFFH)

10 SET_MAGNITUDE(A%,0C249H OR 1234H)
(A% is replaced by 0D23DH)

```
10 SET_MAGNITUDE(A%,0FEFE2222H)
```
(A% is replaced by 02222H)

Even if the result of the value field is more than 16 bits of significance (all integer arithmetic is done internally as 32 bits), only the lower 16 bits are loaded into the variable (single integer variable). If the variable is a double integer variable, all 32 bits of the value field are loaded into the variable.

Without the SET MAGNITUDE statement, the programmer would need to sign extend 16-bit hex values into a 32-bit form.

For example, when the statement A% = 0FFFFH executes, BASIC attempts to put the value 0000FFFFH into the variable A%. This causes an overflow because the hex number is greater than 32767, the largest single integer. When this happens, BASIC logs an error and loads the variable A% with the greatest possible value (32767). The variable A%, since it is a 16-bit value, will hold the quantity 0FFFFH, but it must be sign extended into a 32-bit form to be handled internally and look like a number in the range 32767 to -32768. Since all hex constants are not sign extended but assumed to have leading zeros in the leading hex digits, 0FFFFH is too large. Sign extending 0FFFFH would result in 0FFFFFFFFH, which is expressed in 2's complement decimal format as the number −1. Therefore, the statement A% = − 1 or A% = 0FFFFFFFFH would properly load the value 0FFFFH into the variable A%.

6.4 Transferring Program Control

At times it may be necessary to transfer control to different sections of a task depending on certain conditions (the value of a variable, the occurrence of an event, etc.). BASIC provides the following statements to accomplish this:

1. GOTO (GO TO) statement
2. ON/GOTO statement
3. GOSUB statement/RETURN statement
4. ON/GOSUB statement

6.4.1 GOTO (GO TO) Statement

The GOTO statement causes the statement that it identifies by line number to be executed next, regardless of that statement's position within the program. BASIC executes the statement at the line number specified by GOTO and continues the program from that point.

The following is the GOTO statement format:

```
GOTO line_number
```

or

```
GO TO line_number
```

where:

line number=

next program line to be executed; can be an integer constant or integer expression. The specified line number can be smaller (go backward) or larger (go forward) than the line number of the GOTO statement.

In the following example:

```
30 GOTO 110
```

BASIC branches control to line 110. BASIC interprets the statement exactly as it is written: go to line 110. There are no rules or conditions governing the transfer.

In the sample program below, control passes in the following sequence:

- BASIC starts at line 10 and assigns the value 2 to the variable A%.
- Line 20 sends BASIC to line 40.
- BASIC assigns the value A% + B% to variable C%
- BASIC ends the program at line 50.
- Line 30 is never executed.
- 10 LET A% = 2

  20 GOTO 40

  30 LET A% = B% + 13

  40 LET C% = A% + B%

  50 END

The GOTO statement must be either the only statement on the line or the last statement in a multi-statement line. If you place a GOTO in the middle of a multi-statement line, BASIC will not execute the rest of the statements on the line:

```
25 LET A% = B% + 178\ GO TO 50\ PRINT A%
```

In the above statement, BASIC does not execute the PRINT statement on line 25 because the GOTO statement shifts control to line 50.

If a ! remark statement is specified in the line number to which control is transferred, BASIC will branch to that statement even though it does no direct processing:

```
10 LET A% = 2
20 GO TO 40
30 A% = B% + 13
40 ! THIS IS ANY COMMENT
50 LET C% = A% + B%
60 END
```

At line 20, BASIC transfers control to line 40. No processing is required for line 40, although some time is required to read the line. BASIC then executes the next sequential statement, line 50.

GOTO statements can use integer expressions instead of a constant as the transfer line number; however, the expression must have an integer as its final data type. For example, in your task, you are reading data from DATA statements and, depending on the value of variable OPTION%, you want to execute a specific routine. Assume the values of OPTION% are 0 thru 10. In this example, the routines or option handlers are located at line numbers 1000, 1100, 1200, etc. (they are 100 apart for the starting line number). This GOTO statement multiplies the value of OPTION % by 100 to get the "hundreds" value of the routine (100, 200, 300, etc.). This "hundreds" value is then added to the base value of all the option handlers, which is 1000:

```
10 BASE_VALUE% = 1000
15 INPUT OPTION%
20 GOTO (BASE_VALUE% + (OPTION% * 100))
        .
        .
        .
1000 !---routine for handling option #0
1010...
1020...

1100!---routine for handling option #1
1110...
1120...

1200!---routine for handling option #2
1210...
1220...
```

If the value resulting from the integer expression does not match any line number in the task, the execution of the task will fall through to the next statement and an error will be logged in the task error log.

The above operation can be performed more efficiently using the "ON GOTO" statement.

6.4.2 ON GOTO Statement

The ON GOTO statement is also a means of transferring control within a program to another line, depending on the result of an integer expression.

The ON GOTO statement has the following format:

```
ON integer_expression GOTO line_number_1,..., line_number_N
```

or

```
ON integer_expression GO TO line_number_1,...,
line_number_N
```

where:

integer_expression =
any arithmetic expression that results in an integer value

line_number_1 through line_number_N =
line numbers to which control is transferred depending on the evaluated expression

The line numbers always correspond to the value of the expression. If the expression evaluates to 1, control is transferred to line_number_1. If the expression evaluates to 2, control is transferred to line_number_2, and so on to line_number_N.

There is no corresponding line number for zero (0). Fractional numbers are truncated to an integer value.

For example, if A%=5, the result of the integer expression A%/3 would truncate from 1 2/3 to a value of 1. If there is no corresponding line number, the next sequential statement after the ON GOTO is executed.

The following are valid ON GOTO statements:

```
20 ON A% GO TO 100, 200, 300, 400

20 ON ((A%) −5)GOTO 100, 205, 300,515
```

6.4.3 GOSUB, ON GOSUB, and RETURN Statements (Subroutines)

A subroutine is a block of statements that performs an operation and returns control of the program to the point from which it came. Including a subroutine in a program allows you to repeat a sequence of statements in several places without writing the same statements several times.

In BASIC, you can include more than one subroutine in the same program. Subroutines are easier to locate and edit if they are located together, usually near the end of the program.

The first line of a subroutine can be any legal BASIC statement, including a remark statement. You can nest subroutines (one subroutine within another) up to the point that memory becomes insufficient to keep the return information for the subroutines.

The **GOSUB statement** has the following format:

```
GOSUB line_number
```

where:

line_number =
line number of the entry point in the subroutine; can be an integer constant or integer expression.

If the result of the expression value does not match a line number in the task, execution falls through to the next sequential statement after the GOSUB. No error is reported.

When BASIC executes the GOSUB statement, it stores internally the location of the next sequential statement after the GOSUB and transfers control to the line specified. BASIC executes the subroutine until it encounters a RETURN statement, which causes BASIC to transfer control back to the statement immediately following the calling GOSUB statement. A subroutine called by a GOSUB must exit by a RETURN statement.

The **RETURN statement** has the following format:

```
RETURN
```

When BASIC executes a GOSUB, it stores the return location of the statement following a GOSUB.

Each time a GOSUB is executed, BASIC stores another location. Each time a RETURN is executed,BASIC retrieves the last location and transfers control to it. In this way, no matter how many subroutines there are or how many times they are called, BASIC always knows where to transfer control.

```
100 IF MOTOR_SPEED%<JOG_SPEED% THEN 80
110 GOSUB 200
120...
130...
140...
      .
      .
      .
200 ! THIS IS A COMMENT
210 MOTOR_SPEED%=MOTOR_SPEED%+(GAIN%/2)
      .
      .
      .
290 RETURN
```

The **ON GOSUB statement** has the following format:

```
ON integer_expression GOSUB line number_1,...,line number_N
```

where:

integer_expression =
: any arithmetic expression that results in an integer value

line_number_1,..., line_number_N =
: line numbers to which control is transferred depending on the evaluated expression

Line numbers always correspond to the value of the expression. The value 1 transfers control to the first line number listed. The value 2 transfers control to the second line number listed, etc. There is no corresponding line number for zero (0). Fractional numbers are truncated to an integer value. If no corresponding line number exists for the result of the integer expression, control falls to the next sequential statement after the ON GOSUB. When a RETURN is executed from one of the subroutines referenced in the ON GOSUB, it returns to the next statement after the ON GOSUB.

The following are valid ON GOSUB statements:

```
20 ON (A% + B% + C%) GOSUB 103,220,475,650
30 ON GAIN% GOSUB 200,300,400,500
```

6.4.4 IF-THEN-ELSE Statement

The IF-THEN-ELSE statement provides a transfer of control based on the result of a relational or comparison expression. It is one of the most frequently used statements in a BASIC application task.

The IF-THEN-ELSE statement has the following format:

```
IF expression THEN
    statement(s)
ELSE
    statement(s)
END_IF
```

where:

expression =	boolean variable or valid relational expression
statement =	statement or series of statements separated by backslashes or colons, or a line number to which to transfer control. If a line number is used, it must be defined by a GOTO statement. If the expression following IF is evaluated as true, all statements are executed. If a line number is used instead of a statement, it can be larger or smaller than the line number of the IF-THEN-ELSE statement.

The IF-THEN-ELSE statement does not allow the line continuator symbol (&). Any number of IF-THEN-ELSE statements can be nested if they are balanced properly. If there is no alternative to follow the statement following THEN, do not use the ELSE keyword.

The following are valid IF-THEN-ELSE statements:

```
50  IF MOTOR_SPEED% = 50 THEN
        PRINT JOG_SPEED%
    ELSE
        PRINT MOTOR_SPEED%
    END_IF

300 IF MOTOR_SPEED% > JOG_SPEED% THEN
        GOTO 700
    ELSE
        GOTO 600
    END_IF

400 IF (A > 3) THEN
        A = A -12
    ELSE
        IF (B > C) THEN
            C = D - 33.2
        ELSE
            C = D + 12.998
        END_IF
        B = B * A
    END_IF
```

The following are invalid IF-THEN-ELSE statements:

```
130 IF SWITCH_34% THEN
        GOTO 300
    ELSE
        GOTO 500
    END_IF
```

```
          (SWITCH_34% is not a boolean variable or a valid
           relational expression)
150 IF A% > B% THEN GOTO 700
    ELSE GOTO 400
    END_IF
    (The keyword THEN must be the last item on the first line)
```

Note that Version 2.0 and later of the AutoMax Programming Executive supports both the IF-THEN-ELSE format described above and the IF-THEN format used in Version 1.0 of the AutoMax Programming Executive (M/N 57C304-57C307). Version 1.0, however, supports only the IF-THEN format.

6.5 Program Looping

A loop is the repeated execution of a set of statements. Placing a loop in a program saves you from duplicating routines and enlarging a program unnecessarily.

For example, the following two programs will print the numbers from 1 to 10:

Program Without Loop

```
10 PRINT 1
20 PRINT 2
30 PRINT 3
40 PRINT 4
50 PRINT 5
60 PRINT 6
70 PRINT 7
80 PRINT 8
90 PRINT 9
100 PRINT 10
110 END
```

Program With Loop

```
10 I% =1
20 PRINT I%
30 I% = I% +1
40 IF I% < = 10 THEN 20
50 END
```

Both of these programs would result in the following being printed:

```
1
2
3
4
5
6
7
8
9
10
```

The program with a loop first initializes a control variable, I%, in line 10. It then executes the body of the loop, line 20. Finally, it increments the control variable in line 30 and compares it to a final value in line 40.

Without some sort of terminating condition, a program can run through a loop indefinitely. The FOR and NEXT statements set up a loop wherein BASIC tests for a condition automatically each time it runs through the loop. You decide how many times you want the loop to run and you set the terminating condition.

The FOR statement has the following format:

```
FOR variable = expression_1 TO expression_2 {STEP
expression_3}
```

where:

variable = simple numeric variable known as the loop index.

expression_1 =	initial value of the index; can be any numeric expression.
expression_2 =	terminating condition; can be any numeric expression.
expression_3 =	incremental value of the index; the STEP size is optional. If specified, it can be positive or negative; if not specified, the default is +1. Expression_3 can be any numeric expression.

The NEXT statement has the following format:

```
NEXT variable
```

where:

variable = same variable named in the corresponding FOR statement

The FOR and NEXT statements must be used together. You cannot use one without the other. If you do, the program cannot be compiled. The FOR statement defines the beginning of the loop; the NEXT statement defines the end. Place the statements you want repeated between the FOR and NEXT statements. You are actually building a counter in your program to determine the number of times the loop is to execute when you use FOR and NEXT.

Here is a simple FOR-NEXT statement example:

```
20 FOR M% = 30 TO 90 STEP 3
30...
40...
50 NEXT M%
```

M% is given the initial value of 30, and BASIC tests to determine if M% is less than or equal to the terminating value of 90. The loop is executed because M% is less than 90. When the NEXT statement is encountered, the value of M% is incremented by 3. BASIC goes back to line 20 and tests again to see if M% is greater than 90. When BASIC reaches the NEXT statement and M% has a value of 87, BASIC adds 3 to M% and tests the result against the terminating value. The result, 90, is not greater than the terminating value of 90, so BASIC executes the loop again. When BASIC reaches the NEXT statement again, it adds 3 to M%, producing 93. Because this is greater than the terminating value, BASIC terminates the loop by transferring control to the next sequential statement after the NEXT statement.

The following program will print the numbers 1 through 11 as shown below the program listing:

```
10 FOR I% = 1 TO 10
20 PRINT I%
30 NEXT I%
40 PRINT I%
50 END
1
2
3
```

```
4
5
6
7
8
9
10
11
```

In the above program, the initial value of the index variable is 1. The terminating value is 10, and the STEP size is + 1(the default). Every time BASIC goes to line 30, it increments the loop index by 1 (the STEP size) until the terminating condition is satisfied. The terminating condition is satisfied when the control variable is greater than 10. Therefore, this program prints the values of I% ten times. When the loop is completed, execution proceeds to line 40 and prints I% again which has been incremented already to 11. When control passes from the loop, the last value of the loop variable is retained. Therefore, I% equals 11 on line 40.

You can modify the index variable within the loop. The loop in the program below only executes once because at line 20 the value of 1% is changed to 44 and the terminating condition is reached.

```
10 FOR I% = 2 TO 44 STEP 2
20 LET I% = 44
30 NEXT I%
40 END
```

If the initial value of the index variable is greater than the terminal value, the loop is never executed. The loop below on the left cannot execute because you cannot decrease 20 to 2 with increments of +2. You can, however, accomplish this with increments of −2 as shown in the right loop.

```
10 FOR 1 = 20 TO 2 STEP 2                10 FOR 1 = 20 TO 2
                                              STEP −2
```

It is possible to jump out of a FOR loop that has been started or executed at least once, but you should not transfer control into a FOR loop that has not been initialized by executing the FOR statement. It will result in a fatal error during run time and the task will be stopped. The following is illegal in a BASIC task because line 20 shifts control to line 40, bypassing line 30:

```
10 ! THIS IS ILLEGAL
20 GO TO 40
30 FOR I=1 TO 20
40 PRINT I
50 NEXT I
60 END
```

FOR and NEXT statements can be placed anywhere in a multi-statement line:

```
10 FOR I%=1 TO 10 STEP 5\ PRINT I%\ NEXT I%
20 END
```

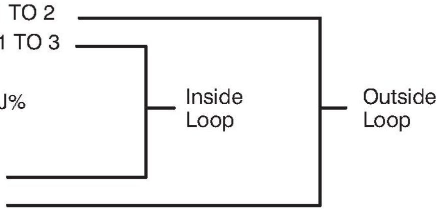
A loop can contain one or more loops provided that each inner loop is completely contained within the outer loop. Using one loop within another is called nesting. Each loop within a nest must contain its own FOR and NEXT statements. The inner loop, the one that starts first, must terminate before the outer loop, which must be completed last. Loops cannot overlap.

The following are two legal nested loops:

```
10 FOR A%=1% TO 10%          10 FOR A%=1 TO 10
20 FOR B=2 TO 20             20 FOR B=2 TO 20
30 NEXT B                    30 NEXT B
40 NEXT A%                   40 FOR C%=3 TO 30
                             50 FOR D=4 TO 40
                             60 FOR E=5 TO 50
                             70 NEXT E
                             80 NEXT D
                             90 NEXT C%
                             100 NEXT A%
```

The following is a program with a legal nested loop:

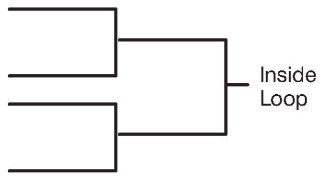
```
10 PRINT "I", "J"
15 PRINT
20 FOR I%=1 TO 2
30 FOR J%=1 TO 3
35 !
40 PRINT I%,J%
45 !
50 NEXT J%
60 NEXT I%
70 END
```



Running the above program would display:

```
 I J

1 1
1 2
1 3
2 1
2 2
2 3
```



The following is an illegal nested loop because the inner loop does not terminate first:

```
10 FOR M=1 TO 10
20 FOR N=2 TO 20
30 NEXT M
40 NEXT N
```

FOR and NEXT statements are commonly used to initialize arrays. As illustrated in this example, line 5 defines a local array with 6 rows and 11 columns. For more information, see section 6.1.1.

```
 5 LOCAL X%(5,10)
10 FOR A%=1 TO 5
```

```
20 FOR B%=2 TO 10 STEP 2
30 X%(A%,B%)=A% + B%
40 NEXT B%
50 NEXT A%
60 END
```

6.6 Statements Used for Multi-Tasking Applications

The typical control application has a variety of functions to perform and monitor. All of these functions require the attention of the CPU, or Processor module. To better service these different functions, BASIC provides a number of multi-tasking capabilities. Multi-tasking is a scheme whereby the operations requiring service are grouped into separate pieces called tasks.

AutoMax Enhanced BASIC provides the following statements to allow the user to break the control application into tasks and then synchronize those tasks:

1. EVENT statement
2. SET statement
3. WAIT statement
4. OPEN CHANNEL statement (INPUT/PRINT channel).

6.6.1 EVENT NAME Statement

An event can be thought of simply as a flag or indicator that one task can set or raise and another task can wait for. There are two types of events used in a BASIC task:

1. Hardware events
2. Software events

Hardware events are generated by an actual external condition, such as an interrupt from a Resolver module (M/N 57C411). Hardware events cannot be used on the AutoMax PC3000. See Appendix C for information on allocating hardware interrupt lines. A software event is simply a flag set by an application task.

An EVENT NAME statement defines a specific event and is used in conjunction with the SET and WAIT statements. The EVENT statement defines a symbolic name for an event. The SET and WAIT statements act on that event. The EVENT statement has two formats, one for a hardware event and one for a software event:

Software Event

```
EVENT NAME = event_name
```

Hardware Event

```
EVENT NAME = event_name,                              &
      INTERRUPT_STATUS = I/O_variable_name,           &
      TIMEOUT = timeout_count
```

where:

event_name =
: symbolic name given to that particular event and the “handle” for further references to that event; not followed by a terminating character.

