

**BASIC CPU Modules and
YM-BASIC/FA Programming
Language**

IM 34M6Q22-01E

Applicable Products:

- **FA-M3 Versatile-range Multi-controller**

Model: F3BP20-0N and F3BP30-0N

Model name: BASIC CPU module

The document number and document code for this manual are as follows:

Document No.: IM 34M6Q22-01E

Document Code: DOCIM

Refer to the document number in all communications; also refer to the document number or the document code when purchasing additional copies of this manual.

◆ Important

■ About This Manual

- (1) This manual should be passed on to the end user.
- (2) Before using the module, read this manual completely to get a thorough understanding of the module.
- (3) This manual explains the functions contained in this product, but does not warrant that those will suit the particular purpose of the user.
- (4) Under absolutely no circumstances may the contents of this manual be transcribed or copied, in part or in whole, without permission.
- (5) The contents of this manual are subject to change without prior notice.
- (6) Every effort has been made to ensure accuracy in the preparation of this manual. However, should any errors or omissions come to the attention of the user, please contact the nearest Yokogawa Electric representative or sales office.

■ Safety Precautions when Using/Maintaining the Product

The following safety symbols are used on the product as well as in this manual.



CAUTION

This symbol indicates that the operator must follow the instructions laid out in this manual in order to avoid the risk of personnel injuries or fatalities or damage to the instrument. The manual describes what special care the operator must exercise to prevent electrical shock or other dangers that may result in injury or the loss of life.



Protective ground terminal

Before using the instrument, be sure to ground this terminal.



Function ground terminal

Before using the instrument, be sure to ground this terminal.



Indicates alternating current.



Indicates direct current.

- (1) The following symbols are used only in the instruction manual.

**WARNING**

Indicates that the operator must refer to the instructions in this manual in order to prevent the instrument (hardware) or software from being damaged, or a system failure from occurring.

**CAUTION**

Draws attention to information essential for understanding the operation and functions.

TIP

Gives information that complements the present topic.

SEE ALSO

Identifies a source to which to refer.

- (2) For the protection and safe use of the product and the system controlled by it, be sure to follow the instructions and precautions on safety stated in this manual whenever handling the product. Take special note that if you handle the product in a manner other than prescribed in these instructions, safety cannot be guaranteed.
- (3) If separate protection and/or safety circuits for this product or the system which is controlled by this product are to be installed, ensure that such circuits are installed external to the product.
- (4) If component parts or consumables are to be replaced, be sure to use parts specified by the company.
- (5) If the product is to be used with a system or equipment whose reliable operation is critical to the lives and safety of personnel - such as nuclear power equipment, devices using radioactivity, railway facilities, air navigation facilities, or medical equipment, consult Yokogawa Electric's sales staff.
- (6) Do not attempt to make modifications or additions internal to the product.

■ Exemption from Responsibility

- (1) Yokogawa Electric Corporation (hereinafter referred to as Yokogawa Electric) makes no warranties regarding the product except those stated in the WARRANTY that is provided separately.
- (2) Yokogawa Electric assumes no liability to any party for any loss or damage, direct or indirect, caused by the user or any unpredictable defect of the product.

■ Software Supplied by the Company

- (1) Yokogawa Electric makes no other warranties expressed or implied except as provided in its warranty clause for software supplied by the company.
- (2) Use the relevant software with one specified computer only. You must purchase another copy of the software for use with each additional computer.
- (3) Copying the software for any purpose other than backup is strictly prohibited.
- (4) Store the floppy disks (originals) of this software in a safe place.
- (5) Reverse engineering, such as decompiling of the software, is strictly prohibited.

- (6) No portion of the software supplied by Yokogawa Electric may be transferred, exchanged, or sublet or leased for use by any third party without prior permission by Yokogawa Electric.

■ General Requirements for Using FA-M3 Controllers

● Avoid installing FA-M3 controllers in the following locations:

- Where the instrument will be exposed to direct sunlight, or where the operating temperature is outside the range 0°C to 55°C (0°F to 131°F).
- Where the relative humidity is outside the range 10 to 90%, or where sudden temperature changes may occur and cause condensation.
- Where corrosive or inflammable gases are present.
- Where the instrument will be exposed to direct mechanical vibration or shock.

● Use the correct types of wire for external wiring:

- Use copper wire with temperature ratings of greater than 75°C.

● Securely tighten screws:

- Securely tighten module mounting screws and terminal screws to avoid problems such as faulty operation.
- Tighten terminal block screws with the correct tightening torque of 0.8 N·m.

● Securely fasten connectors of interconnecting cables:

- Securely fasten connectors of interconnecting cables, and check them thoroughly before turning on the power.

● Interlock with emergency-stop circuitry using external relays:

- Equipment incorporating the FA-M3 controllers must be furnished with emergency-stop circuitry that uses external relays. This circuitry should be set up to interlock correctly with controller status (stop/run).

● Ground FA-M3 controllers to an independent Japanese Industrial Standard (JIS) Class 3 Ground:

- Avoid grounding the FG terminal of the FA-M3 controller to the same ground as high-voltage power lines. Ground the terminal to an independent JIS Class 3 ground (ground resistance up to 100 Ω).

● Observe countermeasures against noise:

- When assigning inputs/outputs, the user should avoid locating AC-supplied I/O modules in the vicinity of the CPU module.

● Keep spare parts on hand:

- Stock up on maintenance parts, including spare modules, in advance.

● Discharge static electricity before operating the system:

- Because static charge can accumulate in dry conditions, first touch grounded metal to discharge any static electricity before touching the system.

● Never use solvents such as paint thinner for cleaning:

- Gently clean the surfaces of the FA-M3 controllers with a piece of soft cloth soaked in water or a neutral detergent.
- Do not use solvents such as paint thinner for cleaning, as they may cause deformation, discoloration, or malfunctioning.

● Avoid storing the FA-M3 controllers in places with high temperature or humidity:

- Since the CPU module has a built-in battery, avoid storing it in places with high temperature or humidity.
- Since the service life of the battery is drastically reduced by exposure to high temperatures, so take special care (storage temperature can be from -20° to 75°C).

● Always turn off the power before installing or removing modules:

- Turn off power to the power supply module when installing or removing modules, otherwise damage may result.

● When installing ROM packs and changing switch settings:

- In some modules you can remove the right-side cover and install ROM packs or change switch settings. While doing this, do not touch any components on the printed-circuit board, otherwise components may be damaged and modules fail.

■ Waste Electrical and Electronic Equipment



Waste Electrical and Electronic Equipment (WEEE), Directive 2002/96/EC
(This directive is only valid in the EU.)

This product complies with the WEEE Directive (2002/96/EC) marking requirement.

The following marking indicates that you must not discard this electrical/electronic product in domestic household waste.

Product Category

With reference to the equipment types in the WEEE directive Annex 1, this product is classified as a “Monitoring and Control instrumentation” product.

Do not dispose in domestic household waste.

When disposing products in the EU, contact your local Yokogawa Europe B. V. office.

◆ Introduction

■ Overview of the Manual

This instruction manual explains the specifications of the F3BP20-0N and F3BP30-0N BASIC CPU modules for the FA-M3 versatile-range multi-controller, as well as the functions and syntax of the YM-BASIC/FA programming language.

For details on the practical method of programming these modules, also refer to the BASIC Programming Tool M3 instruction manual (IM 34M6Q22-02E), YEWMAC500 instruction manual (if the module is F3MP30-0N) and other relevant documents.

■ Structure of the Manual

This manual consists of three parts, Part A, "BASIC CPU Modules," Part B, "Description of YM-BASIC/FA," and Part C, "Syntax of YM-BASIC/FA," as outlined below.

● Part A: BASIC CPU Modules (F3BP□0-0N)

Explains the specifications, basic operation and functions of the BASIC CPU modules as well as the programming tool for the modules.

● Part B: Description of YM-BASIC/FA (F3BP□0-0N)

Explains the features, basic syntax and functions (subprograms, real-time statements, data exchange with ladder sequence programs, methods of access to I/O modules, etc.) of the YM-BASIC/FA programming language.

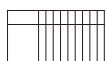
● Part C: Syntax of YM-BASIC/FA (F3BP□0-0N)

Explains the instructions (commands, subcommands, statements, functions, etc.) that can be used with YM-BASIC/FA.

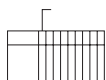
■ Notational Conventions

Symbol Marks Used in This Manual

The following symbol marks are used in Parts, "Description of YM-BASIC/FA" and "Syntax of YM-BASIC/FA," in this manual.



Refers to an instruction applicable to both the F3BP20-0N and F3BP30-0N BASIC CPU modules. This instruction applies whether a personal computer is connected or not.



Refers to an instruction applicable to both the F3BP20-0N and F3BP30-0N BASIC CPU modules. This instruction applies only when a personal computer is connected.

■ Other Instruction Manuals

In addition to this manual, refer to the following instruction manuals as necessary.

- **Refer to the following instruction manual for details on the specifications, configuration, installation/wiring, trial runs, and maintenance/inspection of the FA-M3 versatile-range multi-controller.**

- FA-M3 Versatile-range Multi-controller-Hardware (IM 34M6C11-01E)

Note: For details on the specifications of products other than the power supply modules, base modules, I/O modules, cables, and terminal block units, refer to their respective instruction manuals.