I/O_variable_name =
: name of a symbol referenced as common in the application task and defined in the configuration task by an IODEF statement. This variable is defined in the configuration task to point to the address of an interrupt register on a hardware module that supports hardware interrupts.

timeout_count =
: the longest time that should pass before a hardware event occurs. This is used as a safeguard in the case where something has happened to the piece of hardware that generates the event. If this timeout period is exceeded before the event is triggered, the system will automatically stop the task. The number specified for the timeout period must be an integer in the range 1-32767 and will always refer to the number of TICKS. The tick rate is user-definable for each Processor being used. The range is 0.5 milliseconds to 10 milliseconds. The default tick rate is 5.5 milliseconds.

The following example defines a hardware event where slot 8, register 2, defines the address of an interrupt register on a resolver module in the main chassis:

Configuration task:

```
10 IODEF RESOLVER_INTREG%[SLOT=8,REGISTER=2]
```

BASIC task:

```
10   COMMON RESOLVER_INTREG%
.
.
.
95   EVENT NAME=MARKER_PULSE,                      &
     INTERRUPT_STATUS=RESOLVER_INTREG%,            &
     TIMEOUT=100
.
.
.
255  IF CONSTANT_SPEED@ THEN                       &
     WAIT ON MARKER PULSE
.
.
.
```

The following example defines a software event for a local BASIC task:

```
95   EVENT NAME=SW_EVENT1
.
.
.
255  IF CONSTANT_SPEED@ THEN WAIT ON SW_EVENT1
```

Note that it is possible to disable the timeout period for a hardware event. Disable the timeout for I/O modules that, unlike the Resolver module, do not generate a periodic interrupt. This format of the event definition should be used carefully since the timeout provides an extra level of protection in the event of a hardware failure. The following is the alternate hardware EVENT NAME format:

```
EVENT NAME=event_name,                              &
      INTERRUPT_STATUS = I/O_variable_name,         &
      TIMEOUT = DISABLED
```

Example:

```
10    COMMON IO_INT_REG%
.
.
.
90    EVENT NAME=HW_EV1,                            &
      INTERRUPT_STATUS=IO_INT_REG%,                 &
      TIMEOUT=DISABLED
```

The format is identical to that for a hardware event except that the word “DISABLED” is entered in place of an integer constant for the timeout period. The timeout field cannot be left off or set to zero (0). This forces the user to turn off the timeout by entering the word “DISABLED”.

Note that the limit on the number of hardware and software events in all tasks in a rack is 32.

6.6.2 SET and WAIT ON Statements

The SET statement is used to set an event or indicate that it has occurred. Executing this statement makes any other tasks that were suspended while waiting for that event to occur (or setting of that event) eligible to run. WAIT ON causes task execution to stop until the EVENT NAME is set by the SET statement. The format of the SET statement is:

```
SET event_name
```

where:

event_name =
name of the hardware or software event previously defined by at least one task

The format of the WAIT ON statement is:

```
WAIT ON event_name
```

where:

event_name =
previously defined by at least one task; same as the corresponding SET statement.

The following is an example of EVENT NAME, SET, and WAIT ON statements. Task ABC is performing a calculation every 40 milli-seconds. Task XYZ is suspended waiting for event GAIN_OVER to occur so that it can perform some calculations of its own. Task XYZ needs to run only when event GAIN_OVER occurs, which, in this example, will indicate that the variable GAIN is beyond a certain maximum.

Task ABC

```
10 EVENT NAME=GAIN_OVER
.
.
.
90 IF GAIN>MAX_GAIN THEN SET GAIN_OVER
.
.
.
```

Task XYZ

```
30 EVENT NAME=GAIN_OVER
.
.
.
105 WAIT ON GAIN_OVER
.
.
.
```

6.6.3 OPEN CHANNEL Statement

The OPEN CHANNEL statement equates a logical name with a data channel between two application tasks in the system. This statement enables tasks to communicate with each other using INPUT and PRINT statements. When a channel is opened between two tasks and one task PRINTs to the other, the information is going to the waiting task, not to a printer or serial port. When one task INPUTs data from another, the data is being read from the other task that has opened the channel.

The OPEN CHANNEL statement has two formats depending upon which task it is being used in. The task sending data through the channel must use the OPEN CHANNEL FOR OUTPUT format. The task reading data from the channel must use the OPEN CHANNEL FOR INPUT format. Multiple tasks can open the same channel for output purposes and write data to it, but only one channel may open the channel for input purposes and read from it.

Format for task writing to a channel:

```
OPEN CHANNEL FOR OUTPUT AS FILE #n, TYPE = (type)
```

Format for task reading from a channel:

```
OPEN CHANNEL FOR INPUT AS FILE #n, TYPE = (type), DEPTH = depth
```

n	=	logical number assigned to the channel; range = 1-255; the PRINT and INPUT statements must reference the same number to identify this particular channel.
type	=	variable type or list of types that are to be passed through the channel; multiple types must be separated by a comma; this information allows the system to format each piece of data when it passes it from one task to another.

I	=	single integer
D	=	double integer
R	=	real
S	=	string
B	=	boolean

Any combination of the above variable types is permitted; however, a PRINT to or INPUT from this specific channel must always pass the same number and type of variables and in the same order that the type field specifies.

depth = how many messages this channel can hold before it is considered full; this value must be an integer constant or expression; specified only for the OPEN CHANNEL statement in the task that is reading from the channel.

The task that is reading data (i.e., that contains the OPEN CHANNEL FOR INPUT statement), creates the channel. If a task that is writing data to a channel runs before the task that is reading data from the channel, the task writing data is suspended until the task that is reading data runs.

If a task tries to write data to a channel that is full (has reached the maximum DEPTH), then that task will be suspended until one of the queued messages is read by the task reading from the channel. If a task attempts to read from a channel which is empty, that task will be suspended until a message has been placed in the channel. Note that the INPUT statement has a parameter (EMPTY) that allows the task reading from the channel to transfer control to another line if the channel is empty. The PRINT statement has a parameter (FULL) that allows the task writing to the channel to transfer control to another line if the channel is full.

Reading and writing to a channel can be used to synchronize activities between two or more tasks. For example, task 1 could perform some initialization and then INPUT from a channel. If the channel is empty the task will be suspended until another task PRINTs to the channel. Task 1 could then become active again when one of several other tasks PRINTs information into the channel (such as error or warning conditions). Task 1 could then record this information and INPUT from the channel again, suspending the task until more information is available.

The following are two examples of OPEN CHANNEL statements and the corresponding INPUT and PRINT statements from that channel:

Task ABC

```
10 OPEN CHANNEL FOR OUTPUT AS FILE #1, TYPE = (I,I,S,R)
      .
      .
      .
95 PRINT #1,(GAIN%*13),SPEED%,MESSAGE$,RATIO
      .
      .
      .
```

Task XYZ

```
15 OPEN CHANNEL FOR INPUT AS FILE #1, TYPE = (I,I,S,R), DEPTH = 10
      .
      .
      .
75 INPUT#1,NEW_GAIN%,CALC_SPEED%,TEXT$,RATIO
      .
      .
      .
```

Note that the PRINT and INPUT statements have the same format here as when they are used to access a device, except that OPEN CHANNEL is used to define the logical channel number instead of OPEN “device_name.” See section 6.8.

Tasks do not have to call the variables by the same names when reading and writing to the channel; they simply write an integer quantity (for an integer position in the template) and read an integer quantity into an integer variable. Note that, just like the normal print operation, the print list of items may contain any kind of expression, which will be evaluated first and then printed. The number of items and their data types used in a print or input to a channel must match the OPEN CHANNEL definition; otherwise, an error will occur.

A CLOSE statement is not used with a channel because once it is opened, it remains open. There is no provision to open and then close channels.

6.7 Real Time Enhancements

BASIC provides two statements that allow specified sections of tasks to be executed at a known time.

1. DELAY statement
2. START statement

6.7.1 DELAY Statement

The DELAY statement provides for a simple time delay in a task.

The following is the DELAY statement format:

```
DELAY n time_units
```

where:

n = any arithmetic expression or constant that evaluates to an integer result

time_units = unit of time to be delayed

The possible time units for both the DELAY and the START statement are TICKS, SECONDS, MINUTES, and HOURS. The tick rate is user-definable for each Processor being used. The range is 0.5 milliseconds to 10 milliseconds. The default tick rate is 5.5 milliseconds. The plural form of the time unit must always be used, even when referring to one unit, e.g., DELAY 1 HOURS.

The following are valid DELAY statements:

```
30   DELAY 255 TICKS
80   DELAY ((OLD_TICKS%−2)*5) SECONDS
15   DELAY (1ST_SHIFT_LNG%) HOURS
40   DELAY 1 HOURS
```

In each of the above examples, when the DELAY statement executes, the task will be suspended at that point for the specified amount of time and then be eligible to run when the time interval expires. At that time, the task begins execution at the line following the DELAY statement.

6.7.2 START EVERY Statement

The START EVERY statement format is similar to the DELAY statement format but is used to do a periodic re-start or scan of the task. The format of the START statement is:

START EVERY n time_units

where:

n = any arithmetic expression or constant that evaluates to an integer result

time_units = unit of time to be delayed before starting

The possible time units for both the START and the DELAY statements are ticks, seconds, minutes, and hours. The tick rate is user-definable for each Processor being used. The range is 0.5 milliseconds to 10 milliseconds. The default tick rate is 5.5 milliseconds. The plural form of the time unit must always be used, e.g., DELAY 1 HOURS.

When a START EVERY statement executes, it notifies the operating system that the task needs to be re-started “n time units” from now, starting with the statement following the START EVERY statement. After notifying the operating system, control is passed back to the task, which continues to execute. The task will eventually execute an END statement, which causes that task to relinquish control. When the time interval in the START EVERY statement expires, the operating system makes that task eligible to run. It will run unless there is a higher priority task eligible to run.

The point in time when the task begins executing again is based on how long the task runs after the START is encountered and before an END statement is executed. Consider the following example:

```
10 . . .
20 . . .
      .
      .
      .
200 START EVERY 20 TICKS  ──┐  This body of the pro-
      .                     │  gram will be eligible
      .                     ├─ to run every 20 ticks
      .                     │  starting after the START
850 END                   ──┘  EVERY statement.
```

The DELAY statement tells the operating system to stop the task where it is and continue running it after a certain length of time.The START EVERY statement effectively defines for the operating system a re-start point and a time interval. The operating system will automatically start the task at the next statement after the START EVERY when that time period expires. The execution time required for program body between the START EVERY and the END statements must not be longer than the specified time period or an overlap error will occur.

By using the START EVERY statement, the programmer can program tasks to be scanned at a certain frequency, enabling him to do the slower control loop functions in BASIC if he desires. However, BASIC may be too slow to accommodate some high speed control requirements. For most applications Control Block language should be used. For more information, refer to the AutoMax Control Block Language Instruction Manual (J-3676).

6.8 Communication Capabilities

BASIC communicates with other processing elements in a system, including operator's terminals and other application tasks, using the following statements. A number of these statements access Processor serial ports.

1. OPEN statement
2. CLOSE statement
3. INPUT statement
4. PRINT/PRINT USING statement
5. IOWRITE statement
6. GET statement
7. PUT statement
8. READ statement
9. DATA statement
10. RESTORE statement

6.8.1 OPEN Statement

The OPEN statement allocates a Processor port to allow application tasks to communicate with an external device such as a personal computer. Once you allocate a port, you can use the INPUT, PRINT, GET, or PUT statements to communicate through the port. Depending on the port you are allocating, you may also need to use the CLOSE statement in conjunction with the OPEN statement.

You can allocate a port only in a task running on the Processor that contains the port. The OPEN statement cannot be used to communicate with other tasks. See the OPEN CHANNEL statement description in section 6.6.3 for information about inter-task communication.

Ports You Can Use for Serial Communication

Multibus rack-based Processor modules have two serial ports, labeled "PROGRAMMER/PORT B" and "PORT A", reading top to bottom. Use the PROGRAMMER/PORT B port for connecting the Processor to the personal computer running the AutoMax Programming Executive software. The other port, PORT A, is available for use by application tasks. If there are multiple Processor modules in the rack, only the leftmost PROGRAMMER/PORT B port is reserved for communicating with the Executive software. All other Processor ports in the rack are available for use by application tasks.

The PC3000 Processor also has two serial communication ports. The 25-pin D-shell port works exactly like the PROGRAMMER/PORT B port on the rack-based Processor described above. The 9-pin port works like the

PORT A port on the rack-based Processor. On the PC3000, however, both ports may be available for communication with external devices, depending on the setting of jumper JP2. Refer to the PC3000 User Manual, J2-3096, for more information on setting the jumper.

Differences Between Allocating Ports A and B

How you use the OPEN statement depends on whether it is being used with port A or port B. For port A, the OPEN statement is only used to change the default setup and baud rate for the port. For port B, the OPEN statement is used to change the default setup and baud rate and to allocate the port. Port B must be allocated using the OPEN statement prior to using any of the serial port statements to communicate through the port.

OPEN Statement Format Used To Modify Port Setup Characteristics

OPEN "device_name" AS FILE #logical_device_number,
SETUP=specs, baud_rate

OPEN Statement Format Used To Set Port Allocation Status

OPEN "device_name" AS FILE #logical_device_number,
ACCESS=status

where:

device_name =

The pre-assigned name of the port; PORTA for port A or PORTB for port B.

logical_device_number =

The number assigned to the port in the OPEN statement. The # symbol is required. Range: 1-255.

The number used when referencing port A or port B in the CLOSE statement. See Configuring and Using Port A and Configuring and Using Port B, below.

The number used when referencing port B in the INPUT, PRINT, GET, and PUT statements. See Configuring and Using Port B, below.

specs =

A hexadecimal single word constant or integer expression bit pattern that defines various characteristics for the port. The bit positions are defined below. The parameters SETUP and ACCESS cannot be specified in the same OPEN statement.

The default setting is 0D00 (hex). Bits 0-7 of the word specify the terminating character for the INPUT statement as a hex value for the ASCII character. The specified terminating character must be an ASCII value between 20 hex and 7E hex. Using an ASCII value outside this range causes the error message "Invalid string character received" to be logged when the INPUT statement reads data from the serial port. If these bits are left at zero (0), the default input termination character, a carriage return (0D hex), is used.

When software handshaking (X-ON, X-OFF) is enabled, X-OFF (CTRL-S, 13 hex) will be sent when more than nine characters are in the receive buffer. When a task (INPUT or GET statement) empties the receive buffer, X-ON (CTRL-Q, 11 hex) will be sent. The ASCII characters X-OFF or X-ON can not be transmitted as user data because they would be interpreted as flow control commands, not user data.

When hardware handshaking is enabled for a port, the DTR (Data Terminal Ready) pin on the Processor port is false when more than 53 characters are in the receive buffer. The DTR pin on the Processor port is true when the receive buffer is emptied. Refer to the appropriate Processor manual for the port wiring required for hardware handshaking.

Note: *The DSR pin on the Processor port must be true when hardware handshaking has been enabled. True is defined as +5 to +12 volts and false is defined as −5 to −12 volts.*

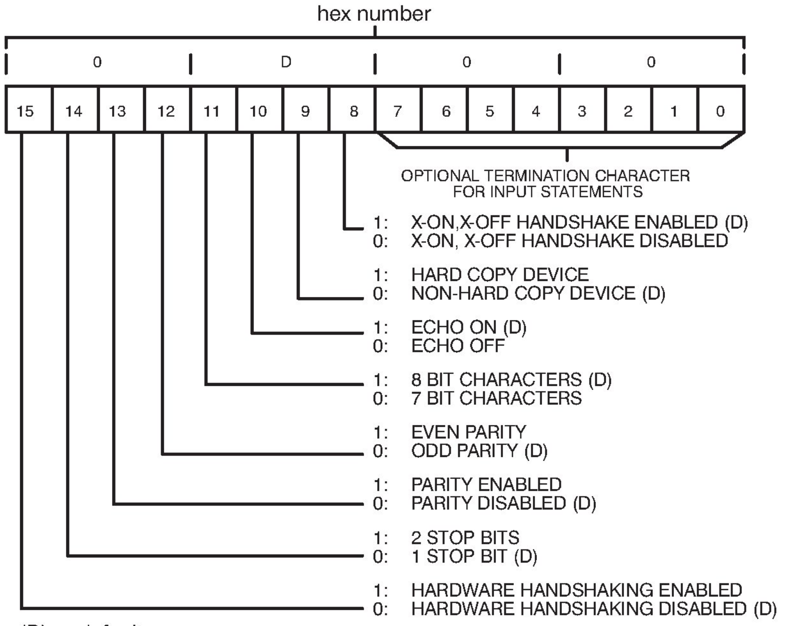
The RTS (Transmit Status, Modem Enable) pin on the Processor port can be controlled in an application task by using the RTS_CONTROL@ function or by the operating system if hardware handshaking has been enabled.

The purpose of the RTS signal is to "bracket" the character transmission. To bracket a character transmission, RTS must be set true prior to the first character being transmitted and remain true until all of the characters have been transmitted. The RTS signal can be used to enable/disable any type of external equipment, such as a tri-state transmit modem, which requires an enable signal to output characters.

When hardware handshaking has been enabled for a port, the operating system will automatically bracket the character transmission. RTS is set true when data is loaded into the transmit buffer and remains true until all the characters have been transmitted.

If the external equipment being controlled requires an enable/disable time of more than 1 msec, RTS must be controlled in an application task using the RTS_CONTROL@function. Refer to the RTS_CONTROL@ function for a description of the operation and an example program.

If hardware handshaking is enabled and RTS is set true using RTS_CONTROL@, RTS will remain true after all the characters have been transmitted so that the application task can use RTS_CONTROL@ to set it false.



(D) = default

baud_rate =

134, 150, 200, 300, 600, 1200 (default), 1800, 2400, 4800, 9600, and 19200 (19,200 for Processor module M/N 57C435 and PC3000 Serial Option card). The baud rate must be specified if the SETUP parameter is specified.

status =

Required only when using port B on a Processor. See Configuring and Using Port B, below, for details. An optional parameter. Specify as NON_EXCLUSIVE when more than one task in the Processor must have access to port B. The parameters SETUP and ACCESS cannot be specified in the same OPEN statement.

If the ACCESS parameter is not specified, the access status is EXCLUSIVE. If the status is EXCLUSIVE, no other task can read or write to the port until the port is closed. Refer to Configuring and Using Port B, below, for details.

Configuring and Using PORT A

For port A, the Processor's operating system allocates and de-allocates the port for access by any task in the Processor without using an OPEN statement in any task. The operating system will automatically allocate the port before executing the INPUT, PRINT, GET or PUT statements and de-allocate the port after executing the statements. It will manage the port allocation and error handling between tasks without any software interlocking being added in any of the tasks.

If the operating system default port setup and baud rate described are acceptable for your application, you can simply use the INPUT, PRINT, GET, or PUT statements in any task on the Processor to communicate through port A.

If the default setup or baud rate are not suitable for your application, use the following OPEN statement format to temporarily allocate the port for EXCLUSIVE access to change the default setup or baud rate. This changes the default characteristics of the port until the next power cycle or Stop-All-Clear occurs. Use the CLOSE statement to de-allocate the port. The CLOSE statement must follow the OPEN statement with no intervening INPUT, PRINT, GET or PUT statements. The CLOSE statement must use the same device number assigned in the OPEN statement.

```
OPEN "PORTA" AS FILE #logical_device_number, SETUP=specs,
    baud_rate \&

CLOSE #logical_device_number
```

Note: *When an OPEN statement has a setup parameter, the port is opened with EXCLUSIVE access. Therefore, specifying ACCESS=NON_EXCLUSIVE causes a compile error to be logged. It is not necessary to OPEN port A for NON_EXCLUSIVE access; this is the default mode of operation. Do NOT use the device number assigned in the OPEN statement to access the port in any subsequent INPUT, PRINT, GET or PUT statements because the port has been de-allocated by the CLOSE statement above.*

Examples:

```
    INPUT INP%
    INPUT:EMPTY=line_no, INP%
    PRINT MSG$
    GET CHAR$
    GET:EMPTY=line_no, CHAR$
    PUT CHAR$
```

Port A - Application Notes

1. Use port A instead of port B when port A is available because the operating system manages port allocation and error handling between tasks.
2. "OPEN" the port to set required port characteristics, "CLOSE" the port, then use INPUT, PRINT, GET and PUT statements without a device number in any task to access port A.
3. All tasks have only NON_EXCLUSIVE access to the port.
4. The port is opened and closed after each operation.
5. Stop-All-Clear commands and power cycle close all ports. The port characteristics are set to OS default. The user must set the port to the required characteristics.
6. Stopping a task does not change the port setup and baud rate.
7. Errors logged to task information (for example, Framing, Overrun, etc.) will have no major effect because the port is opened and closed after each operation.
8. The serial port statements Print, Input, Put, and Get will operate after an error has been logged because the port is opened and closed after each operation.

Programming Example

The following example uses two tasks on the Processor to transmit and receive data from port A. KYBD.BAS sets the characteristics for port A and then closes the port. If USERNAME$ is blank, KYBD.BAS prompts the user to enter his name. DISPLAY.BAS displays the message “Hello World” and the user name if USERNAME$ is not blank. Because KYBD.BAS closes the port after setting the port characteristics, any task in the Processor can access (read from or write to) the port without using a logical device number with the INPUT, PRINT, GET or PUT statements.

{KYBD.BAS task}

```
100 COMMON USERNAME$
 :
900 USERNAME$ = “”
 :
1000        OPEN “PORTA” AS FILE #1, SETUP=(0800H,
            9600) \ CLOSE #1
 :
2000        IF USERNAME$ = “” THEN
              PRINT ; CLRSCR$(2); CURPOS$( 10,10);
              “Please enter your name “;
              INPUT  USERNAME$
            END_IF
2100        DELAY 10 TICKS \ GOTO 2000
 :
32767       END
```

{DISPLAY.BAS}

```
100         COMMON USERNAME$
 :
900         USERNAME$ = “”
 :
2000        IF USERNAME$ <> “” THEN
              PRINT ; CURPOS$(20,10); “Hello World, my
              name is “; USERNAME$;
            DELAY 5 SECONDS \ USERNAME$ = “”
            END_IF
2100        DELAY 10 TICKS \ GOTO 2000
 :
32767       END
```

Configuring and Using PORT B

For port B, you must use the OPEN statement in a task to allocate the port before the task can communicate through the port, even if the default port setup and baud rate are acceptable for your application.

If the default setup or baud rate are not suitable for your application, use the following OPEN statement format to temporarily allocate the port for EXCLUSIVE access to change the default setup or baud rate. This changes the default characteristics of the port until the next power cycle or Stop-All-Clear occurs. Use the CLOSE statement to de-allocate the port. The CLOSE statement must follow the OPEN statement with no intervening INPUT, PRINT, GET or PUT statements. The CLOSE statement must use the same device number assigned in the OPEN statement.

```
OPEN “PORTB” AS FILE #logical_device_number,
SETUP=specs, baud_rate \&

CLOSE #logical_device_number
```

Note: *When an OPEN statement has a setup parameter, the port is opened with EXCLUSIVE access. Therefore, specifying ACCESS=NON_EXCLUSIVE causes a compile error to be logged. A second OPEN statement must be added after the port is closed as shown below.*

If the default or configured port setup and baud rate are proper for your application, use the following OPEN statement format to allocate the port for NON_EXCLUSIVE access so the task can communicate through the port. The port must remain open to execute subsequent INPUT, PRINT, GET, or PUT statements.

```
OPEN “PORTB” AS FILE #logical_device_number,
ACCESS=NON_EXCLUSIVE
```

The format of the INPUT, PRINT, GET, or PUT statements used to communicate through the port is INPUT #, PRINT #, GET # or PUT #, where # is the device number assigned to the port in the OPEN statement. Whenever you refer to the port in subsequent statements, always use the device number assigned to the port in the OPEN statement.

In addition, if you want to make the port accessible to multiple tasks in the Processor at the same time, you must add an OPEN statement with the ACCESS parameter specified as NON_EXCLUSIVE in each task that must communicate through the port using the same logical_device_number that was used to identify the port in the first task.

You must add software interlocking to each task attempting to allocate the port to ensure that the task can determine whether the port setup and baud rate have been initialized prior to opening the port for NON_EXCLUSIVE access. Only one task needs to allocate the port and define the setup and baud rate initially. Software interlocking is also needed for port error handling. Tasks may need to close and re-open the port on error conditions.

Port B Application Notes

1. Port B requires port allocation and error handling to be handled by the application tasks.
2. “OPEN” the port to set the required port characteristics, then “CLOSE” the port.
3. “OPEN” the port for NON_EXCLUSIVE access in each task that needs access to the port. Use the same device number to open the port in all tasks.
4. Use INPUT #, PRINT #, GET #, and PUT # statements with the device number used in the OPEN statement to access port B.
5. Stop-All-Clear commands and power cycle close all ports. Port characteristics are set to the OS default; the user must set the port to the required characteristics.
6. Stopping a task does not change the port setup and baud rate.
7. Stopping a task that has EXCLUSIVE access to the port closes the port on the Processor.
8. Stopping a task that has NON_EXCLUSIVE access only disables access to the port for the task that was stopped. The other tasks referencing the port can still transmit or receive data.
9. The operating system does not de-allocate the port (make the port available for EXCLUSIVE access) until all tasks referencing the port have closed the port.
10. Port errors logged to task information (for example, Framing, Overrun, etc.) will close the port that logged the error.
11. If the port is closed via CLOSE or error, then port statements PRINT, INPUT, PUT and GET will fail. They will log a “Port Not Open (properly)” error message and abort the operation. No data will be sent for PRINT and PUT. INPUT and GET will not suspend the task but will proceed to the next statement on the current line or next line number. INPUT:EMPTY= and GET:EMPTY= will not branch to the specified EMPTY line number, but will operate like the INPUT and GET statements described above.

Programming Example

The following example uses two tasks on the processor to transmit and receive data from port B. Software interlocking must be added to each task attempting to allocate the port to ensure that the tasks can determine whether the port setup and baud rate have been initialized prior to opening the port for NON_EXCLUSIVE access. KYBD.BAS sets the characteristics for port B, closes the port, re-opens the port for NON_EXCLUSIVE access, and then sets the software interlocking flag KYBD_RDY@ TRUE so that DISPLAY.BAS can open the port for NON_EXCLUSIVE access.

If USERNAME$ is blank, KYBD.BAS prompts the user to enter his name. DISPLAY.BAS displays the “Hello World” message and the user name if USERNAME$ is not blank. Because the port must be opened by each task that needs access to the port, the logical device number used in the OPEN statement must be used with the INPUT, PRINT, GET or PUT statements.

{KYBD.BAS task}

```
100         COMMON USERNAME$
110         COMMON KYBD_RDY@,     DISP_RDY@
 :
900         USERNAME$ = “”
910         KYBD_RDY@ = FALSE
 :
1000        IF DISP_RDY@ THEN DELAY 10 TICKS \ GOTO
            1000
1100        OPEN “PORTB” AS FILE #2, SETUP=(0800H,
            9600) \ CLOSE #2
1110        OPEN “PORTB” AS FILE #2,
            ACCESS=NON_EXCLUSIVE
1120        ! Add initialization code as required
 :
1200        KYBD_RDY@ = TRUE
1210        IF NOT DISP_RDY@ THEN DELAY 10 TICKS \
            GOTO 1210
 :
2000        IF USERNAME$ = “” THEN
              PRINT #2; CLRSCR$(2); CURPOS$( 10,10);
              “Please enter your name “;
             INPUT #2  USERNAME$
            END_IF
2100        DELAY 10 TICKS \ GOTO 2000
 :
32767       END
```

{DISPLAY.BAS}

```
100         COMMON USERNAME$
110         COMMON KYBD_RDY@,     DISP_RDY@
 :
900         USERNAME$ = “”
910         DISP_RDY@ = FALSE
1000        IF KYBD_RDY@ THEN DELAY 10 TICKS \ GOTO
            1000
1010        ! Add initialization code as required
 :
1200        IF NOT KYBD_RDY@ THEN DELAY 10 TICKS \
            GOTO 1200
1210        OPEN “PORTB” AS FILE #2,
            ACCESS=NON_EXCLUSIVE
1220        DISP_RDY@ = TRUE
 :
2000        IF USERNAME$ <> “” THEN
              PRINT #2 ; CURPOS$(20,10); “Hello World,
              my name is “; USERNAME$;
            DELAY 5 SECONDS \ USERNAME$ = “”
            END_IF
2100        DELAY 10 TICKS \ GOTO 2000
 :
32767       END
```

6.8.2 CLOSE Statement

The CLOSE statement is used to de-allocate a channel or port to allow other application tasks to have access to it. The CLOSE statement has the following format:

```
CLOSE #logical_device_number, #logical_
device_number,...
```

where:

logical_device_number = number assigned to device by OPEN statement (1 to 255)

The following are valid CLOSE statements:

```
20 CLOSE #1,#2
99 CLOSE #3
```

The number symbol (#) must always be present in front of the logical device number assigned to the device.

If PRINT or INPUT statements use the default port (e.g., PRINT F% or INPUT M$), there is no need for either an OPEN or CLOSE statement.

Channels that are opened, as opposed to devices, remain opened until the task is restarted.

Refer to the OPEN statement description (6.8.1) for more informtion on how the CLOSE statement is used with the OPEN statement.

6.8.3 INPUT Statement

The INPUT statement is used to prompt an operator to input data from a device or channel. It has the format:

```
INPUT #logical_device_number, input_list
```

where:

logical_device_number =
number assigned to device or channel by OPEN statement (1 to 255). Not used if default device is used.

input_list =
list of variables to be read. Variables can be simple or subscripted (no expressions).

The optional parameter :EMPTY=n can be used to transfer control of the program in the event that the channel is empty. This option is only allowed for channels and not devices. The parameter is added immediately after the device number; n is the line number to which to transfer control.

If more than one variable is to be read, the fields must be separated by commas when entered. Any extra spaces or tabs between the input fields are ignored.

When an INPUT statement is executed in BASIC, an input prompt is printed to the device to indicate that the system is waiting for something to be entered. The input prompt is a question mark followed by a space (?).

The number of fields requested in the INPUT statement are then entered at the device, followed by the terminator character (<CR>). If all of the requested fields are not entered prior to the (<CR>), the system will prompt again, indicating that it is waiting for more data. The system will also prompt again if the operator enters just a <CR> by itself. It will reprompt after every <CR> until all the expected data is received. **Note that the receive FIFO queue will accept a maximum of 64 bytes. If more than 64 bytes are received without the queue being emptied, the additional data is discarded.**

In the following example, 3 integer fields are expected by the INPUT routine, but only 2 are entered the first time. BASIC prompts again for the third number to be entered. The fields can be separated with commas or spaces if the prompt is off. If the prompt is on, each field is separated with <CR>.

In the task:

```
10 INPUT A%, B%, C%
```

On the display screen:

```
?34,56          <CR>

?46             <CR>
?
```

If you make a mistake while entering data, use the backspace key to back up the cursor and enter the correct data. Enter <Crtl-U> to cancel any data on the line. If the data entered is totally incorrect, enter <Ctrl-C> to cancel the INPUT completely. If this is done, none of the variables in the INPUT statement will be updated and execution will continue with the next statement after the INPUT.

When the prompt is enabled, certain error messages are displayed to the user if his input data is invalid. For example, if the statement is INPUT A% and the value entered is either out of range (not +32767 to −32768) or of the wrong type (string instead of integer), the system will prompt:

```
>>>>>INVALID DATA TYPE - INPUT AGAIN<<<<<<
```

and re-prompt for data that will fit into a single integer (+32767 to −32768).

You can enable or disable the input prompt by writing to the variable "_PROMPT@". This symbol is automatically allocated as a local variable when the task is created. At power-up, it has the value TRUE (print a prompt character during input). If the value of this variable is changed to FALSE, the prompt is disabled.

You may find it desirable to disable the prompt when you are soliciting input from a personal computer and do not wish to have the prompt character appear on the display. However, when the prompt is disabled, any input errors that occur will result in a run-time error. Program execution will continue with the next statement. Therefore, when you disable the prompt, you should use the ON ERROR GOTO statement to initiate an error-handling routine.

The following are valid INPUT statements:

```
20 INPUT #1,A%,B%

30 INPUT A%,B%

40  INPUT #3:FULL=80, C%
```

It is possible to define a character other than carriage return (<CR>)as the terminator for data in an INPUT statement. This is done by loading the ASCII value of the character into the lower 8 bits of the device characteristics word in the SETUP portion of the OPEN statement. For example, if you wanted to use a question mark (hexadecimal 3F in ASCII) as the terminating character, you would open the device as follows:

```
OPEN "PORTA" AS FILE #2, SETUP=(OD3FH,1200)
```

The same rules apply to inputs not terminated by a carriage return as to the disabled prompt input: no error messages and all data must be entered at the same time.

Using a different terminator is more useful for those tasks communicating to other devices or computers where the data transmitted is not terminated with <CR> but another character. This allows reading data from one of those devices without having to accumulate the characters one at a time with a GET statement.

Refer to the OPEN statement description (6.8.1) for more information on how the INPUT statement is used with the OPEN statement.

6.8.4 PRINT/PRINT USING Statements

The PRINT and PRINT USING statements are used to communicate with I/O devices, such as a personal computer, or a line printer, or another BASIC task. The OPEN statement is used to select to which I/O port the PRINT/INPUT applies. Refer to the OPEN statement description (6.8.1) for more information on how the PRINT statement is used with the OPEN statement. The PRINT statement has the following basic format (PRINT USING is defined separately in this section):

```
PRINT #logical_device_number, print_list
```

where:

logical_device_number = logical number (1 to 255) assigned to a device or channel during an OPEN statement. If no device number is given, the default device is PORTA on the Processor module on which this tasks resides. Most application task I/O will be handled through this default port.

print_list = list of data items to be printed, such as:

- Integer variables and integer expressions
- Real variables and real expressions
- Boolean variables and boolean expressions
- String variables and string expressions

The optional parameter :FULL=n can be used to transfer control of the program in the event that the channel is full. This option is only allowed for channels and not devices. The parameter is added immediately after the device number; n is the line number to which to transfer control.

If the logical_device_number specifies a channel and not a device, the number and type of items printed must match exactly the channel template as specified in the OPEN CHANNEL statement.

The following is a typical PRINT statement using the default port (PORTA):

```
40 PRINT A%,B%, MESSAGE$,C% + D%
```

The output line would be 13 29 MOTOR 53, assuming the following values for this statement:

```
A%=13
B%=29
C%=36
D%=17
MESSAGE$ = "MOTOR"
```

The following are all valid PRINT statements:

```
 5  OPEN CHANNEL FOR OUTPUT AS FILE #6 , TYPE=(S)
10  PRINT #6:FULL=30, STRING$
20 PRINT SPEED%,ESTOP_RELAY@,5,7,19*B%
30 PRINT "THIS IS THE REFERENCE",REFERENCE%
40 PRINT ((SPEED% + GAIN%)/14),B%,C%,GAIN OVR@
50 PRINT #2,A$ +"STUFF"+LEFT$(A$,4)
```

Note that when a boolean variable is printed, it is displayed as "TRUE" or "FALSE".

When printing items in BASIC, the individual fields can be separated in one of two ways: with a comma or a semicolon. If the fields are separated by a comma, the items will be printed right-justified in "print zones" of 15 character positions wide. The following is an example:

```
10 PRINT A%,C!,B@
```

This prints as follows:

```
            298           12376           FALSE
123456789012345 123456789012345 123456789012345*
```

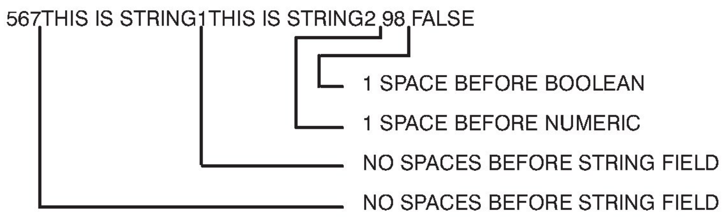
* Used in this example only to show print zones.

The only exception to this is when a string that is greater than 15 characters is printed. The print zone for that string will be the next integral multiple of 15 greater than the string length. If the string is 1 to 15 characters, the print zone is 15. If the length is 16 to 30 characters, the print zone is 30 characters, and so on.

If the items in the print list are separated by a semicolon, the print fields are separated by a single space unless they are string fields. If they are string fields, there are no spaces between them:

```
10 PRINT ABC%;STRING_1$;STRING_2$,XYZ%;BOOL@
```

This prints as follows:



```
567THIS IS STRING1THIS IS STRING2 98 FALSE
```

The semicolon can be used at the beginning of a statement to prevent zoning of the first item printed. The semicolon can also be used at the end of a PRINT statement to tell BASIC not to advance the print pointer to the next line after the print occurs but to leave it at the end of the line. This is useful when an INPUT statement immediately follows the PRINT statement.

In Enhanced BASIC, all decimal numbers have 8 digits of significance. Therefore, only 8 digits can be printed for the number, whether it is very small or very large. To print very large numbers or very small numbers, a scientific or exponential format must be used.

For example, the numbers 1.7634736E+17 and 2.8876582E−09 only have 8 digits of precision printed but use scientific notation to show the number of decimal places to the right or left where the real decimal point would be placed.

In BASIC, unformatted (using the PRINT statement instead of the PRINT USING statement) decimal numbers are printed according to the following rules:

1. BASIC will attempt to locate the decimal point such that there is no need for an exponent.

   Example: 123.45678 or 0.12345678 or 1234567.8

2. If this is not possible, the number will arbitrarily be printed with one number to the left of a decimal point and an exponent or scientific notation will be used.

   Example: 3.7836524E+12 or 4.8873633E−17

3. If the number is a true fraction and requires no exponent, there will always be a leading zero in front of the decimal point.

   Example: 0.98272635 or 0.18735354 or −0.87725433

PRINT USING allows you to print numeric fields with a specific number of decimal places and field width:

    PRINT USING #logical_device_number,formatted_ print_ list

where:

logical_device_number =
: the logical number assigned to a device(PRINT USING to a channel is not permitted) during an OPEN statement. If no device number is given, default device is PORTA on the Processor module on which the task resides. Most application task I/O can be handled through the default port.

formatted_print_list =
list of formatted data fields

The individual formatted data fields have the following form:

<format_type> <field_width> : <variable>

where:

<format_type> =

L	Left-Justify the print expression
R	Right-Justify the print expression
C	Center the print expression in the field
Z	Load leading zeros in front of the print expression
D	Print the numeric field in a decimal format

<field_width> =

The field width has only one part if the format type is L/R/C/Z, in which case the field width is an integer or integer expression from 1 to 132.