- **Refer to the following instruction manual for details on the system-wide restrictions on module installation.**

- FA-M3 Hardware Manual (IM 34M6C11-01E)

- **Refer to the following instruction manuals if your system uses an F3SP□□ sequence CPU module.**

- Sequence CPU Modules — Functions (IM 34M6P12-02E)
- Sequence CPU Modules — Instructions (IM 34M6P12-03E)

- **Refer to the following instruction manual if your system uses an F3FP36 sequence CPU module.**

- Sequence CPU Module F3FP36 (IM 34M6P22-01E)

- **Refer to the following instruction manual when creating programs using ladder language.**

- Ladder Diagram Support Program M3 (IM 34M6Q13-01E)

- **Refer to the following instruction manual when creating programs using SFC language.**

- Sequence Programming Tool for Windows POPMUSCAT (IM 34M6Q51-01E)

◆ Copyright and Trademarks Notices

■ All Rights Reserved

The copyrights of the programs, online manuals and other works contained on the CD-ROM belong to Yokogawa Electric Corporation. The online manuals are protected by PDF security from unauthorized modification; however, they can be output via a printer. When using the online manuals in a printed form, make sure the printouts are consistent with the latest versions of the manuals by checking the versions with those shown on the latest version of the CD-ROM.

No part of the online manuals and other software products included on the CD-ROM may be reproduced, or transferred, sold or distributed (including distribution through a personal computer communication network) to any third party. Nor may the online manuals be stored or recorded on videotapes or other media.

■ Trademark Acknowledgements

- Microsoft and Windows are registered trademarks of Microsoft Corporation in the USA and other countries.
- The trade and company names that are referred to in this document are either trademarks or registered trademarks of the respective companies.

FA-M3

BASIC CPU Modules and YM-BASIC/FA Programming Language

IM 34M6Q22-01E 1st Edition

CONTENTS

◆	Important	ii
◆	Introduction	vii
◆	Copyright and Trademarks Notices	ix

Part A: BASIC CPU Modules (F3BP□0-0N)

A1.	Overview	A1-1
A2.	Specifications	A2-1
A2.1	Function Specifications	A2-1
A2.2	Operating Environment	A2-2
A2.3	Model and Suffix Code	A2-2
A2.4	Components and Their Functions	A2-3
A2.5	External Dimensions	A2-4
A2.6	Attaching/Removing the BASIC CPU Module	A2-5
A2.7	System Configuration and Restrictions on Module Installation	A2-7
A2.7.1	System Configuration	A2-7
A2.7.2	Restrictions on Module Installation	A2-7
A3	Basic CPU Operation and the CPU's Functions	A3-1
A3.1	CPU's Operating Modes	A3-1
A3.2	Module Operation during Power-on/off Sequences	A3-2
A3.2.1	Module Operation during Power-on Sequence	A3-2
A3.2.2	Module Operation during Power-off Sequence	A3-2
A3.3	Module Operation during Momentary or Total Power Failure	A3-3
A3.3.1	Module Operation during Momentary Power Interruption	A3-3
A3.3.2	Setting the Mode for Detecting Momentary Power Interruption	A3-4
A3.3.3	Module Operation during Power Failure	A3-4
A3.4	Configuration Function	A3-5
A3.4.1	Setting the Sizes of User and Common Areas	A3-5
A3.4.2	Configuring the Shared Devices	A3-7
A3.5	Program Residence Function	A3-9
A3.6	ROM Writer Function	A3-10
A3.7	Access Using a Personal Computer Link	A3-11
A3.7.1	Personal Computer Link System	A3-11
A3.7.2	Accessing the Common Area	A3-12

A4.	Programming Tool	A4-1
A5.	Corrective Actions in Case of Failure	A5-1

Part B: Description of YM-BASIC/FA (F3BP□0-0N)