If the decimal format D is used, the field width has 2 parts: <integer1>.<integer2>

Integer1 is the total field width for the result. This includes the minus sign (if any), the number part to the left of the decimal point, the decimal point, and any number on the right of the decimal point. This total field may be in the range 3 to 132.

Integer2 specifies the number of decimal places to the right of the decimal point to be printed. This number may be in the range 0 to 26.

Integer1 and integer2 may be either integer literals or integer expressions. If expressions are used, they must be enclosed in parentheses.

<variable> =

Data item to be printed. Multiple items are separated by commas or semi-colons. See the PRINT statement for more information on commas and semi-colons when printing.

The following are PRINT USING examples using the L/R/C/Z FORMATS:

```
 2   A% = 14 \ B% = 26 \ BOOLEAN@ = TRUE
 5   STRING$ = “CHARACTERS”
10   PRINT USING L40:STRING$
20   PRINT USING R40:STRING$
30   PRINT USING C(A% + B%):STRING$
40   PRINT USING Z40:STRING$
50   PRINT USING L40:BOOLEAN@
```

Statement 10 will left justify “CHARACTERS” in a 40-character field beginning with column 1.

The next print expression would start in column 41:

```
column 1                                column 41
CHARACTERS

              40–CHARACTER FIELD
```

Print statements 20,30,40, and 50 would appear as follows:

```
                                  CHARACTERS

                  CHARACTERS

000000000000000000000000000000CHARACTERS
TRUE.
```

The L/R/C/Z formats are used mainly for string fields. These formats can, however, be used to print any data type. If these formats are used to print a numeric field, the resulting number will be in the same decimal format as described for the unformatted PRINT:

- BASIC will use the 8 digits of significance available and attempt to locate the decimal point such that there is no need for an exponent (123.45678 or .12345678 or 1234567.8).
- If this is not possible, the number will be given one numeric position to the left of a decimal point and an exponent will be used (3.7836524E+12).

In either case, the number will be treated just like a “string” and will be right or left justified using the same rules as a string would be when using the L/R/C/Z formats.

Only numeric expressions can use the decimal (D) format of PRINT USING. This includes integers or floating point numbers (decimal numbers). The following are some examples using the D format:

```
5 BIG_NUMBER =21.7654E+10
7 FIELD_WIDTH = 40
10 PRINT USING D(FIELD_WITH+5) .2:BIG_NUMBER
15 SPEED = 2.887654
20 PRINT USING D25.6:SPEED
25 REFERENCE% = 124
30 PRINT USING D10.1:REFERENCE%
```

```
column 1        column 26       column 46
↓                        ↓                   ↓
123456789012345678901234567890123456789012345
   124.0
Statement 30    2.887654
            Statement 20        21765400
0000.00
                         Statement 10
```

Note that, when the integer REFERENCE% was printed with a format of D10.1, the numbers after the decimal were added and set to zero even though REFERENCE% as an integer has no fractional part. This will occur with all integers when the number of decimal places to the right of the decimal point is specified as greater than zero.

The following are all valid PRINT USING statements:

```
    20    PRINT USING L24:“HI”,R34:“THERE”
    20    PRINT USING L40:“HI” + “THERE”
    20    PRINT USING D80.0:CURRENT_REFERENCE
    20    PRINT USING &
          D(LINEWIDTH%−16).(DECPTS%):SPEED
    20    PRINT USING L42:INTEGER%,SPEED
```

In the last example, an unformatted print field follows a formatted one of L42:INTEGER%, which is legal. Unformatted print expressions may be mixed with formatted expressions in PRINT USING. However, an unformatted PRINT statement may not contain formatted print fields. If an unformatted print expression is used in a PRINT USING statement, it will follow the same rules for “zoning” as the PRINT statement.

Commas or semicolons may be used to separate fields in a PRINT USING statement the same way as in the PRINT statement. If formatted fields are used back-to-back, they are not zoned but get their format information from the format type they are printing (L/R/C/Z/D/). If a non-formatted field follows a formatted field and they are separated by a comma, the non-formatted field will follow the same rules as the normal PRINT statement: it will zone the unformatted value.

6.8.5 IOWRITE Statement (Accessing Foreign I/O)

See J-3649, J-3750 or J2-3045 for the requirements for using foreign modules in an AutoMax system. Once it is determined that all the requirements have been met, you can use the IOWRITE statement to write to the foreign modules. The IOWRITE statement can also be used to write to AutoMax Modules that have not been configured.

The base address of each slot in an AutoMax chassis is:

slot #0	-Address Range 200000 -	20FFFF(Hex)
slot #1	-Address Range 210000 -	21FFFF(Hex)
slot #2	-Address Range 220000 -	22FFFF(Hex)
slot #3	-Address Range 230000 -	23FFFF(Hex)
slot #4	-Address Range 240000 -	24FFFF(Hex)
slot #5	-Address Range 250000 -	25FFFF(Hex)
slot #6	-Address Range 260000 -	26FFFF(Hex)
slot #7	-Address Range 270000 -	27FFFF(Hex)
slot #8	-Address Range 280000 -	28FFFF(Hex)
slot #9	-Address Range 290000 -	29FFFF(Hex)
slot #10	-Address Range 2A0000 -	2AFFFF(Hex)
slot #11	-Address Range 2B0000 -	2BFFFF(Hex)
slot #12	-Address Range 2C0000 -	2CFFFF(Hex)
slot #13	-Address Range 2D0000 -	2DFFFF(Hex)
slot #14	-Address Range 2E0000 -	2EFFFF(Hex)
slot #15	-Address Range 2F0000 -	2FFFFF(Hex)

Each slot in the rack has 64 K of address space.

Attempting to access memory on AutoMax Processors is not permitted. The following is the format of the IOWRITE statement:

```
IOWRITE(option, data, address)
```

where:

option = the kind of write to take place; literal or expression:

1 = single byte write (low byte of data is written to the address)

2 = double byte write (address must be even)(MSB is written to the address) (LSB is written to the address + 1) This option is used to write to foreign I/O modules that only support an 8-bit data path.

3 = word write (address) = A 16-bit word is written to the designated address. This option writes data to modules that support AutoMax addressing and data conventions.

4 = long word write (address) = MSB
(address + 1) = next 8 bits
(address + 2) = next 8 bits
(address + 3) = LSB

data = integer variable name or expression defining data to output; literal or expression

address = double integer variable name or expression defining the destination address; literal or expression; must be $\geq$ 220000H

Note that all task-to-task communication information is managed by the system on the Common Memory module (M/N 57C413).

6.8.6 GET Statement

The GET statement is used to input a single character from a device (not a channel). The GET statement has the following format:

GET #logical_device_number, string_variable

where:

logical_device_number =
: the logical number assigned to a device during an OPEN statement. If no device number given, the default device is PORTA on the Processor module on which the task resides.

string_variable =
: a variable of data type string only

The GET statement reads a single character from a device and loads the character into a string variable. The character is NOT echoed to the device as it is with an INPUT operation.

The optional parameter :EMPTY=n can be used to transfer control of the program in the event that the channel is empty. The parameter is added immediately after the device number; n is the line number to which to transfer control.

The following are valid GET statements:

```
20  GET #2,A$
30  GET A$
40  GET #4:EMPTY=50, CHAR$
```

Refer to the OPEN statement description (6.8.1) for more information on how the GET statement is used with the OPEN statement.

6.8.7 PUT Statement

The PUT statement is used to output a single character from a device (not a channel). The PUT statement has the following format:

PUT #logical_device_number, string_variable

where:

logical_device_number =
: the logical number assigned to a device during an OPEN statement. If no device number is given, the default device is PORTA on the Processor module on which the task resides.

string variable =
: variable of data type string

The PUT statement outputs a single character from a string variable to a device. The operation does not generate a <CR> <LF> as a standard PRINT operation does.

The following are valid PUT statements:

```
20 PUT #2,A$
30 PUT A$
```

Refer to the OPEN statement description (6.8.1) for more information on how the GET statement is used with the OPEN statement.

6.8.8 READ Statement

The READ statement directs the system to read from a list of values built in a data block by a DATA statement. A READ statement is not legal without at least one DATA statement.

A READ statement causes the variables listed in it to be given the value of the next expression in the next DATA statement. BASIC has a pointer to keep track of the data being read by the READ statement. Each time the READ statement requests data, BASIC retrieves the next expression indicated by the data pointer.

The READ statement has the following format:

```
READ variable_1,variable_2,...,variable_N
```

where:

variable_1 through variable_N = the value(s) listed in the DATA statements

The variables can be simple (A%) or subscripted (A%(4)) and can be any of the five variable types: integer, double integer, real, string, and boolean. All variables should be separated by commas. String variables should be enclosed within single or double quotes.

The following is a valid READ statement:

```
10 READ A%,B%,C$,D%(5)
```

6.8.9 DATA Statement

The DATA statement has the following format:

```
DATA expression_1, expression_2,... ,expression_n
```

where:

expression_1 to expression_n =
expression that, after evaluation, is loaded into the corresponding variable in a READ statement when the READ is executed

The data type of the expression in the DATA statement must be the same as the data type of the variable that corresponds to it in the READ statement.

The program will run faster with READ and DATA statements as opposed to the INPUT statement simply because the system does not have to wait the extra time it takes for the system to stop and request data. The data is already within the program.

The DATA statements may be formatted with any number of expressions as long as the same number is being requested by the READ statements:

```
10 READ A,B,C,D,E,F
20 ...
30 ...
40 ...
800 DATA 17
900 DATA 25,30,43,76,29
```

The number of variables in the READ statement does not have to match the number of expressions in a specific DATA statement. BASIC will simply go to the next sequential DATA statement in the program to acquire the value. The variable type and expression must match.

A READ statement is not legal without at least one DATA statement. However, you can have more than one DATA statement as long as there is one READ statement in the program:

```
10 READ A,B,C,D,E,F
20...
30...
40...
50 DATA 17,25,30
60 DATA 43,76,29
```

A READ statement can be placed anywhere in a multi-statement line. A DATA statement, however, must be the only statement on a line.

If you build your READ statement with more variables than you include in the data block, the system will print an error message when it attempts to access the next DATA expression and finds one is not there.

The following is an example of a READ and DATA sequence:

```
10    READ A%,B,C1!,D2%,E4@,E6$,Z$
20    DATA 2,32.9987,(83+19),–6, TRUE, “CAT”,’DOG’
30    PRINT A%,B,C1!,D2%,E4@,E6$,Z$
```

In the example above, BASIC assigns these values:

```
A% = 2
B = 32.9987
C1! =102
D2% =–6
E4@ = TRUE
E6$ = CAT
Z$ = DOG
```

READ and DATA are useful to initialize the values of variables and arrays at the beginning of a program. To do this, place READ statements at the beginning of the program. You can put the DATA statements anywhere in the program. It is often useful to put them all at the end of the program just before the END statement. See also the RESTORE statement.

6.8.10 RESTORE Statement

The RESTORE statement is used when reading from a DATA statement more than once. When you READ data from a series of DATA statements, BASIC advances an internal pointer to the next item in the data list every time a READ is done. Once you have read all the items in the data list, this internal pointer points to the end of the data list. If you want to read starting with the first DATA statement, the RESTORE statement tells BASIC to reset its pointer to the beginning of the DATA statements. The RESTORE statement has the format:

```
RESTORE
```

or

```
RESTORE line_number(expression)
```

The effect of the first format (with no line number) is to move the DATA statement pointer back to the first DATA statement in the program. The effect of the second format (with the line number) is to reset the DATA statement internal pointer to the DATA statement at the line number specified following the RESTORE. This line number may be specified either by an integer constant or integer expression. There must be a DATA statement at the line number that follows the RESTORE or the system will generate a STOP ALL error. A RESTORE can be used at any time, not only when all the DATA statements have been read or at the end of the data.

6.9 Error Handling

During the execution of a BASIC task, error conditions can occur that are not severe enough to stop the task but are worth noting. All errors that happen during execution are logged in the task error log, accessible through the on-line menu of the Programming Executive software. If the error is severe, it is displayed on the two 7-segment LEDs on the Processor module (M/N 57C430, 57C430A, 57C431, and 57C435) and all tasks are stopped. See J-3684, J-3750 or J2-3045 for more information.

BASIC provides two statements to help deal with errors that occur during execution.

1. ON ERROR statement
2. RESUME statement

6.9.1 ON ERROR Statement

The ON ERROR statement is used to define where the task should transfer control if a non-fatal error occurs. The ON ERROR statement has the following format:

```
ON ERROR GOTO line_number
```

where:

line_number =	line_number where error handling routine begins.

When BASIC executes the ON ERROR statement, it stores the line number referenced for later reference. When an error occurs, BASIC transfers control of the program immediately to that line number. The ON ERROR statement may be executed as many times as desired. BASIC re-loads the error handler line number each time.

To tell the user what kind of error occurred and where it took place. BASIC provides two pre-defined symbols:

- ERR% - The error number of the logged error (decimal error number)
- ERL% - The line number where the error occurred

These symbols are automatically defined when the task is created and can be accessed the same as any other variable. Refer to Appendix B for a complete listing of run-time error codes.

6.9.2 RESUME Statement

After BASIC has transferred control to an error handling routine, RESUME tells BASIC that the error handling is complete. The RESUME statement has the following format:

```
RESUME
```

The RESUME statement returns processing to the statement that was in progress when the error condition diverted it to the error handling routine.

The following is a program with a valid RESUME statement:

```
10 ON ERROR GOTO 850
      .
      .
      .
850!--------Run time error handler-------
860!
870 IF ERR% = INT_VAR_OVRFLOW% AND  &
      ERL% = 720 THEN GAIN = 0
880...
890...
895...
900 RESUME
```

6.9.3 CLR_ERRLOG

The statement CLR_ERRLOG is used to clear the error log for the application task, regardless of the number of logged errors. See 7.35 for more information about testing the error log. The format of the statement is:

```
CLR_ERRLOG.
```

6.10 INCLUDE Statement

The INCLUDE statement allows the programmer to include a file containing BASIC statements in the task as it is compiling. Multi-statement lines and multi-line statements are permitted. The file must not include line numbers. The compiler will add line numbers in increments of 1, beginning with the INCLUDE statement line number. The programmer must allow enough line numbers between the INCLUDE statement and the statement that follows it to accommodate the lines automatically generated. There are no limits on the number of INCLUDE statements in a task. However, no INCLUDE statements are permitted in a file that will be included in the task. The format of the INCLUDE statement is:

INCLUDE “filename.INC”

where:

filename =	name of the file containing the statements to be included. The extension .INC is required. No drive or subdirectory specification is permitted. The file to be included must be located in the drive and subdirectory in which the task is located.

When you save a reconstructible task from the processor, the system will write the lines included back to a file with the same filename as specified in the INCLUDE statement. The reconstructed file will look exactly like the source file. See J-3684, J-3750 or J2-3045 for more information on reconstructible tasks.

The following is an example of a valid INCLUDE statement:

```
50 INCLUDE "IODEFS.INC"
```

The file IODEFS.INC contains the following:

```
IODEF  RELAY_1@  [SLOT=3,REGISTER=1,BIT=4]
IODEF  RELAY_2@  [SLOT=3,REGISTER=1,BIT=5]
IODEF  RELAY_3@  [SLOT=3,REGISTER=1,BIT=6]
```

When the file is compiled, it will look like this to the compiler:

```
50  INCLUDE "IODEFS.INC"
51  IODEF  RELAY_1@  [SLOT=3,REGISTER=1,BIT=4]
52  IODEF  RELAY_2@  [SLOT=3,REGISTER=1,BIT=5]
53  IODEF  RELAY_3@  [SLOT=3,REGISTER=1,BIT=6]
```

6.11 Stopping Execution (STOP and END Statements)

WARNING

CAREFULLY REVIEW MACHINE OPERATION TO INSURE THAT UNSAFE MOVEMENT IS NOT INITIATED BY STOPPING ALL APPLICATION SOFTWARE. FAILURE TO OBSERVE THIS PRECAUTION COULD RESULT IN BODILY INJURY OR DAMAGE TO EQUIPMENT.

The STOP statement will stop all tasks in the system and should only be used when a severe error has occurred. It will not be possible to continue from a "stopped" state without re-starting all tasks in the rack. The STOP statement has the following format:

```
STOP
```

The following are valid STOP statements:

```
20   IF OVER_TEMP@ THEN STOP
20   IF GAIN%>MAXGAIN% THEN PRINT          &
     ERROR_MESSAGE$\STOP
```

The END statement is used to end the task execution or to place the task in a suspended state until the time interval that was programmed in a START EVERY statement expires. If a task has a periodic execution defined, it will at some later point be re-activated when that period or interval expires. Refer to section 6.7 for more information.

The END statement has the format:

```
END
```

The END statement must be at the physical end of the task.

7.0 FUNCTIONS

AutoMax Enhanced BASIC incorporates numerous intrinsic functions, i.e., functions that can be used within expressions. Some are standard and some have been added to complement the AutoMax environment. The following is a list of the current functions available for use in an AutoMax system:

Function Name	Description
SIN	Sine
COS	Cosine
TAN	Tangent
ATN (ATAN)	Arctangent
LN	Natural logarithm (log base e)
EXP	Exponential (e**x)
SQRT	Square root
ABS	Absolute value
CHR$	Get character from ASCII value
ASC%	ASCII value of string character
LEN%	Lenth of string
STR$	String from numeric expression (integer)
BINARY$	Binary form of input string (integer)
HEX$	Hexadecimal value of input string (integer)
LEFT$	Substring from left side of string
RIGHT$	Substring from right side of string
MID$	Substring from center of string
VAL%	Integer value of integer string expression
VAL	Real value of real string expression
FIX	Whole part only of real
CURPOS$	Position cursor (VT100)
CLRSCR$	Clear the screen (VT100)
CLRLINE$	Clear line (VT100)
IOREAD%	Read from foreign I/O board
BIT_SET@	Test if bit is set
BIT CLR@	Test if bit is clear
BIT_MODIFY@	Modify bit value
SHIFTL%	Shift a numeric field left
SHIFTR%	Shift a numeric field right
ROTATEL%	Rotate a numeric field left
ROTATER%	Rotate a numeric field right
BCD_OUT%	Output the BCD number from a decimal number
BCD_IN	Input the decimal number from a BCD number
BLOCK_MOVE@	Move a block of integers from/to registers
GATEWAY_CMD_OK@	Gateway transfer function
VARPTR%	Return pointer to variable
TEST_ERRLOG@	Test state of error log for task
READVAR%	Reads variable, returns value
WRITEVAR%	Writes value to variable
FINDVAR!	Returns pointer to variable
CONVERT%	Converts data formats

7.1 SIN Function

Format:

SIN(expression)

where:

expression must be of numeric (integer or real) type. It represents radians.

The function returns a real value equal to the sine of the input. The result is in real format.

7.2 COS Function

format:

COS(expression)

where:

expression must be of numeric (integer or real) type. It represents radians.

The function returns a real value equal to the cosine of the input. The result is in real format.

7.3 TAN Function

Format:

TAN(expression)

where:

expression must be of numeric (integer or real) type. It represents radians.