B1.	Standard Specifications and Features of YM-BASIC/FA	B1-1
B1.1	Standard Specifications of YM-BASIC/FA	B1-1
B1.2	Features of YM-BASIC/FA	B1-3
B2.	Basic Syntax of YM-BASIC/FA	B2-1
B2.1	Programs and Commands	B2-1
B2.2	Sentences and Lines	B2-1
B2.3	Character Set	B2-4
B2.4	Data Types	B2-5
B2.5	Constants	B2-6
B2.6	Variables	B2-8
B2.6.1	Naming a Variable	B2-8
B2.6.2	Declaration of the Type of Variable	B2-8
B2.6.3	Declaration of Variables and Their Defaults	B2-9
B2.6.4	Length of a Character-string Variable	B2-9
B2.6.5	Array Variables	B2-10
B2.7	Type Conversion	B2-12
B2.8	Expressions and Operations	B2-15
B2.8.1	Arithmetic Operation	B2-16
B2.8.2	Relational Operation	B2-17
B2.8.3	Logical Operation	B2-17
B2.9	Character-string Operation	B2-18
B2.9.1	Concatenation of Character Strings	B2-18
B2.9.2	Comparison between Character Strings	B2-18
B2.10	Functions	B2-19
B2.10.1	Intrinsic Functions	B2-19
B2.10.2	User-defined Functions	B2-19
B2.11	Priority of Operations	B2-20
B3.	Subprograms	B3-1
B3.1	Structure of a Program	B3-1
B3.2	Subprograms	B3-2
B3.3	Call of Subprograms	B3-3
B3.4	Independency of Programs	B3-4
B3.5	Arguments Transferable to Subprograms	B3-6
B3.6	Subprograms and Subroutines	B3-7
B3.7	Variables and Labels	B3-7
B4.	Real-time Statements	B4-1
B4.1	Execution Modes	B4-1

B4.2	Wait for Events (WAIT)	B4-2
B4.3	Interrupt	B4-3
B5.	Common Variables	B5-1
B5.1	Common Area	B5-1
B5.2	Basics of How to Use Common Variables	B5-2
B5.2.1	Functions of COM Statement	B5-2
B5.2.2	Clearing the Common Area	B5-3
B5.2.3	Restrictions on the Use of Common Variables	B5-3
B5.3	Statements Related to COM Statement	B5-4
B5.3.1	SUBCOM Statement	B5-4
B5.3.2	RECOM Statement	B5-6
B5.4	Data Exchange with Subprograms	B5-7
B6.	Data Exchange with a Ladder Sequence Program	B6-1
B6.1	Data Exchange between CPU Modules	B6-1
B6.1.1	Data Exchange Using Common Variables	B6-3
B6.1.1.1	Sharing of Sequence Devices	B6-3
B6.1.1.2	BASIC Common Variables and Sequence Devices ...	B6-4
B6.1.1.3	COM #S Statement	B6-5
B6.1.1.4	Example of Data Exchange	B6-11
B6.1.2	Data Exchange Using an ENTER or OUTPUT Statement	B6-14
B6.1.2.1	ENTER Statement	B6-15
B6.1.2.2	OUTPUT Statement	B6-18
B6.1.2.3	Selectable Sequence Devices	B6-21
B6.1.2.4	Example of Data Exchange	B6-25
B6.1.3	Synchronization between Programs	B6-26
B6.1.4	Precautions with Data Exchange	B6-28
B6.2	Starting/Stopping a Ladder Sequence Program	B6-30
B6.2.1	Starting/Stopping a Ladder Sequence Program	B6-31
B6.2.2	Starting/Stopping a Ladder Sequence Program Block	B6-31
B6.3	Reading the Operating Status of a Ladder Sequence Program	B6-32
B6.4	Error Codes	B6-33
B7.	Methods of Access to I/O Modules	B7-1
B7.1	Means of Access to I/O Modules	B7-2
B7.2	Slot Number and Terminal Number	B7-3
B7.3	Declaring Use of I/O Modules	B7-5
B7.4	Access to Contact I/O Modules	B7-7
B7.4.1	Contact Input Modules	B7-7
B7.4.2	Interrupt from a Contact Input Module	B7-16
B7.4.2.1	Interrupt from a Contact Input Module	B7-16
B7.4.2.2	Interrupt Input from a High-speed Input Module	B7-21
B7.4.3	Contact Output Modules	B7-24
B7.4.4	Defining the Operating Mode of a Contact Output Module	B7-31
B7.4.5	Contact I/O Modules	B7-33
B7.5	B7-35

B7.6	Contact Input/Contact Output Modules-Programming Exercise	B7-36
B7.6.1	Contact Input Modules	B7-36
B7.6.2	Contact Output Modules	B7-37
B8.	Libraries	B8-1
B8.1	What Is a Library?	B8-1
B8.2	Incorporating Libraries into a User Program	B8-1
B8.3	Program Flow	B8-2

Part C: Syntax

C1.	Syntax Usage	C1-1
C1.1	Positioning the Part "Syntax"	C1-1
C1.2	Terms Used in This Part	C1-2
C2.	List of YM-BASIC/FA Functions	C2-1
C2.1	Commands and Subcommands	C2-1
C2.2	Statements	C2-4
C2.3	Functions	C2-7
C2.4	Libraries	C2-9
C3.	Syntax	C3-1
ABS	C3-4
ALLOCATE	C3-4
APPEND (A)	C3-5
ARNAM	C3-6
ASC	C3-6
ASSIGN	C3-6
ATN	C3-7
AUTO	C3-7
BCD	C3-7
BINAND	C3-8
BINNOT	C3-8
BINOR	C3-8
BINXOR	C3-9
BIT	C3-9
BLN	C3-9
BYE	C3-10
CALL	C3-11
CALLLIB	C3-12
CHG(C)	C3-12
CHR\$	C3-13
COM	C3-14
CONT	C3-15
CONTROL	C3-16

COS	C3-16
DATA	C3-17
DATE\$	C3-17
DEF FN	C3-18
DEFAULT	C3-19
DEFINT/DEFLNG/DEFSNG/DEFDBL	C3-19
DEL	C3-20
DIM	C3-21
DISABLE	C3-22
DISP (DP)	C3-22
DISP USING (DU)	C3-22
DIV	C3-23
EDIT	C3-23
ELSE	C3-24
ENABLE	C3-25
ENABLE INTR	C3-25
END	C3-25
END WHILE	C3-26
ENDIF	C3-26
ENTER	C3-27
ERLIST	C3-28
ERRC	C3-28
ERRCE	C3-28
ERRL	C3-28
EXP	C3-29
FIND (F)	C3-29
FOR-NEXT	C3-30
FREE	C3-31
FREE	C3-31
GOSUB-RETURN	C3-32
GOTO	C3-32
HALT	C3-32
HEX\$	C3-33
HINSTR	C3-33
HLEFT\$	C3-33
HLEN	C3-34
HMID\$	C3-34
HRIGHT\$	C3-34
IF ... THEN	C3-35
IFPCNV	C3-36
IMAGE	C3-38
INICOMM3	C3-43

INIT COM	C3-43
INSTR	C3-43
INT	C3-43
IOSIZE	C3-44
LASTBIT	C3-44
LBCD	C3-44
LBINAND	C3-45
LBINNOT	C3-45
LBINOR	C3-45
LBINXOR	C3-46
LBIT	C3-46
LCOPY	C3-47
LEFT\$	C3-47
LEN	C3-47
LET	C3-48
LHEX\$	C3-48
LINKLIB	C3-49
LIST (L)	C3-50
LOAD	C3-51
LOG	C3-51
LROTATE	C3-52
LSHIFT	C3-52
MERGE	C3-53
MID\$	C3-53
MOD	C3-53
MOVE	C3-54
NAM	C3-54
NEW	C3-54
NEXT	C3-54
ON EOT/OFF EOT	C3-55
ON ERROR/OFF ERROR	C3-56
ON INT/OFF INT	C3-57
ON SEQEVTV/OFF SEQEVTV	C3-58
ON TIME/OFF TIME	C3-59
ON TIMEOUT/OFF TIMEOUT	C3-60
ON TIMER/OFF TIMER	C3-61
ON ... GOSUB	C3-62
ON ... GOTO	C3-62
OPTION BASE	C3-62
OUTPUT	C3-63
PAUSE	C3-63
PI	C3-64

PRINT (PR)	C3-65
PRINT USING (PU)	C3-66
PROG	C3-67
QUIT (Q)	C3-67
RANDOMIZE	C3-67
READ	C3-68
RECOM	C3-68
REM	C3-68
RENUM	C3-69
RESET	C3-69
RESET STATUS	C3-70
RESTORE	C3-70
RETURN	C3-70
RETURN RETRY	C3-70
RIGHT\$	C3-71
RND	C3-71
RNPAR	C3-71
ROTATE	C3-72
RUN	C3-72
SAVE	C3-73
SCRATCH	C3-75
SCRATCHP	C3-75
SCRATCHV	C3-75
SEQACTV	C3-76
SET STATUS	C3-76
SET TIMEOUT	C3-77
SETMD RES	C3-77
SETMD RUN	C3-78
SGN	C3-78
SHIFT	C3-79
SIN	C3-79
SPC	C3-79
SQR	C3-79
STATUS	C3-80
STEP	C3-81
STOP	C3-81
STR\$	C3-82
SUB	C3-82
SUBCOM	C3-82
SUBEND	C3-83
SUBEXIT	C3-83
SUBEXIT RETRY	C3-83

SWAP	C3-83
TAN	C3-84
TIME\$	C3-84
TIMEMS	C3-84
TRACE	C3-85
TRACEP	C3-86
TRACEV	C3-87
TRANSFER	C3-88
VAL	C3-89
WAIT	C3-90
WHILE/END WHILE	C3-91
C4. Error Code List	C4-1
C4.1 YM-BASIC/FA Error Codes	C4-1
C4.2 Detail Error Codes	C4-6
Appendix 1. Listing of Internal Codes	App1-1
Appendix 1.1 Data Formats	App1-1
Appendix 1.2 Character Code Format	App1-3
Appendix 1.3 Listing of Alphanumeric Character Codes	App1-4
Appendix 2. Listing of Reserved Words	App2-1
Appendix 3. Listing of MS-DOS Special Editing Functions	App3-1

Revision History

A1. Overview

■ Product Overview

The F3BP20-0N and F3BP30-0N BASIC CPU modules use high-speed real-time BASIC, i.e. YM-BASIC/FA, that is in use with the FA500 controller and YEWMAC computer* and has received favorable recognition within the industry. The modules are especially useful where there is a need for communication or information handling.