The function returns a real value equal to the tangent of the input. The result is in real format.

7.4 ATN (ATAN) Function

Format:

ATN(expression)

or

ATAN(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a radian value equal to the arctangent of the input. The units are in radians.

7.5 LN Function

Format:

LN(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a real value equal to the natural log of the input. The result is in real format.

7.6 EXP (e**x) Function

Format:

EXP(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a real value equal to (e**expression) where e is 2.71828. The result is in real format.

7.7 SQRT Function

Format:

SQRT(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a real value equal to the square root of the input and the same data type as the input.

7.8 ABS Function

Format:

ABS(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a real or integer value equal to the absolute value of the input and the same data type as the input.

7.9 CHR$ Function

Format:

CHR$(expression)

where:

expression must be of integer type (hexadecimal is also considered integer).

The function returns a string character, corresponding to the decimal ASCII value of the input expression:

```
STRINGS$ = CHR$(41H)          (STRING$ is loaded with 'A')
STRINGS$ = CHR$(30H)          (STRING$ is loaded with '0')
STRINGS$ = CHR$(2AH)          (STRING$ is loaded with '*')
```

7.10 ASC% Function

Format:

ASC%(string)

where:

string can be a string variable or string expression.

The function returns the ASCII value of a single string character (the opposite of the CHR$ function):

```
STRING$ = '5'

NUMBER% = ASC%(STRING$) (NUMBER% has the
                         value 35H)
```

7.11 LEN% Function

Format:

LEN%(string expression)

where:

expression must be of string type.

The function returns the length of the string expression:

```
STR_LEN% = LEN%('STRING') (STR_LEN% is loaded
                           with 6)

STR1$ = 'AVCDFG'

STR_LEN% = LEN%(STR1$ + "23") (STR LEN% is
                               loaded with 8)
```

7.12 STR$ Function

Format:

STR$(expression)

where:

expression must be of numeric (integer or real) type.

The function returns a string of characters from a numeric expression:

```
STRING$ = STR$(1224)          (STRING$ is loaded with 1224)

NUM1% = 32

STRING$ = STR$(NUM1%*3) (STRING$ is loaded with 96)

STRING$ = STR$(388.73612) (STRING$ is loaded with
                           +.38873612E03)
```

In the last example, the format of the real number returned as a string is the same number but in a different representation. All real numbers will be returned by the STR$ function in this format.

The string will always be 14 characters long with a sign (+ or −), a decimal point (.), 8 digits (12345678), an exponent sign (E), and 2 digits of exponent (XX). The sign character is always present after this conversion even if the number is positive.

7.13 BINARY$ Function

Format:

BINARY$(expression)

where:

expression must be an integer or integer expression.

The function returns the binary form of the input as a string of 1s and 0s. For example:

```
BIT$=BINARY$(NUMBER_1%)
```

7.14 HEX$ Function

Format:

HEX$(expression)

where:

expression must be an integer or integer expression.

The function returns the hexadecimal value of the input as a string. For example:

```
HEX_VAL$=HEX$(X%)
```

7.15 LEFT$ Function

Format:

LEFT$(string,str_length)

where:

string can be a string variable or expression.

str_length is the number of characters to take from the left side of the string (string expression) in parameter 1.

The function returns a substring that is equal to ‘str_length’ (parameter 2) of the leftmost characters of the string or string expression (parameter 1):

SUB_STRING$ = LEFT$(‘ABCDEFG’,4) *(SUB_STRING$ has the value ABCD)*

STR1$ = ‘12345’

SUB_STRING$ = LEFT$(STR1$ + ‘ABC’,6) *(SUB_STRING$ has the value 12345A)*

7.16 RIGHT$ Function

Format:

RIGHT$(string,str_length)

where:

string can be a string variable or expression.

str_length is the number of characters to take from the right side of the string (expression) in parameter 1.

The function returns a substring that is equal to 'str-length'(parameter 2) of the right-most characters of the string or string expression (parameter 1):

SUB_STRING$ = RIGHT$('ABCDEFG',4) *(SUB_STRING* has the value DEFG)

STR1$ = '12345'

SUB_STRINGS$ = RIGHT$(STR1$ + 'ABC',6)
(SUB_STRING$ has the value 345ABC)

7.17 MID$ Function

Format:

MID$(string, start, end)

where:

string can be a string variable or expression.

start is the starting character position of the substring.

end is the ending character position of the substring.

The function returns a substring from within another string. The three parameters are, respectively, the string to operate on, the starting character position, and the ending character position. The substring is then taken from those two inclusive positions:

SUB_STRING$ = MID$('ABCDEFG',2,3) *(SUB_STRING$ has the value BC)*

STRING$ = "BIGLONGSTRING"

SUB_STRING$ = MID$(STRING$,3,7) *(SUB_STRING$ has the value GLONG)*

7.18 VAL% Function

Format:

VAL%(string)

where:

string can be a string variable or expression.

The function returns the integer value of a string in an integer format. If the string is not in an integer format, the returned value will be zero and an error is logged:

STR_VAL% = VAL%('123' + '74') *(STR_VAL% has the value 12374)*

7.19 VAL Function

Format:

VAL(string)

where:

string can be a string variable or expression.

The function returns the real value of a string in a real format. If the string is not in a real format, the returned value will be zero and an error is logged:

STR_VAL = VAL('9.8827') *(STR_VAL has the value 9.8827)*

7.20 FIX Function

Format:

FIX(expression)

where:

expression can be a real variable or real expression.

The function returns the whole part of a real or decimal number:

REAL_VAL = 87.88763

WHOLE PART = FIX(REAL_VAL) *(WHOLE PART HAS THE VALUE 87.00)*

7.21 CURPOS$ Function

Format:

CURPOS$(row,column)

where:

row and column are integer variables or expressions that represent a cursor location on the screen of a VT100 compatible terminal (1 to 24 rows inclusive and 1 to 132 columns inclusive).

CURPOS$ returns an ASCII string or escape sequence. This function is used in combination with a PRINT statement to position the cursor at a specific location on the screen (specified by row and column) for a device that recognizes the ANSI standard cursor position escape sequence (ESC [row; col H):

ROW% = 10

COL% = 4

HEADING$ = "TABLE #1"

PRINT;CURPOS$(ROW%,COL%) ;HEADING$ *(Prints "TABLE #1" at cursor position 10,4)*

When using the CURPOS$ function with the PRINT statement, a semicolon should always be located in front of the function call to tell BASIC not to put the string (generated by the function call) in a zoned field (a field that is a multiple of 15 characters).

If the semicolon is not used, BASIC will pad the front of the escape sequence with spaces, which will most likely not have the desired effect: to move the cursor rather than to print something. Further information about printing and zoned fields can be found in PRINT/PRINT USING Statements in section 6.8.

7.22 CLRSCR$ Function

Format:

CLRSCR$(option)

where:

option is an integer variable or integer expression that defines the clear screen option to perform.

CLRSCR$ returns an ASCII string or escape sequence. This escape sequence will clear various parts of a screen on a terminal that recognizes the ANSI standard erase-in-display escape sequence (ESC [option J]). The option values are as follows:

0 = Erase from the active cursor position to the end of the screen, inclusive.

1 = Erase from the start of the screen to the active cursor position, inclusive.

2 = Erase all of the display.

For example,

```
CLR_SCREEN% = 2
PRINT; CLRSCR$(CLR_SCREEN%);  (Clears entire
                                display screen)
```

When using the CLRSCR$ function with the PRINT statement, a semicolon should always be located in front of the function call to tell BASIC not to put the string (generated by the function call) in a zoned field (a field that is a multiple of 15 characters).

If the semicolon is not used, BASIC will pad the front of the escape sequence with spaces, which will most likely not have the desired effect: to move the cursor rather than to print something. Further information about printing and zoned fields can be found in PRINT/PRINT USING statements in section 6.8.

7.23 CLRLINE$ Function

Format:

CLRLINE$(option)

where:

option is an integer variable or integer expression that defines the clear line option to perform.

CLRLINE$ returns an ASCII string or escape sequence.
This escape sequence will clear various parts of a display line on a terminal that recognizes the ANSI standard erase-in-line escape sequence. The options are as follows:

ESC [0K Erase from the active cursor position to the end of the line, inclusive.

ESC [1KErase from the start of the line to the active cursor position, inclusive.

ESC [2K Erase all of the line, inclusive.

CLR_TO_ENDLINE% = 0

PRINT;CLRLINE$(CLR_TO_ENDLINE%); *(Clears from cursor to end of display line)*

When using the CLRLINE$ function with the PRINT statement, a semicolon should always be located in front of the function call to tell BASIC not to put the string (generated by the function call) in a zoned field (a field that is a multiple of 15 characters).

If the semicolon is not used, BASIC will pad the front of the escape sequence with spaces, which will most likely not have the desired effect: to move the cursor rather than to print something. Further information about printing and zoned fields can be found in PRINT/PRINT USING statements in section 6.8.

7.24 IOREAD% Function

The IOREAD% function is used to access I/O from foreign modules that are byte accessible only. The function returns an integer value.

Format:

IOREAD%(option,address)

where:

option is an integer variable or expression that defines the type of read operation to perform:

option 1 =	byte read
option 2 =	double byte read (address must be even) (address) = MSB (address + 1) = LSB This option is used to read from foreign I/O modules that only support an 8-bit data path.
option 3 =	word read (address) = MSB (address + 1) = LSB The 16-bit word is read from the designated address. This option reads data from modules that support AutoMax addressing and data conventions.
option 4 =	long word read (address) = MSB (address + 1) = next 8 bits (address + 2) = next 8 bits (address + 3) = LSB

address is a double integer variable or expression that contains the address from where data is to be read; the address must be $\geq$220000H.

See J-3649, J-3750 or J2-3045 for more information on accessing foreign I/O.

7.25 BIT_SET@ Function

Format:

BIT_SET@(variable,bit-number)

where:

variable is a single or double integer variable

bit-number is the bit number within the variable to test (0 to 15 for single integer; 0 to 31 for double integer).

This function tests the value of a bit within the variable as specified by the bit-number. If the bit is set to a value of one (1), the boolean result of the function is TRUE. If the value of the bit is zero (0), the result of the function is FALSE. This function does not change the state of a bit; it simply tests it.

BIT_VALS% = 012A2H:! *(BIT_VALS% = 0001 0010 1010 0010)*

IF BIT_SET@(BIT_VALS%,1) THEN 250

In this example, the 'IF' statement will take the branch to line 250 because the condition is TRUE. The value of bit 1 of BIT_VALS% is TRUE or 1.

7.26 BIT_CLR@ Function

Format:

BIT_CLR@(variable, bit_number)

where:

variable is a single or double integer variable.

bit_number is the bit number within the variable to test (0 to 15 for single integer; 0 to 31 for double integer).

This function tests the value of a bit within the variable as specified by the bit-number. If the bit is clear or a value of zero (0), the boolean result of the function is TRUE. If the value of the bit is one (1), the result of the function is FALSE. This function does not change the state of a bit; it simply tests it.

BIT_VALS% = 1128H:! *(BIT_VALS% = 0001 0001 0010 1000)*

IF BIT_CLR@(BIT_VALS%,2) THEN 250

In this example, the 'IF' statement will take the branch to line 250 because the condition of the bit test was TRUE. (The bit was clear or equal to zero.) The value of bit 2 of BIT_VALS% is FALSE; however, the entire function is TRUE if the value of the bit is zero, so the function is TRUE.

7.27 BIT_MODIFY@ Function

Format:

BIT_ MODIFY@(variable, bit_number,option)

where:

variable is a single or double integer variable.

bit_number is the bit number within the variable to test.
(0 to 15 if single integer; 0 to 31 if double integer variable).

option defines the change to be made to the bit and may be an integer or boolean expression.

This function tests the variable as specified by the bit_number. Based on that value, the bit within the variable will be modified:

0 = Unconditionally clear the bit (set to 0)
1 = Unconditionally set the bit to 1
2 = If the bit is already zero, set it to 1
3 = If the bit is 1, clear the bit (set to 0).

The function itself is a boolean function and not an integer function. Therefore, the value that it returns is not the updated value of the variable modified but the status of the bit change operation. The return status of the function is TRUE if the requested bit change operation was completed or FALSE if the requested bit change operation was not completed.

Two conditions will stop a request from being completed. The first is that the variable is in a forced condition and the bits cannot be modified. The second is that the requested change is already the current state of the bit.

If the request is to clear a bit if it is set and the current state of the bit is already cleared, the function status will be FALSE because the operation was not completed, i.e., there was no need to make the change. This tells the user the state of the bit prior to the bit modify operation. Unless the variable is forced, options 0 and 1 will always return a TRUE function status since they are unconditional operations.

Boolean values can be used as the option number for the function. This will result in an option 0 if the boolean is FALSE or in an option 1 if the boolean is TRUE. This can be used to change the state of a bit to match another boolean.

For example:

```
IF NO_FAULTS@ THEN SYSTEM_RUNNING@ = TRUE

BIT_VALS% = 0005H

    .
    .
    .

IF BIT_MODIFY@(BIT_VALS%,2,SYSTEM_RUNNING@) THEN
300
```

In this example, bit 2 of BIT_VALS% will be given the same value as the boolean SYSTEM_RUNNING@.

7.28 SHIFTL% Function

Format:

```
SHIFTL%(variable, shift_count)
```

where:

variable is a single or double integer variable.

shift_count is the number of bit positions to shift the integer expression (0 to 15 for single integer; 0 to 31 for double integer).

This function returns an integer value, equal to the integer expression that was input, shifted the specific number of binary places to the left. (Bit 15, or bit 31 if double integer, comes off the end). The binary places vacated by the shift are filled with zeros.

For example:

```
LSBITS_0_4% = SHIFTL%(INPUT_CARD%,12) AND 0F000H
```

In this example, a 16-bit value is read from an input module. The least significant group of 4 bits (bits 0 to 3) are shifted to the left into the most significant position. The remaining lower bits are masked off by “anding” with hex value 0F000.

7.29 SHIFTR% Function

Format:

```
SHIFTR%(variable,shift_count)
```

where:

variable is a single or double integer variable.

shift_count is the number of bit positions to shift the integer expression (0 to 15 for single integer; 0 to 31 for double integer).

This function returns an integer value equal to the integer expression that was input, shifted the specific number of binary places to the right. Bit 0 comes off the end. The result is not sign extended but treated as a logical shift.

For example:

```
DECADE_2% = SHIFTR%(INPUT_CARD%,4) AND 0FH
```

In this example, a 16-bit value is read from an input module. The second group of 4 bits (bits 4 to 7) are shifted to the right into the least significant position. The remaining upper bits are masked off by “anding” with hex value OF.

7.30 ROTATEL% Function

Format:

```
ROTATEL%(variable, rotate_count)
```

where:

variable is a single or double integer variable or expression.

rotate_count is the number of bit positions to rotate the integer expression (0 to 15 for single integer; 0 to 31 for double integer).

This function returns an integer value, equal to the integer expression that was input, rotated the specific number of binary places to the left (Bit 15, or bit 31 if double integer, wraps around to bit 0).

For example:

```
MOVE_3_4% = ROTATEL%(INPUT_CARD%,4)
```

In this example, a 16-bit value is read from an input module and bits (0 to 11) are rotated into the most significant bit positions (4 to 15). Bits (12 to 15) are rotated into bit positions (0 to 3).

7.31 ROTATER% Function

Format:

ROTATER%(variable, rotate_count)

where:

variable is a single or double integer variable or expression.

rotate_count is the number of bit positions to rotate the integer expression (0 to 15 single integer; 0 to 31 for double integer).

This function returns an integer value, equal to the integer expression that was input, rotated the specific number of binary places to the right (Bit 0 wraps around to bit 15, or bit 31 if double integer).

For example:

```
SWAP_4_1% = ROTATER%(INPUT_CARD%,4)
```

In this example, a 16-bit value is read from an input module and bits (4 to 15) are rotated into bit positions (0 to 11). Bits (0 to 3) are rotated into bit positions (12 to 15).

7.32 BCD_OUT% Function

Format:

BCD_OUT%(variable)

where:

variable is a single or double integer variable or expression (variable value $\leq$ 9999 decimal).

This function returns an integer value, equal to the integer expression that was input, with each decimal digit converted to a hexadecimal 4-bit binary equivalent (opposite of the BCD IN% function):

```
SWITCH_VALS% = 2156
OUTCARD_CARD% = BCD_OUT%(SWITCH_VALS%)
```

In the above example, the decimal value 2156 is converted to a 16-bit hexadecimal value equal to 2156H (hex).

If the value input to BCD_OUT% is greater than 9999 decimal, the output will be hex 9999H and an error will be logged. The same is true if the input number is negative; the output will be 0000H (hex) and an error will be logged.

7.33 BCD_IN% Function

Format:

BCD_IN%(variable)

where:

variable is a single or double integer variable or expression (variable value $\leq$ 9999 hex).

This function returns an integer value, equal to the integer expression that was input, with each digit converted from a hexadecimal 4-bit binary to the decimal equivalent (opposite of the BCD_OUT% function). In the following example, assume the value at INPUT_CARD% is 2381 hex:

```
IN_VALS%=BCD_IN%(INPUT_CARD%)
```

The hex value 2381 is converted to a 16-bit decimal value equal to 2381.

If the value input to BCD_IN% is greater than 9999 hex, the output will be decimal 9999 and an error will be logged.

7.34 BLOCK_MOVE@ Function

Format:

```
BLOCK_MOVE@(source, dest, size)
```

where:

source and dest must be integer, double integer, boolean, or real variables or one-dimensional integer arrays. If integer, double integer, or boolean variables, they must be variables defined in the configuration task that points to the desired specific source or destination register on the network. If the source or destination is an array, it must be 1- dimensional. The array may be common or local.

size is the number of registers (16-bit words) to transfer from the source to the destination.

The BLOCK_MOVE function transfers data to and from the network. It is a boolean function and is TRUE if the operation is successful, FALSE if not. The function allows the user to specify only a starting variable name for the destination or source variable list and a length of registers to be transferred. This eliminates having to define each variable explicitly in the rack configuration and then transferring the data one variable at a time with a long list of LET statements.

The BLOCK_MOVE function checks the individual register values as they are moved; if any of the values are FORCED, the forced value will be used. Thus, the execution time for the function varies considerably and is dependent on the number of variables in the current force list (maximum of 16) and whether any of those variables are in the source or destination range of the BLOCK_MOVE variables. In the following example, 32 registers are moved:

```
IF NOT BLOCK_MOVE@(NETW_1_32%, NETW_2_32%,32)
THEN STOP
```

Keep in mind the following when forcing variables and overlaying variable definitions (an integer is overlayed over several booleans):

1. If boolean variables are forced within a forced integer, double integer, or real variable, the integer forced value will prevail.

   For example, If A@ and B@ are I/O defined as bits within XYZ%, XYZ% and A@ and B@ are all forced, and XYZ% is used as the source variable in a BLOCK MOVE function; the result will be that the XYZ% forced value will take precedence over the A@ and B@ forced values.

2. If the destination is forced and the source is forced, the destination forced value will prevail. The BLOCK MOVE function will return a FALSE status if any of the following occurs:

The BLOCK_MOVE function will return a FALSE status if any on the following occurs:

1. The number of registers to transfer (transfer size) is:
   - Less than or equal to zero
   - Greater than 32767
2. The source or destination variable address is less than the valid starting offboard address for that rack configuration:
   - If there is no Common Memory module (M/N 57C413), the starting address must be equal to or greater than 200000(hex)
   - If there is a Common Memory module (M/N 57C413), the starting address must be equal to or greater than 220000(hex)
3. The number of registers to transfer is greater than the number of elements in the source or destination array. If the source or destination is an array and the number of elements is greater than the number of registers to move, only the number requested will be moved and the rest of the array will not be affected.

7.35 GATEWAY_CMD_OK@ Function

GATEWAY_CMD_OK@ is a boolean function that performs register transfers to or from the Interface modules, including the Allen-Bradley (M/N 57C418), Modbus (M/N 57C414), and AutoMate (M/N 57C417) modules. This function cannot be used on the AutoMax PC3000. If the operation is successful, the function returns a 0 value. If the operation is not successful, the operation returns an error code that is determined by the hardware with which the Interface module is communicating.

Format:

```
GATEWAY_CMD_OK@(status, cmd_code, slave_drop, &
                 slave_reg, master_var, num_regs)
```

where:

status is an integer variable representing the location where the resulting command status is stored; status will contain a zero if the transfer operation is successful and an error code if it is unsuccessful. The error code is dependent on the module used. See the instruction manual for the specific Interface module for the error codes.

cmd_code is a variable name or expression of type integer representing the command sent to the Interface module; the commands available are specific to each interface module. See the instruction manual for the specific Interface module for the available commands.

slave_drop is a variable name or expression of type integer representing the device address (e.g., node number) of the hardware the Interface module is communicating with slave_reg is a variable name or expression of type string representing the starting register in the device that is to be read from or written to. The format of this parameter is dependent on the module used. See the instruction manual for the specific Interface module for the correct format.

master_var is a variable name or expression of type double integer, representing the actual hexadecimal address of the first location that is to be read from or written to in the Interface module. See below for more information.

num_regs is variable name or expression of type integer that defines the number of bits or registers (16 bits each) to be transferred; cmd_code determines whether the variable represents bits or registers. Note that if you are transferring double integer (32-bit) variables, the num_regs parameter must specify the number of 16-bit variables to move. For example, if you want to move two double integer variables starting at address 270480H, num_regs must be equal to 4.

The master_var parameter can be specified using any one of the three methods described below.

1. The absolute hexadecimal address of the first bit, integer, or double integer on the Interface module to be read or written. For example:

   ```
   270480H
   ```

2. A variable name which represents the absolute hexadecimal address of the first bit, integer, or double integer on the Interface module to be read or written. Typically, the VARPTR! function is used to yield the absolute address. For example, if in your task you define the following:

   ```
   ADDRESS! = VARPTR! (FIRST_REG%)
   ```

   then you can use ADDRESS! as the master_var parameter. This assumes that FIRST_REG% has been defined as a 16-bit integer variable on the Interface module

3. An expression which, when solved, will yield an absolute hexadecimal address of the first bit, integer, or double integer to be read or written. For example, the master_var parameter could be:

   ```
   VARPTR! (FIRST_BIT@)
   ```

   where FIRST_BIT@ has been defined as a boolean on the Interface module.

For more detailed information, see the instruction manual for each specific Interface module.

7.36 VARPTR! Function

Format:

```
VARPTR!(variable)
```

where:

variable is a common boolean (single bit), integer (16 bits), or double integer (32 bits) defined in the configuration as the actual physical address of the first location on the Interface module from or to which data is to be read or written.

This function is usually used in conjunction with the GATEWAY_CMD_ OK@ function. It returns the actual address (in hexadecimal format) of the bit, integer, or double integer which represents the first location on the Interface module to which or from which data is to be transferred.

7.37 TEST_ERRLOG@ Function

The function TEST_ERRLOG@ tests the state of the error log maintained for the application task by the Programming Executive software. This error log can be accessed through the On-Line menu of the software. The function TEST_ERRLOG@ provides a method of checking the error log while the application task is running. This function is cleared with CLR_ERRLOG (see 6.9.3). Note that for Version 3.3A and later of the Programming Executive software, this function can be entered as TEST_ERRLOG@ or TST_ERRLOG@.

Format:

```
TEST_ERRLOG@(variable)
```

where:

variable is an integer variable only; used to store the number of errors logged (1 to 3) for the task

If there are no errors logged, the result of the boolean function is FALSE and the number parameter is indeterminant. If there are errors, then the function is true and the variable specified is loaded with the current number of logged errors (1 to 3). This variable must be an integer variable.

7.38 READVAR% Function

This function requires the AutoMax operating system with the Ethernet option.

Format:

```
READVAR%( vn$, value )
```

where:

vn$	is a string expression for the name of the variable to read. It can be a boolean, integer, double integer, real, or an array of these types. Only single dimension arrays are allowed.
value	is the variable where the value read is written.

This function accepts a variable name as a string expression and returns the value in variable VALUE. The string that defines the variable name must have a suffix as follows:

@	Booleans
%	Integers
!	Double integer
No suffix	Reals

If specifying an array element, the subscript must be after the data type character if there is one. Only common variables can be accessed.

Values Returned:

1	Success
−22	Variable not found
−23	Data type mismatch

For example:

```
VARIABLE_NAME$ = "SET_POINTS(17)"

STATUS% = READVAR%( VARIABLE_NAME$, VALUE )
```

7.39 WRITEVAR% Function

This function requires the AutoMax operating system with the Ethernet option.

Format:

```
WRITEVAR%( vn$, value )
```

where:

vn$	is a string expression for the name of the variable to write to. It can be a boolean, integer, double integer, real, or an array of these types. Only single dimension arrays are allowed.
value	is the variable that has the value to write.

This function accepts a variable name as a string expression and a value to write into the variable. The string that defines the variable name must have a suffix as follows:

@	Booleans
%	Integers
!	Double integers
No suffix	Reals

If specifying an array element, the subscript must be after the data type character if there is one.

If the data type of the variable, as defined in the string vn$, is different than that of VALUE, an error is generated. Only common variables can be accessed.

Values Returned:

1	Success
−22	Variable not found
−23	Data type mismatch
−24	Variable forced

For example:

```
VARIABLE_NAME$ = "SET_POINTS(17)"

VALUE = 12.345

STATUS% = WRITEVAR%( VARIABLE_NAME$, VALUE )
```

7.40 FINDVAR! Function

This function requires the AutoMax operating system with the Ethernet option.

Format:

```
FINDVAR! ( varname$ )
```

where:

varname$	is a string expression for the name of the variable to find.

This function accepts a variable name as a string expression and returns a pointer to that variable. This may then be used in the SENDL% and RECVL% functions.

@	Booleans
%	Integers
!	Double integers
$	Strings
No suffix	Reals

If specifying an array element, the subscript must be after the data type character if there is one.

Values Returned:

>0	Pointer to Variable
−22	Variable not found

For example, to find a pointer to XYZ%(10):

```
VARIABLE_NAME$ = “XYZ%(10)”

POINTER! = FINDVAR!( VARIABLE_NAME$ )
```

7.41 CONVERT% Function

This function requires the AutoMax operating system with the Ethernet option. This function can be used on the AutoMax PC3000 although the PC3000 does not support the other Ethernet functions.

Format:

```
CONVERT%     ( src_variable, src_subscript, dest_variable,   &
             dest_subscript, num_of_words, mode )
```

where:

src_variable	is the variable that selects were to get data from. This parameter may be a scalar or an array of any data type. If src_variable is an array, it should only be the base name and any data type character.
src_subscript	is only used if the src_variable is an array. It determines where in the array to begin reading. If not an array, the value should be 0.
dest_variable	is the variable that selects were to move the data. This parameter may be a scalar or an array of any data type. If dest_variable is an array, it should only be the base name and any data type character.

dest_subscript	is only used if destination_variable is an array. It determines where in the array to begin writing. If not an array, the value should be 0.
num_of_words	selects the number of words to move.
mode	determines the mode of operation.

VALUE	FUNCTION
0	Move data with no change in format
1	Convert from Motorola Floating Point to IEEE format
2	Convert from IEEE Floating Point to Motorola format
4	Word swap (0102H to 0201H)
8	Long word swap (01020304H to 04030201H)
9	Motorola to IEEE followed by long word swap
10	Long word swap followed by IEEE to Motorola

All other values are illegal

This function is used to convert between data formats used by AutoMax and data formats used by other computers.

Values Returned:

1	Success
−26	Array is not single dimension
−32	Beyond end of array
−33	Illegal mode value
−34	Zero number of words
−35	Odd number of words on long word swap
−36	Number of words > dest data type when dest memory is on CPU

For example, to move 30 real numbers beginning at SRC_ARRAY(10) to DST_ARRAY(20) converting from Motorola to IEEE and inverting the byte order:

```
STATUS% = CONVERT%( SRC_ARRAY, 10, DST_ARRAY, 20,
            15, 9 )
```

7.42 RTS_CONTROL@ Function

Format:

```
RTS_CONTROL@(#n, control_val%)
```

where:

#n is the logical number assigned to the port in the OPEN statement.

control_val% (input constant of variable integer) is non-zero to turn the RTS signal on and 0 to turn the RTS signal off.

This function provides control of the RTS modem control signal. If hardware handshaking is enabled and RTS is set true using this function, RTS will remain true after all characters have been transmitted so that the application task can set it false after a delay. Refer to the OPEN statement for a description of the purpose of the RTS signal.

This function returns false only under the following conditions:

- #n is not assigned to an opened port
- #n is not assigned to a serial communication port (PORTA on the leftmost Processor; PORTA or PORTB on other Processors)

The following example would turn RTS on, wait 55ms, send the text, wait for all characters to be sent, wait 110ms, then turn RTS off.

```
10  OPEN "PORTA" AS FILE #1, SETUP=0,9600
30  CONTROL_VAL% = 1                /turn RTS on
40  STATUS@ = RTS_CONTROL@(#1, CONTROL_VAL%)
50  DELAY 10 TICKS                  /delay 55 ms
60  PRINT #1, SEND$                 /send text out port A
65  IF (ALL_SENT(#1)) THEN GOTO 70
66  DELAY 1 TICK : GOTO 65          /wait for all sent
70  DELAY 20 TICKS                  /delay 110 ms
80  CONTROL_VAL% = 0                /turn RTS off
90  STATUS@ = RTS_CONTROL@(#1, CONTROL_VAL%)
```

7.43 ALL_SENT@ Function

Format:

ALL_SENT@(#n)

where:

#n is the logical number assigned to the port in the OPEN statement.

This function provides a method to detect when all characters in a message are sent.

This function returns true if all of the characters from the previous print statement have been transmitted (that is, the transmit queue is empty).

This function returns false only under the following conditions:

- Some characters have not yet been transferred.
- #n is not assigned to an opened port
- #n is not assigned to a serial communication port (PORTA on the leftmost Processor; PORTA or PORTB on other Processors)

Refer to the OPEN statement for a description of the purpose of the RTS signal.

7.44 WRITE_TUNE Function

Format:

WRITE_TUNE(tunable, new_value, in_limit@)

where:

tunable is the tunable variable. It can be an integer, double integer, or real.

new_value is the new value for the tunable variable. It can be a variable, an expression, or a literal value. The compiler will report an error if 'tunable' and 'new_value' are not the same data type.

in_limit@ is true if 'new_value' is less than or equal to the tunable variable's high limit and greater or equal to the tunable variable's low limit. This parameter is optional.

This function provides the ability to change the current value of tunable variables. This function will respect the tunable variable's high and low limits. The compiler will disallow use of any other statement, function, or control block to write to tunable variables.

The "modified" flag for the task (in the on-line task list) will be set when the task executes this function. On the UDC module, the new current value will be written to non-volatile flash memory. The new value will not be written to flash memory if it is the same as the old value.

On the UDC module, if the tunable variable is a Power Module Interface (PMI) gain, then all the PMI gains will be multiplexed with the setpoint data to the PMI when the tunable value is changed. The gains will not be sent if the new value of the tunable variable is the same as the current value.

Note that frequent tunable updates via this function may slow the response time of the Programming Executive and will count towards the maximum limit on write operations to the UDC. See the UDC module instruction manual, S-3007, for more information.

8.0 ETHERNET COMMUNICATION FUNCTIONS

The following functions are used only with the Ethernet Network Interface (ENI) module (M/N 57C440). Before any of the following functions can be used, the AutoMax operating system with the Ethernet option must be loaded onto the Processor. Note that tasks that use these functions must be run on the leftmost Processor in the rack. These functions cannot be used on the AutoMax PC3000.

Function Name	**Description**
ENI_INIT%	Initializes ENI
SOCKET%	Creates socket
BIND%	Binds socket
CONNECT%	Assigns destination for socket
ACCEPT%	TCP only; directs a passive open
SEND%	Sends specified type of variable
SENDL%	Sends double integer array
RECV%	Writes received variable
RECVL%	Writes received array
SETSOCKOPT%	Sets socket option
GETSOCKOPT%	Reads socket option
SHUTDOWN%	Closes socket

8.1 ENI_INIT% Function

Format:

```
ENI_INIT%( slot%, addr$, tcp%, udp%, ether% )
```

where:

slot%	is the logical slot the ENI is to be in. This can be a variable or a constant. The only legal values are 2 or 4.
addr$	is the Internet address to assign to the ENI. This is a string of four decimal numbers separated by decimal points, each ranging from 0 to 255. A typical address is 128.0.0.10.
tcp%	defines the number of sockets to use for the TCP protocol.
udp%	defines the number of sockets to use for the UDP protocol.
ether%	defines the number of sockets to use for Raw Ethernet.

The ENI_INIT% function commands the Ethernet Network Interface to go through its initialization. The ENI supports three types of protocols: TCP, UDP, and Raw Ethernet. Up to 64 channels (sockets) can be assigned to each ENI. Part of the initialization selects how may sockets to allow for each protocol. The green LED on the front of the ENI will turn off for approximately 10 seconds while the initialization is performed.

Values Returned:

1	Success
−1	ENI failed self test
−8	Bus error
−10	Error allocating interrupts
−11	Bad slot number
−12	Bad Internet address
−13	Total number of sockets >64

For example:

```
STATUS% = ENI_INIT%( 4, “128.0.0.10”, 32, 10, 3 )
```

8.2 SOCKET% Function

Format:

```
SOCKET%( slot%, type% )
```

where:

slot%	is the logical slot the ENI is to be in. This can be a variable or a constant. The only legal values are 2 or 4.
type%	is used to select the protocol for this socket. Legal values for type are: 1 for a TCP socket 2 for a UDP socket 3 for a Raw Ethernet socket

This function will find an available socket of the requested type. If successful, the value returned is the number of the socket allocated. All subsequent function calls to communicate with the ENI use this socket number to select the socket to talk through.

Values Returned:

>0	The socket number allocated
−2	ENI not initialized
−3	Did not create socket
−9	No buffer space
−11	Bad slot number
−14	Bad socket type

For example:

```
SOCKET_NUM% = SOCKET%( 4, 1 )
```

8.3 BIND% Function

Format:

```
BIND%( sn%, port% )
```

where:

sn%	is the number of the socket you want to bind. This is the value that was returned from the SOCKET%

	function. This can be specified as a simple variable or as an element of an array.
port%	is the local port number you want to give to the socket. Begin assigning TCP and UDP port numbers at 5000. For raw Ethernet sockets, this value is used to select the value of the 16-bit packet type in the message header.

This function assigns a local port number or Ethernet packet type to a socket.

Values Returned:

1	Success
−2	ENI not initialized
−4	Did not bind socket
−9	No buffer space
−15	Bad socket number

For example:

```
STATUS% = BIND%( SN%, 5000 )
```

8.4 CONNECT% Function

Format:

```
CONNECT%( sn%, addr$, port% )
```

where:

sn%	is the number of the socket you want to connect to a destination. This is the value returned from the SOCKET% function. This can be specified as a simple variable or as an element of an array.
addr$	is the destination Internet or Ethernet address you want to connect to. See ENI_INIT for applicable rules for Internet addresses. Ethernet addresses are 12-digit Hex number strings.
port%	is the destination port number you want to connect to.

This function assigns a permanent destination for a socket. It must be done before any messages can be sent using any of the three protocols. For Raw Ethernet or UDP sockets, this function is used only to specify the destination address. For TCP sockets, it directs the ENI to do an active open. A passive open (ACCEPT%), done by the destination TCP socket, must occur prior to this function being executed to establish a connection. After the connection is made, messages can be exchanged.

If connecting a TCP socket and the other end is not ready to accept the connection, the socket will be closed. To try to connect again, the application must create a new socket and bind it again.

For raw Ethernet sockets, the port number defines the packet type of all messages that will be sent. The receiving end must do a BIND% with the same value for the port number.

Values Returned:

1	Success
−2	ENI not initialized
−9	No buffer space
−12	Bad InterNet address
−15	Bad socket number
−102	Socket not connected

For example:

```
STATUS% = CONNECT%( SN%, DEST_INET_ADDR$, 5001 )
```

8.5 ACCEPT% Function

Format:

```
ACCEPT%( sn%, nsn% )
```

where:

sn%	is the number of the TCP socket that should begin waiting for a connection to be made. This can be specified as a simple variable or as an element of an array.
nsn%	is filled in by this function with a new socket number created when a connection has been established. This must be a simple variable; array elements are not allowed.

This function is used to direct the ENI to do a passive open. This is valid only on TCP sockets. This function suspends execution of the task and waits until a connection is established. When a connection arrives, it creates a new socket with the attributes of the given socket to service the connection. The application program may then shut down the original socket sn%, or it may loop back to the ACCEPT% to wait for another connection to come in. In this way a given service may have more than one client at a time. Communication will take place through the new socket.

Values Returned:

1	Success
−2	ENI not initialized
−7	Did not accept
−9	No buffer space
−15	Bad socket type
−16	Not a TCP socket

For example:

```
STATUS% = ACCEPT%( SN%, NSN% )
```

8.6 SEND% Function

Format:

```
SEND%( sn%, var, len% )
```

where:

sn%	is the number of the socket through which the message is to be sent. This is the value that was returned from the SOCKET% or ACCEPT% function. This can be specified as a simple variable or as an element of an array.
var	is the variable that has the data to send. It can be a boolean, integer, double integer, real, string, or an array of these types. It may be local or common. If an array is specified, no subscript may be given. It will always start with the zeroth element of the array.
len%	is the number of bytes to send beginning at var. This parameter can be a constant, an integer, or a double integer. If var is an array, and len% is zero, the length to send is the size of the array. An error is generated if len% is greater than the size of the array.

This function causes a message to be sent to the destination as defined by the socket number.

If a TCP socket is specified, it must be connected. If the connection was established with an ACCEPT%, the destination is the station that established the connection.

Values Returned:

>0	Number of bytes transferred
−2	ENI not initialized
−9	No buffer space
−15	Bad socket number
−17	Message too long, UDP > 1472, ETH > 1500
−18	Zero length for non-array
−26	Array is not single dimension
−32	Beyond end of array
−102	Socket not connected

For example:

```
XMIT_LEN% = SEND%( SN%, SET_POINTS%, MSG_LEN% )
```

where SET_POINTS% is the name of an array.