* The FA500 is an intelligent programmable controller and the YEWMAC is an FA computer developed by Yokogawa Electric Corporation.

■ Features

- Especially useful with a communication module that cannot be controlled using ladder sequence, or when advanced computing is required.
- One of these modules can be installed in one of slots 1 to 4 of the main unit. Since the module does not need a sequence CPU module to be used, it is possible to build a BASIC controller.
- Can have direct access to I/O modules.
- Can exchange data with the ladder sequence, and can also be synchronized with the ladder sequence by events.
- Structured programming is possible with subprograms.
- Access to the common data is possible using the PC link module, etc.
- Can be equipped with a ROM pack to execute ROM-based operation or store programs or common data.
- Allows programs to be developed or debugged on a general-purpose personal computer.

A2. Specifications

A2.1 Function Specifications

Table A2.1 Specifications

Item	Specifications	
	F3BP20-0N	F3BP30-0N
Programming language	YM-BASIC/FA	
Method	Interpreter (with pre-run function)	
Number of tasks	1	
Program capacity	120 KB	510 KB
Shared device	Shared registers (R): 1024 maximum (Shared relays and extended shared relays/registers cannot be used.)	
Self-diagnosis function	Memory failure, CPU failure, power failure, etc.	
Other functions	Configuration function (setting of the sizes of user and common areas) Program residence function Error log saving function Program development and debugging function Date/time function (year/month/day/hour/minute/second) Access to common data (writing/reading) using PC link module Writing of program data to ROM	
Maximum number of modules installed	1/unit	
Current consumption	200 mA (5 V DC)	
External dimensions	28.9 (W) × 100 (H) × 83.2 (D) mm ^(Note)	
Weight	105 g	

TA020101.EPS

Note: Excluding protrusions. (See the dimensional figures for more details.)

A2.2 Operating Environment

The following table summarizes requirements for CPU modules that can be used in combination with the F3BP□0-0N BASIC CPU module.

Table A2.2 Applicable CPU Modules and Their Revisions

CPU Module	Revision
F3SP21, F3SP25, F3SP35	No restrictions due to revision.
F3FP36	No restrictions due to revision.

TA020201.EPS

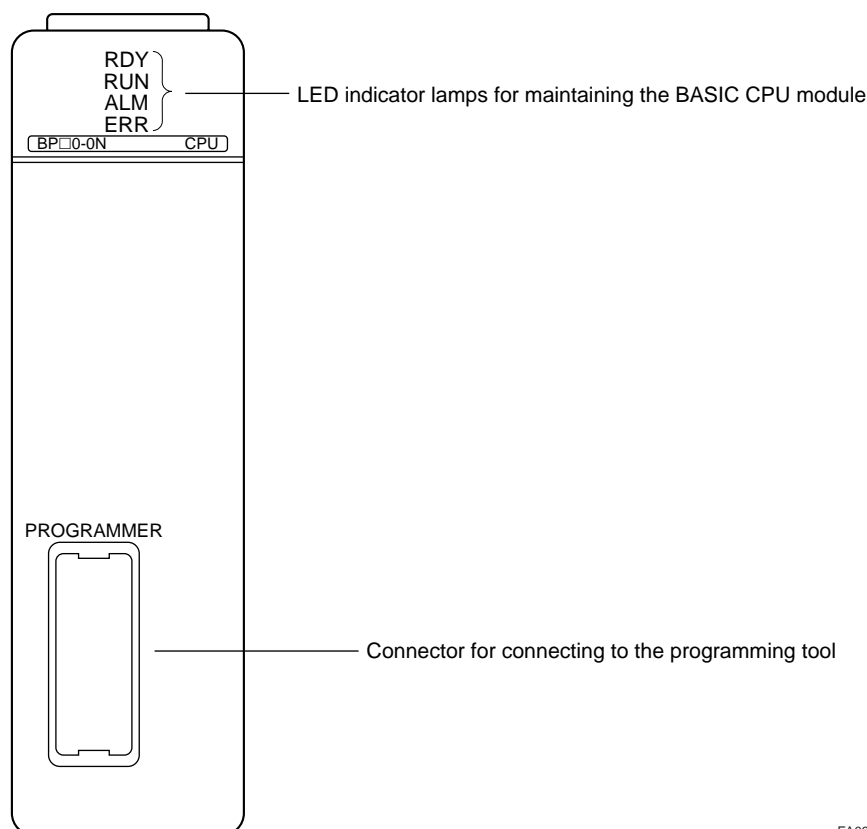
A2.3 Model and Suffix Code

Table A2.3 Model and Suffix Code

Model No.	Suffix Code	Style Code	Option Code	Remarks
F3BP20	-0N	120-KB memory
F3BP30	-0N	510-KB memory