8.7 SENDL% Function

Format:

```
SENDL%( sn%, list! )
```

where:

sn%	is the number of the socket through which the message is to be sent. This is the value that was returned from the SOCKET% or ACCEPT% function. This can be specified as a simple variable or as an element of an array.
list!	is a single dimension double integer array whose size is limited only by memory capacity. The values in this array define where to get the data to send. No subscript is given on this parameter. Beginning at list!(0), the even elements of the array contain pointers to the data to send and odd elements of the array contain the number of bytes to transfer. The number of bytes must be even. The value for pointers is found with the VARPTR! or FINDVAR! function. The list is terminated by a pointer with a value of zero at the even boundary.

This function causes a message to be sent to the destination as defined by the socket number.

If a TCP socket is specified, it must be connected. If the connection was established with an ACCEPT%, the destination is the station that established the connection.

Values Returned:

>0	Number of bytes transferred
−2	ENI not initialized
−9	No buffer space
−15	Bad socket number
−17	Message too long
−18	Zero length
−19	Illegal Pointer
−25	Not a double integer array
−26	Not a single dimension array
−27	Bad array format
−30	Odd number of bytes in list parameter
−102	Socket not connected

For example:

```
XMIT_LEN% = SENDL%( SN%, NETWORK_LIST! )
```

8.8 RECV% Function

Format:

RECV%(sn%, var, len%)

where:

sn%	is the number of the socket through which the message is to be received. This is the value that was returned from the SOCKET% or ACCEPT% function. This can be specified as a simple variable or as an element of an array.
var	is the variable where the data received is written. It can be a boolean, integer, double integer, real, string, or an array of those types. If an array is specified, no subscript may be given.
len%	is the number of bytes to receive. This parameter can be a constant, an integer, or a double integer. If var is a simple variable and len% is greater than the size of the simple variable, then var must be defined as I/O to avoid overwriting AutoMax memory. If var is an array, and len% is zero, the length to receive is the size of the array. An error is generated if len% is greater than the size of the array. For TCP only, if len% is −1, the number of bytes received will be returned to the sender.

This function writes up to len% bytes of data from socket SN% into the variable VAR.

If a TCP socket is specified, it must be connected.

A socket can be selected as blocking or non-blocking. If the socket is blocking and no data has come in, the task will be suspended until data arrives. If the socket is non-blocking and no data has come in, the RECV% command will return with the error No message waiting. The default mode is blocking.

Values Returned:

>0	Length of message received
−2	ENI not initialized
−9	No buffer space
−15	Bad socket number
−17	Message too long
−18	Zero length for non-array
−26	Array is not single dimension
−29	Max size of strings are not equal
−31	Max size of string < recv size of string
−32	Beyond end of array
−101	No message waiting
−102	Socket not connected

For example:

```
RECV_LEN% = RECV%( SN%, SET_POINTS%, LEN%)
```

8.9 RECVL% Function

Format:

```
RECVL%( sn%, list! )
```

where:

sn%	is the number of the socket through which the message is to be received. This is the value that was returned from the SOCKET% or ACCEPT% function. This can be specified as a simple variable or as an element of an array.
list!	is a single dimension double integer array whose size is limited only by memory capacity. The values in this array define where to put the data to received. No subscript is given on this parameter. Beginning at list!(0), the even elements of the array contain pointers to where to put the data and odd elements of the array contain the number of bytes to accept. The number of bytes must be even. The value for pointers is found with the VARPTR! or FINDVAR! function. The list is terminated by a pointer with a value of zero at the even boundary.

This function receives data from socket SN% into memory pointed to by the list. All pointers must reference variables defined as I/O. Pointers may not reference variables defined in the Common Memory Module or AutoMax Processor.

If a TCP socket is specified, it must be connected first.

A socket can be selected as blocking or non-blocking. If the socket is blocking and no data has come in, the task will be suspended until data arrives. If the socket is non-blocking and no data has come in, the RECVL% command will return with the error No message waiting. The default mode is blocking.

Values Returned:

>0	Number of bytes transferred
−2	ENI not initialized
−9	No buffer space
−15	Bad socket number
−17	Message too long
−18	Zero length
−19	Illegal pointer
−25	Not a double integer array
−26	Not a single dimension array
−27	Bad array format
−30	Odd number of bytes in list parameter
−101	No message waiting
−102	Socket not connected

For example:

```
RECV_LEN% = RECVL%( SN%, NETWORK_LIST! )
```

8.10 SETSOCKOPT% Function

Format:

```
SETSOCKOPT%( sn%, opnum%, opval% )
```

where:

sn%	is the number of socket whose option you want to set.
opnum%	is the number of the option to set.
opval%	is the value to write into the ENI.

This function is used to select different modes of operation. OPNUM% selects which option to change, and OPVAL% selects the mode of operation.

Options	**OPNUM%**	**OPVAL%**	**Description**
Keep Alive	0008h	0	Keep alive is disabled (Default)
		1	Keep alive is enabled

This option is only used on TCP sockets. When enabled, the ENI will periodically send an empty message to maintain the connection. If this option is not used and a frame is not received within 8 minutes the ENI will assume it has been broken.

Linger	0080h	0	Linger is disabled (Default)
		1	Linger is enabled

This option is only used on TCP sockets to select how the SHUTDOWN function will operate. When linger is enabled and there are messages in any transmit or receive queues the ENI will process those messages before doing the shutdown.

Non Blocking	0200h	0	Non Blocking is disabled (Default)
		1	Non Blocking is enabled

This option is used to select how the RECV% and RECVL% function will operate. If Non blocking is enabled and no message has arrived for the RECV% or RECVL%, control is returned to the application program and an error code -101 is returned by the RECV% or RECVL%.

Values Returned:

1	Success
−2	ENI not initialized
−5	Did not set option
−9	No buffer space
−15	Bad socket number
−20	Bad option number
−21	Bad option value

For example, to set the socket to nonblocking:

```
STATUS% = SETSOCKOPT%( SN%, 0200h, 1 )
```

8.11 GETSOCKOPT% Function

Format:

GETSOCKOPT%(sn%, opnum%, opval%)

where:

sn%	is the number of socket whose option you want to read.
opnum%	is the number of the option to read.
opval%	is the name of the option variable where the current value is written.

This function is used to examine what modes of operation are selected. OPNUM% selects which option to look at, and OPVAL% displays the current status.

Options	**OPNUM%**	**OPVAL%**	**Description**
Keep Alive	0008h	0	Keep alive is disabled (Default)
		1	Keep alive is enabled

This option is only used on TCP sockets. When enabled, the ENI will periodically send an empty message to maintain the connection. If this option is not used and a frame is not received within 8 minutes, the ENI will assume it has been broken.

Options	**OPNUM%**	**OPVAL%**	**Description**
Linger	0080h	0	Linger is disabled (Default)
		1	Linger is enabled

This option is only used on TCP sockets to select how the SHUT-DOWN function will operate. When linger is enabled and there are messages in any transmit or receive queues the ENI will process those messages before doing the shutdown.

Options	**OPNUM%**	**OPVAL%**	**Description**
Non Blocking	0200h	0	Non Blocking is disabled (Default)
		1	Non Blocking is enabled

This option is used to select how the RECV% and RECVL% function will operate. If Non Blocking is enabled and no message has arrived for the RECV% or RECVL%, control is returned to the application program and an error code -101 is returned by the RECV% or RECVL%.

Options	**OPNUM%**	**OPVAL%**	**Description**
Connected	0800h	0	Socket not connected
		1	Socket connected

This option is only used on TCP sockets. It allows the application program to test if a connection is established without doing a SEND% or RECV%.

Values Returned:

1	Success
−2	ENI not initialized
−6	did not get option
−9	No buffer space
−15	Bad socket number
−20	Bad option number
−100	No buffer space

For example, to test if the socket is connected:

```
STATUS% = GETSOCKOPT%( SN%, 0800h, OPTION_VALUE%)
```

8.12 SHUTDOWN% Function

Format:

```
SHUTDOWN%( sn% )
```

where:

sn%	is the number of the socket for which the connection should be terminated.

This function closes the socket to allow it to be reused at a later time.

TCP sockets need to be shut down at only one end. Either the active or passive side may close the connection. The other side will automatically shut down. UDP and Raw Ethernet sockets need to be shut down at both ends.

Values Returned:

1	Success
−2	ENI not initialized
−15	Bad socket number
−28	Socket closed by destination

For example:

```
STATUS% = SHUTDOWN%( SOCKET_NUM% )
```

Appendix A

Converting a DCS 5000 BASIC Task to AutoMax

You can easily convert any DCS 5000 Version 4 task to AutoMax. Simply re-compile the task using the AutoMax Executive Programming software. Refer to J-3684, the ReSource AutoMax Programming Software instruction manual, for more information on compiling a BASIC task.

Converting a Version 1.0 BASIC Task to Version 2.0

You can easily convert any AutoMax Version 1.0 BASIC task created with M/N 57C304-57C307 Executive software to run in a 2.0 (M/N 57C390-57C393) system. Simply re-compile the task using AutoMax Executive Programming software V 2.0 (M/N 57C390-57C393). Refer to J-3684 for more information on compiling a BASIC task.

Converting a Version 2.0 BASIC Task to Version 3.0

You can easily convert any AutoMax Version 2.0 BASIC task created with M/N 57C390-57C393 Executive software to run in an AutoMax V3 system. Simply copy the task into AutoMax V3 using the AutoMax Executive Programming software V 3.0 (M/N 57C390-57C393), and then re-compile the task. Refer to J-3750 for more information on copying and compiling a BASIC task.

Appendix B

BASIC Compiler and Run Time Error Codes

The following error codes are displayed on the screen when tasks are compiled.

Control Block Error Codes (BASIC Compiler)

257	Bad control block statement format
258	Unrecognized name for control block
259	Missing END statement in control block task
260	Not assigned
261	Variable used in control block not defined
262	Bad literal value for KI, KP, or KD
263	Bad WLD * KP/C value (See AutoMax Control Block Language Instruction Manual; J-3676.)
264	Bad literal value for DEAD_BAND, MAX_CHANGE, or LOOP_TIME
265	Invalid data type for literal in control block
266	Incomplete input pairs or input/output pairs in a control block
267	Bad SCALE, REQUIRED_SAMPLES, or MAX_COLUMNS Value
268	Bad specification for array in control block
269	Control block not the only statement for that line number
270	CML specified literal field out of range
271	SCAN_LOOP block not allowed with CML block
272	Integer literal field too large
273	Invalid parameter keyword in control block
274	Calculated K value out of range
275	Literal symbol too long
276	Required control block field missing
277	Required control block literal missing
278	Control block field must be literal
279	Control block field must be variable
280	Non-contiguous inputs, input pairs, or input/output pairs in control block
281	Missing SCAN_LOOP block in control block task
282	Signed boolean literal or numeric variable not allowed
283	WLD value out of range
284	Invalid value for Lead Lag W
285	Invalid value for WM
286	Invalid value for WLD
287	Word size out of range
288	Array specified has too many subscripts
289	Integer literal > 24 bits; can't be accurately converted to real
290	Invalid value for Max_Input
291	More than 1 SCAN_LOOP call in a control block task
292	Fast floating point overflow
293	Fast floating point underflow
294	Fast floating point divided by zero
295	Meaningless tangent argument
296	Minimum number of inputs or outputs not programmed

297	Invalid data type for variable in Control Block
298	Parameter keyword previously defined in Control Block
299	Data structure symbol name too long
300	Data Structure requires more than maximum storage
301	Number of inputs/outputs greater than data structure definition
302	Duplicate definition or incorrect data structure type
303	Invalid Control Block Mode specified
304	Bad NOTCH bock Q_FACTOR value
305	Bad NOTCH block WN value

IODEF, RIODEF, NETDEF, RNETDEF, MODDEF Error Codes

306	Bad IODEF statement format
307	IODEF address must not be odd
308	Bad IODEF variable type
309	IODEF hex address too large
310	Invalid bit number specification in RIO/NET DEF
311	Invalid literal in RIO/NET DEF
312	Missing master slot specification in RIODEF
313	Bad literal in IODEF
314	Missing bit field specification
315	Missing slot specification
316	Bad RIO/NET DEF statement format
317	Missing drop specification
318	Bad MODDEF statement format
319	Bad GATEWAY register specification
320	Bad RNETDEF statement format
321	Bad RNETDEF register specification
322	Invalid variable data type in GATEWAY definition
323	Bad ABDEF statement format
324	Bad file specification in ABDEF statement
325	Bad BOOLEAN literal specification
326	Bad GBLDEF statement format
327	Invalid network specification
328	Type of NET_NAME does not match type of variable
329	Network file not found
330	No network file drive
331	Invalid network file drive
332	Error opening OBN file
333	Error reading from OBN file
334	Network variable name not found on OBN file
335	No memory to build network functions

Function Call Error Codes

336	Invalid function call format
337	Incorrect number of parameters in function call
338	Bad parameter data type in function call or bad array subscript
339	Parameter symbol not defined
340	Variable must be simple (not array variable)
341	Invalid function parameter
342	Invalid function expression

343	Bad function variable
344	Bad array; must be 1 dimension (integer)
345	Bad BLOCK_MOVE variable
346	Variable in function call not defined as COMMON
347	Ticks per scan too high

Insufficient Memory Error Codes

356	Insufficient memory to compile array
357	Insufficient memory to compile FOR statement
358	Insufficient memory to build symbol table
359	Insufficient symbol table memory
360	Object code buffer overflow
361	Opcode position overflow; statement too long
362	No more user stack
363	No more program stack
364	No more type stack; expression too long
365	No more operator stack; expression too long
366	No more memory to link object code buffer

FOR-NEXT Error Codes

376	FOR control variable cannot be a tunable variable
377	NEXT control variable does not match FOR control variable
378	Control variable must be simple variable (not array).
379	Invalid data type on control variable in FOR statement
380	Bad FOR statement format
381	Invalid statement type following THEN in IF statement
382	Missing expected THEN
383	Invalid data type for expression in FOR statement
384	Missing corresponding FOR statement
385	FOR loops nested too deep
386	IF-THEN-ELSE statement >32K
387	Missing <CR> or old and new IF-THEN formats mixed

OPEN, CLOSE, INPUT, PRINT Error Codes

396	Bad device name
397	Bad logical file number specification
398	Bad device name for OPEN statement
399	Bad baud rate in OPEN SETUP parameter
400	Invalid device specification
401	Bad OPEN statement format
402	Duplicate logical file number
403	Invalid CLOSE statement format
404	Invalid device name
405	Missing expected print field
406	Specified file has not been defined (no OPEN)
407	Device must be accessed by OPEN first
408	Invalid data type for print using format
409	Bad PRINT USING format
410	Specified format field width too wide
411	Cannot have print using with channel

412	Bad GET statement format
413	Bad PUT statement format
414	Bad INPUT statement format
415	Cannot close a channel
416	Cannot GET from a channel
417	Cannot PUT to a channel
418	Bad SETUP specification in OPEN device
419	Open device attempted on a channel
420	Bad format in OPEN FOR READ or OPEN FOR WRITE
421	Invalid keyword in OPEN configuraton
422	Bad ACCESS parameter (must be EXCLUSIVE or NON_EXCLUSIVE)
423	SETUP requires EXCLUSIVE access

START, WAIT, DELAY, EVENT Error Codes

426	Invalid time units specification
427	Missing DELAY expression
428	Bad START EVERY statement
429	Bad WAIT statement format
430	Invalid event name
431	Bad EVENT statement format
432	Bad time units in START statement
433	Delay time units must be an integer
434	Duplicate event name
435	Missing start interval
436	Missing event definition

Channel I/O Error Codes

446	Missing DEPTH parameter on OPEN CHANNEL FOR INPUT
447	Bad OPEN CHANNEL format
448	Bad channel template in OPEN statement
449	Invalid DEPTH specification for OPEN CHANNEL
450	INPUT/PRINT reference does not match channel template
451	Not assigned
452	Channel template too large
453	Channel packet too large
454	Channel was opened for input but output attempted
455	Channel was opened for output but input attempted

Array Error Codes

466	Array requires more than maximum storage
467	Bad array subscript
468	Number of subscripts does not match definition
470	Missing array dimension
471	Too many array subscripts

Miscellaneous Compiler Error Codes

484	New value must be same type as tunable in WRITE_TUNE
485	Tunable variable expected
486	Missing delimiter
487	Missing equal sign "="
488	Missing left parenthesis "("
489	Missing right parenthesis ")"
490	Missing expected comma "," or semicolon ";"
491	Missing line number
492	Invalid line number
493	Line number out of range (must be 1 to 32767)
494	Invalid data type mixing in expression
495	Invalid variable type
496	Variable name same as reserved symbol
497	Variable name too long
498	Missing variable name
499	Variable name too long
500	Invalid subscripted variable
501	Invalid variable specified in READ statement
502	Missing variable definition
503	Invalid statement terminator; expecting EOS
504	Task must be a CONFIGURATION task
505	Missing operand (symbol or literal)
506	Missing arithmetic/relational operator
507	Not a valid statement for this task type
508	Invalid integer expression for ON GOTO
509	Invalid ON GOTO statement format
510	Missing expected TO
511	Expected expression not found
512	Missing expected line number
513	Invalid boolean expression
514	Invalid tunable statement definition
515	Symbol already defined; duplicate definition
516	Invalid data type for a tunable variable
517	Tunable variable ranges are inconsistent
518	Undefined variable or statement not permitted in this type of task
519	Invalid tunable variable definition format
520	Tunable cannot be array, left side of equal, or control block output
521	Missing expected variable
522	DATA statement not first statement for this line number
523	Not assigned
524	Overflow in ASCII to binary integer conversion
525	Numeric literal too large
526	Real literal too large
527	Null buffer overflow; statement too large
528	Object buffer overflow; statement too large
529	Expression evaluator; stack integrity lost;expression too long
530	Compiler integrity lost
531	Illegal symbol in REM statement
532	CALL statement not first statement for this line number

533	Task not of type BASIC, CONTROL, or CONFIGURATION
534	Invalid task statement format
535	Invalid task priority
536	Invalid task name
537	Invalid slot specification
538	Missing string variable in GET or PUT statement
539	Illegal on board I/O address specified
540	Bad IOWRITE format
541	Bad IOWRITE option expression
542	Bad IOWRITE value expression
543	Bad IOWRITE address expression
544	REM statement not first statement on the line
545	Bad ON ERROR statement format
546	Fatal expression evaluation error; no opcode match
547	String literal too large
548	Too many total elements for an array
549	Array variable was referenced as a simple variable
550	Illegal state in expression evaluation; integrity lost
551	Bad expression in SET_MAGNITUDE statement
552	Bad SET_MAGNITUDE statement format
553	Bad variable type in SET_MAGNITUDE statement
554	Invalid TIMEOUT expression in EVENT statement
555	Symbol > 255 characters long; statement too long
556	Bad IF statement transfer line number
557	Invalid characters after the ampersand continuator
558	Remark statement too long
559	Line number out of range
560	Must be first statement on the line
561	Symbol is not a variable name
562	Loss of precision in converting real number
563	ELSE or END_IF encountered while not compiling an IF statement
564	ELSE not needed (condition already satisfied with ELSE or END_IF)
565	Missing THEN in IF−THEN ELSE statement
566	IF−THEN SECTION nested to deep
567	ELSE not valid in old style IF−THEN format statement
568	Path prefix not supported or bad format for include file name
569	Nested include files not supported
570	Undefined compiler directive
571	String length specified > 256 characters
572	String length specified not of type integer
573	Cant update INCLUDE structure info − object code corrupted
574	Include file could not be opened
575	Invalid file extension - must be .INC
576	Missing rate specification
577	Invalid rate specification
578	Redefinition of tick rate for this slot
579	Invalid tickdef statement format
580	No executable statements encountered! No object code can be generated

Control Block Related Error Codes

590	Invalid value for WLG
591	Invalid value for ORDER
592	Control Blocks encountered before SCAN_LOOP
593	Invalid value for RES_N
594	Invalid value for RES_D
595	Invalid value for ZETA_N
596	Invalid value for ZETA_D

Resolution Error Codes

656	Line used in RESTORE is not a DATA statement
657	FOR and NEXT variables do not match
658	Insufficient memory to compress object code
659	Object code larger than 32K
660	Stack requirements too large
661	Data structures too large
662	Symbol table integrity lost
663	Insufficient memory for post-compile resolution
664	Line number not resolved
665	Symbol offset too big; task too large
666	No TASK statement in configuration task
667	No symbols in configuration task
668	Duplicate data pointers with same data type; caused by assigning two DIFFERENT variables of the SAME type to the SAME register or bit: IODEF YES@[slot=3,REGISTER=1,BIT=1] IODEF NO@[slot=3,REGISTER=1,BIT=1] or NETDEF FIRST%[slot=2,DROP=1,REGISTER=2] NETDEF SECOND%[slot=2,DROP=1,REGISTER=2]
669	Symbol table too large; too many symbols
670	Invalid condition; integer literal in BASIC task symbol table
671	Unable to allocate enough space for symbol table
672	Symbol table integrity lost
673	Too many COMMON integers, double integers, booleans used
674	Unable to allocate space for the BASIC runtime structure header
675	Too many LOCAL integers, double integers, booleans used
676	Too many LOCAL integers, double integers, boolean literals used
677	Too many COMMON reals, strings, arrays used
678	Too many LOCAL reals, strings, arrays used
679	Too many OPEN CHANNEL statements
680	Too many arrays used
681	Too many FOR loops used
682	Too many real literals used
683	Too many real tunable variables defined
684	Invalid condition; literal in CONFIGURATION task
685	Invalid condition; string literal type in the symbol table
686	Offset to real literal in CONTROL task greater than 16 bits
687	Invalid condition; LOCAL variable in CONFIGURATION task
688	Invalid condition; relative symbol number not resolvable
689	Offset required for relocatable reference greater than 16 bits

690	Error opening the object output file
691	Error writing to object output file
692	Task with READ statements but no DATA statements
693	Too many LOCAL integers, double integers, boolean variables used
694	Unable to allocate enough space for object code
695	Undefined Control Block data structure found
696	Error closing source file (disk may be full)
697	Error closing log file (disk may be full)
698	Error closing include file
699	Error attempting to load time/date into object file
700	Object size >32767 in Control Block task
701	Symbol & data size >32767 in Control Block task
702	Object+Symbol+Data size >20480 in UDC control block task
725	Invalid number of array dimensions

Run Time Error Codes

The following error codes are displayed in the error log accessible from the ON-LINE menu when the task is running.

756	Arithmetic integer overflow code
757	Arithmetic real overflow code
758	String concatenate overflow
759	Divide by zero
760	Integer multiply overflow
761	Integer assign overflow
762	Single integer conversion overflow in real to single integer
763	Double integer conversion overflow in real to double integer
764	Real to double integer conversion yields number > 24 bits
765	String overflow
766	Precision lost in real to integer array element conversion
767	Precision lost in real to double integer array element conversion
768	Precision lost in real to single integer conversion
769	Array subscript out of bounds
770	Requested substring > string
771	DATA type in READ statement does not match DATA statememt
772	No more DATA statements
773	Bad line number for RESTORE statement
774	Overflow in conversion of real to integer of FOR loop control variable
775	Overflow in conversion of real to integer of FOR statement TO value
776	Overflow in conversion of real to integer of FOR statement STEP value
777	Integer > 24 bits in STEP value integer to real conversion
778	Bad IOWRITE
779	Integer control variable overflow in FOR statement
780	Double integer control variable overflow in FOR statement
781	Real control variable overflow in FOR statement
782	Negative delay
783	Delay value too large (0 to 32767)
784	Negative start interval
785	Delay value too large (0 to 32767)
786	Not assigned
787	Hardware event # ticks < 0

788	Hardware event ticks overflow
789	Print buffer overflow; print field too long
790	Device not open properly
791	OPEN with bad device address
792	Device not open for write
793	No stack space for print
794	Device not allocated
795	No buffer for print operation; insufficient memory
796	Fatal print error
797	Device already open
798	Device OPENed differently from attempted operation
799	Bad allocate
800	Bad default OPEN
801	Device already closed
802	Device opened as a channel
803	Bad device close; no address
804	Default device not allocated
805	Channel not open
806	Print integer channel overflow
807	Message overflow
808	Unsuccessful channel open
809	Integer > 24 bits in real conversion
810	Real to integer overflow
811	No buffer for GET operation
812	No print buffer
813	Device closed on GET
814	GET not open for read; GET attempted on unopened device
815	Bad GET operation
816	No buffer for PUT operation
817	No print buffer
818	Device closed on PUT statement
819	PUT not open for write; PUT attempted on unopened device
820	Unsuccessful PUT operation
821	Device should be open
822	Invalid baud rate
823	Bad SETUP re-configuration
824	Precision out of range
825	Width too long printing integer field in PRINT USING D format
826	Width too long printing integer field in PRINT USING with L/R/C/Z format
827	Negative decimal places
828	Number decimal points greater than maximum precision allowed
829	Width less than zero
830	Field width overflow
831	Requested substring width less than zero
832	Requested width greater than maximum
833	No space for requested PRINT USING field
834	String greater than field width
835	Bad channel depth
836	Device not open
837	Attempted negative square root

838	First substring position specification greater than string length
839	Not assigned
840	Not assigned
841	Wrong data type input for boolean
842	Another error occurred during execution of ON ERROR routine
843	Could not allocate for write
844	Wrong data type input for string
845	Last substring position less than first substring position
846	First substring position specification ≤ 0
847	Last substring position specification ≤ 0
848	Rotate count greater than 31
849	Overflow on absolute value function
850	Not assigned
851	Device open at END
852	Channel not open on input
853	Wrong type for integer
854	Next character after field not legal
855	Bad next character
856	No input channel I/O buffer
857	Not allocated for re-configuration
858	Bad BCD digit
859	Channel already open
860	Wrong token for comma
861	Not open for read
862	No comma between fields
863	Wrong data type input for real
864	No buffer space can be allocated for I/O
865	Not assigned
866	Invalid re-configuration
867	Missing line number
868	Bad device on input
869	Wrong type for double integer
870	No device address
871	Number greater than 24 bits
872	Not open for write
873	No device address
874	Attempt to execute a null opcode
875	Unbalanced GOSUB-RETURN
876	NEXT does not match loop variable in FOR statement
877	NEXT does not match FOR
878	Bad START statement format
879	Bad hardware event call
880	Undefined opcode
881	Stack overflow
882	No channel buffer space
883	STOP executed
884	Opcode not assigned
885	No event address defined
886	GOSUBs not balanced
887	Bad VAL function conversion

888	BCD output number greater than 9999
889	Bad bit number in function call
890	Bad option number in function call
891	Invalid GATEWAY transfer call Note: Check returned status variable of GATEWAY_CMD_OK@ function; decode status error numbers as follows (01 through 07 are MODBUS exception codes):

Decimal

01	Illegal function code
02	Illegal starting register
03	Illegal data
04	PC aborted
05	Not assigned
06	PC busy
07	Not assigned
08	Illegal data in response message
09	Response timeout error
20	Dual port address error
21	Gateway card not found or not accessible
22	No available Gateway channel
23	Illegal register number
24	Illegal number of register
25	Illegal command number
26	Illegal command number/register set
27	Illegal register number/number of registers
28	Illegal device address

892	BLOCK_MOVE invalid source parameter
893	BLOCK_MOVE invalid destination parameter
894	BLOCK_MOVE invalid transfer size parameter
895	RESUME without executing ON ERROR statement first
956	UDC tick rates do not match
958	PMI gains out of range
959	UDC Processor error
960	Floating point format conversion error
961	Spurious interrupt detected from Multibus
962	CCLK must be enabled to execute task
963	OS background not completing
964	Flash update not completing in time; check utilization

Corrective action: For error code 757, check real and double integer value ranges.

Error Codes Caused By UDC Modules

1000	SCR fault
1001	M-contactor fault
1002	(not used)
1003	Instantaneous overcurrent fault
1004	Synch loss (AC line voltage)
1005	Conduction timeout
1006	Field loss fault
1007	Tach loss fault

1008	Broken wire in resolver
1009	(not used)
1010	Overspeed trip
1011	Power Technology module fault
1012	PMI power supply fault
1013	PMI bus fault
1014	UDC run fault
1015	Fiber optic link com. fault

Corrective action: These errors reflect the status of the drive fault register (A=202, B=1202) on the UDC module. See S-3006 for more information.

Serial I/O Error Codes

The possible causes of serial I/O errors include:

A hardware problem with device connected to serial ports on the face of the Processor module.

1064	EIA control (carrier detect) lost
1065	Parity error (when enabled)
1066	Overrun error
1067	Framing error

Distributed Power System Error Codes

2060	Bad ωN in NOTCH block
2061	Bad Q value in NOTCH block
2062	ResN or resD out of range
2063	ZetaN or zetaD out of range
2064	LIM_BAR is out of range
2065	TRIP_TIME is out of range
2066	THRESHOLD is out of range

Appendix C

Hardware Interrupt Line Allocation

Current Minor Loop (CML) tasks or tasks that use BASIC hardware EVENT statements require Processors to allocate hardware interrupt lines on the rack backplane. This is because some portion of task execution depends upon receiving a user-defined hardware interrupt from another module in the rack, e.g., a Resolver Input module. This appendix will describe the basic method by which interrupt lines are allocated. See the Control Block Language instruction manual (J-3676) for more information on CML tasks.

Because the number of interrupt lines is limited to four, it is necessary to take into account the rules by which they are allocated in order to prevent errors when application tasks are put into run. Each of the four interrupt lines can "service" one of the following:

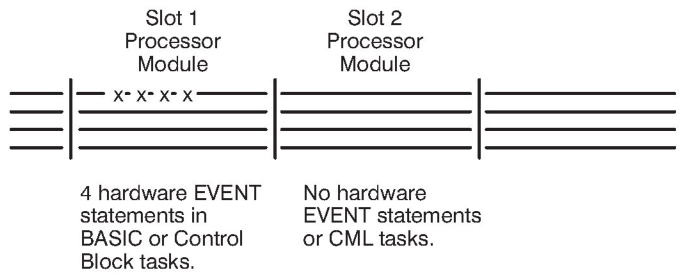
a) four BASIC language hardware EVENT statements in BASIC or Control Block tasks

b) one CML task (used in racks containing drive cards only)

Any one Processor module can allocate one of the four interrupt lines for up to four EVENT statements and one line for a CML task. (CML tasks are limited to two per rack because of drive module configuration restrictions.) A minimum of one hardware interrupt line is allocated for a Processor Module regardless of whether one or four hardware EVENT statements are used in the application tasks loaded on that Processor.

The following examples of interrupt line allocation assume that there are two Processor modules in the rack. Note that these examples do not take into account the efficiency of distributing application tasks between Processor modules in this manner (in terms of system performance).

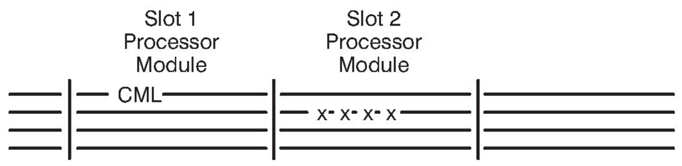
Example #1



Slot 1 Processor Module
Slot 2 Processor Module
x x x x
4 hardware EVENT statements in BASIC or Control Block tasks.
No hardware EVENT statements or CML tasks.

———	=	Interrupt Line
x	=	Hardware EVENT Statement
CML	=	CML Task

Example #2

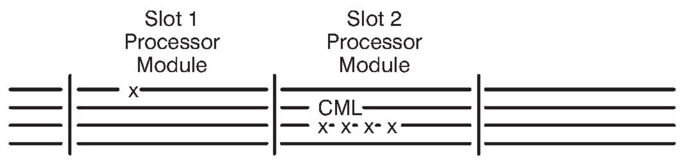


1 CML task

4 hardware EVENT statements

——	=	Interrupt Line
x	=	Hardware EVENT Statement
CML	=	CML Task

Example #3

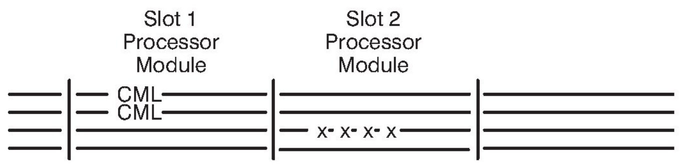


1 hardware EVENT statement

1 CML task and 4 hardware EVENT statements

——	=	Interrupt Line
x	=	Hardware EVENT Statement
CML	=	CML Task

Example #4



2 CML tasks

4 hardware EVENT statements

——	=	Interrupt Line
x	=	Hardware EVENT Statement
CML	=	CML Task

Appendix D

BASIC Language Statements and Functions Supported in UDC Control Block Tasks

The following BASIC statements are supported in UDC Control Block tasks:

CLR_ERRLOG
COMMON
END
FOR-NEXT
GOSUB
GOTO
IF-THEN-ELSE
INCLUDE
LET
LOCAL
ON ERROR
REM OR !
RESUME
RETURN
SET_MAGNITUDE

The following BASIC functions are supported in UDC Control Block tasks:

ABS
ATAN
BCD_IN%
BCD_OUT%
COS
EXP
LN
ROTATEL%
ROTATER%
SHIFTL%
SHIFTR%
SIN
SQRT
TAN
TST_ERRLOG@
WRITE_TUNE

The following expressions are supported in UDC Control Block tasks:

+	Addition, Unary+
−	Subtraction, Unary−
*	Multiplication
/	Division
**	Exponentiation
AND	Logical “AND”
OR	Logical “OR”
XOR	Logical “exclusive-OR”
NOT	Unary boolean operator performs a boolean complement
=	Equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to
><	Not equal to

Appendix E

AutoMax Processor Compatibility with Versions of the AutoMax Programming Executive

The list that follows shows which AutoMax Processor modules can be used with the AutoMax Programming Executive software.

AutoMax Programming Executive Software	**Compatible Processor Module**
Version 1.0	
M/N 57C304, 57C305	M/N 57C430
M/N 57C306, 57C307 (updates)	M/N 57C430
Version 2.0	
M/N 57C390, 57C391	M/N 57C430A
M/N 57C392, 57C393 (updates)	M/N 57C430A
Version 2.1D and later	
M/N 57C391	M/N 57C430A
	M/N 57C431
	M/N 57C435
M/N 57C393 (update)	M/N 57C430A
	M/N 57C431
	M/N 57C435
Version 3.0	
M/N 57C395	M/N 57C430A
M/N 57C397 (update)	M/N 57C430A
Version 3.1	
M/N 57C395	M/N 57C430A
	M/N 57C431
	M/N 57C435
M/N 57C397 (update)	M/N 57C430A
	M/N 57C431
	M/N 57C435
Version 3.3* and later	
M/N 57C395	M/N 57C430A
	M/N 57C431
	M/N 57C435
M/N 57C397 (update)	M/N 57C430A
	M/N 57C431
	M/N 57C435

*Note that if you are using the Programming Executive for drive control applications, the Universal Drive controller (UDC) module (B/M 57552) is supported only in Version 3.3 and later of the Programming Executive software.

Appendix F

New Features

The following are either **new or changed in BASIC for Version 3.0** of the AutoMax Programming Executive.

1. ENI_INIT%
2. SOCKET%
3. BIND%
4. CONNECT%
5. ACCEPT%
6. SEND%
7. SENDL%
8. RECV%
9. RECVL%
10 SETSOCKOPT%
11. GETSOCKOPT%
12. SHUTDOWN%
13. READVAR%
14. WRITEVAR%
15. FINDVAR!
16. CONVERT%

The following are either **new or changed in BASIC for Version 3.3** of the AutoMax Programming Executive.

Section 2.3 -	describes the use of BASIC statements and functions in UDC Control Block tasks.
Appendix D -	lists the BASIC functions and statements supported in UDC Control Block tasks.
Appendix E -	lists the AutoMax Processors that are compatible with versions of the AutoMax Programming Executive software.

The following are either **new or changed in BASIC for Version 3.4** of the AutoMax Programming Executive.

Section 4.1.1 has been revised to state that "E" notation should be used to indicate an exponent in BASIC.

Section 8.0, describing the Ethernet functions, now states that tasks that use the Ethernet functions must be run on the left-most Processor in the rack.

The following functions have been added to BASIC:

RTS_CONTROL@
ALL_SENT@
WRITE_TUNE

Additional BASIC statements and functions that can now be used in UDC tasks:

FOR-NEXT
SET_MAGNITUDE
SIN, COS, TAN, ATAN
EXP
LN
BCD_IN%, BCD_OUT%
WRITE_TUNE

For additional information
1 Allen-Bradley Drive
Mayfield Heights, Ohio 44124 USA
Tel: (800) 241-2886 or (440) 646-3599
http://www.reliance.com/automax

www.rockwellautomation.com

Corporate Headquarters
Rockwell Automation, 777 East Wisconsin Avenue, Suite 1400, Milwaukee, WI, 53202-5302 USA, Tel: (1) 414.212.5200, Fax: (1) 414.212.5201

Headquarters for Allen-Bradley Products, Rockwell Software Products and Global Manufacturing Solutions
Americas: Rockwell Automation, 1201 South Second Street, Milwaukee, WI 53204-2496 USA, Tel: (1) 414.382.2000, Fax: (1) 414.382.4444
Europe/Middle East/Africa: Rockwell Automation SA/NV, Vorstlaan/Boulevard du Souverain 36, 1170 Brussels, Belgium, Tel: (32) 2 663 0600, Fax: (32) 2 663 0640
Asia Pacific: Rockwell Automation, 27/F Citicorp Centre, 18 Whitfield Road, Causeway Bay, Hong Kong, Tel: (852) 2887 4788, Fax: (852) 2508 1846

Headquarters for Dodge and Reliance Electric Products
Americas: Rockwell Automation, 6040 Ponders Court, Greenville, SC 29615-4617 USA, Tel: (1) 864.297.4800, Fax: (1) 864.281.2433
Europe/Middle East/Africa: Rockwell Automation, Brühlstraße 22, D-74834 Elztal-Dallau, Germany, Tel: (49) 6261 9410, Fax: (49) 6261 17741
Asia Pacific: Rockwell Automation, 55 Newton Road, #11-01/02 Revenue House, Singapore 307987, Tel: (65) 6356-9077, Fax: (65) 6356-9011

Publication J-3675-6 - April 1998

Copyright © 2002 Rockwell Automation, Inc. All rights reserved. Printed in U.S.A.