TA020301.EPS

A2.4 Components and Their Functions



FA020401.EPS

Figure A2.1 Components and Their Functions

● LED Indicator Lamps for Maintaining the BASIC CPU Module

Failures are indicated by these LED indicator lamps, as classified by the failure level, in the upper section of the BASIC CPU module's front panel.

Table A2.4 Status Indication by LED Indicator Lamps

LED Indicator Lamp	Meaning
RDY (Ready) - green	● Serious failure if unlit: The key hardware is disabled. Example: CPU failure, memory failure
RUN (Run) - green	● Running when lit: The user program is running.
ALM (Alarm) - yellow	● Minor failure when lit: The BASIC CPU module is abnormal, though the user program can still be run. Example: Problem with power supply, I/O module failure, communication failure - Or - ● Debug mode when lit: The debug mode is in progress. (This indicator lamp comes on when a personal computer is connected to the BASIC CPU module and is developing or debugging programs.)
ERR (Error) - red	● Moderate failure when lit: The user program cannot be started or run any further. Example: Program failure, I/O module failure, instruction processing failure

TA020401.EPS

A2.5 External Dimensions

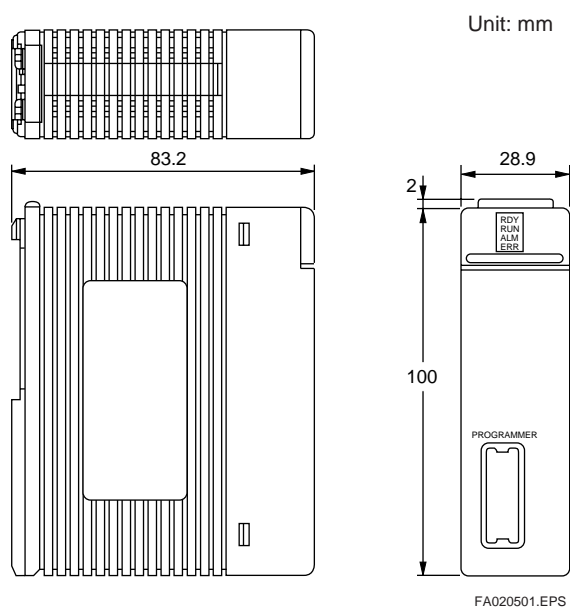


Figure A2.2 External Dimensions

A2.6 Attaching/Removing the BASIC CPU Module

■ Attaching the Module

Figure A2.3 shows how to attach the BASIC CPU module to a base module. Hook the BASIC CPU module onto one of the protruding tabs at the bottom of the base module. Push the top of the communication module toward the base module in the direction indicated by the arrow in the figure. Push in the BASIC CPU module until the lock clip on top fully engages the opening at the top of the base module. Before attaching or removing the BASIC CPU module, be sure to turn off the power to the module.

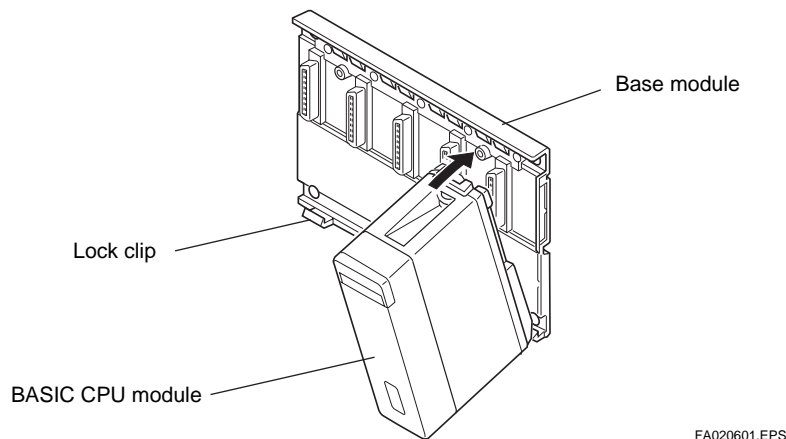


Figure A2.3 Attaching the BASIC CPU Module



CAUTION

When attaching the BASIC CPU module, exercise care to avoid bending any of the connector pins on the back of the module. Do not force the module onto the base module before they are securely connected. Otherwise, the pins will be bent, resulting in the malfunction of the BASIC CPU module. If the pins are bent during installation of the BASIC CPU module, the message "Module Installation Error" will be given during self-diagnosis.

■ Removing the Module

Remove the BASIC CPU module from the base module in reverse order from the way it is attached. Press in on the lock clip to disengage the module, and then pull the module toward you and lift it out.

■ Attaching the Module in Areas of Intense Vibration

In anticipation of possible exposure to intense mechanical vibration, the BASIC CPU module is designed to be securely fastened onto the base module with a screw. Figure A2.4 shows how to fix the BASIC CPU module with a screw. Insert the screw specified below into the hole on the top of the module. Using a Phillips screwdriver, fasten the screw to the base module. Since the screwdriver needs to be somewhat angled to do this, allow a clearance of at least 80 mm between the BASIC CPU module and the duct.

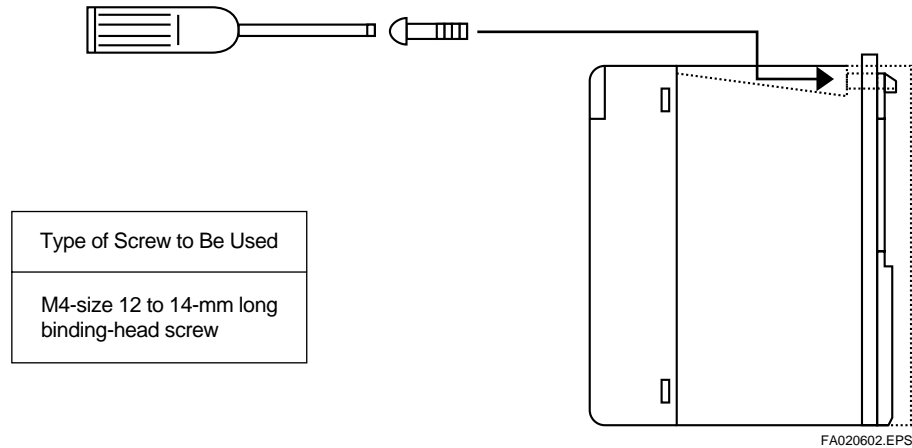


Figure A2.4 Fixing the Module with a Screw



CAUTION

When fixing the BASIC CPU module, avoid overtightening the screw.

A2.7 System Configuration and Restrictions on Module Installation

A2.7.1 System Configuration

For details on the system configuration, see the “FA-M3 Hardware Manual” (IM 34M6C11-01E) or “Sequence CPU Instruction Manual — Functions” (IM 34M6P12-02E)

A2.7.2 Restrictions on Module Installation

For details of the Restrictions on module installation, see the “FA-M3 Hardware Manual” (IM 34M6C11-01E).



WARNING

If you install the BASIC CPU module in the 5th or a higher-numbered slot and turn on the power, the CPU memory is completely cleared and the module reverts to the state in which it was shipped.



CAUTION

- [illegible]

A3 Basic CPU Operation and the CPU's Functions

A3.1 CPU's Operating Modes

The CPU has two operating modes: the REAL mode and the DEBUG mode.

■ REAL Mode

The REAL mode is the mode for practical system operation. If you specify the program in question as a resident program, it starts automatically when the FA-M3 controller is turned on. For details on the program residence function, see Section A3.5, "Program Residence Function." When the program is running in REAL mode, the RDY and RUN indicator lamps are lit. When the program stops, only the RUN indicator lamp goes out.

■ DEBUG Mode

The DEBUG mode is the mode for developing and debugging programs with a personal computer connected to the BASIC CPU module. It is also referred to as the command input mode because you can input commands and statements from a keyboard. When the BASIC CPU module is in DEBUG mode, the RDY and ALM indicator lamps are lit. When you run the program in DEBUG mode, the RUN indicator lamp also comes on.

If you start the BASIC Programming Tool M3 and the program stops, the BASIC CPU module changes from the REAL mode to the DEBUG mode. If you quit the BASIC Programming Tool M3, the BASIC CPU module returns to REAL mode from DEBUG mode. It is also possible to enable the program specified as a resident program to run automatically in REAL mode (BYE&RUN mode) at the end of the DEBUG mode.

The LED indicator lamps turn on or off in the combinations shown in the following table, depending on the operating mode selected.

Table A3.1 States of LED Indicator Lamps During Operation Modes

Operating Mode LED Indicator Lamp	REAL Mode		DEBUG Mode	
	Program in progress	Program at a stop	Program in progress	Program at a stop
RDY	○	○	○	○
RUN	○	●	○	●
ALM	●	●	○	○
ERR	●	●	●	●

TA030101.EPS

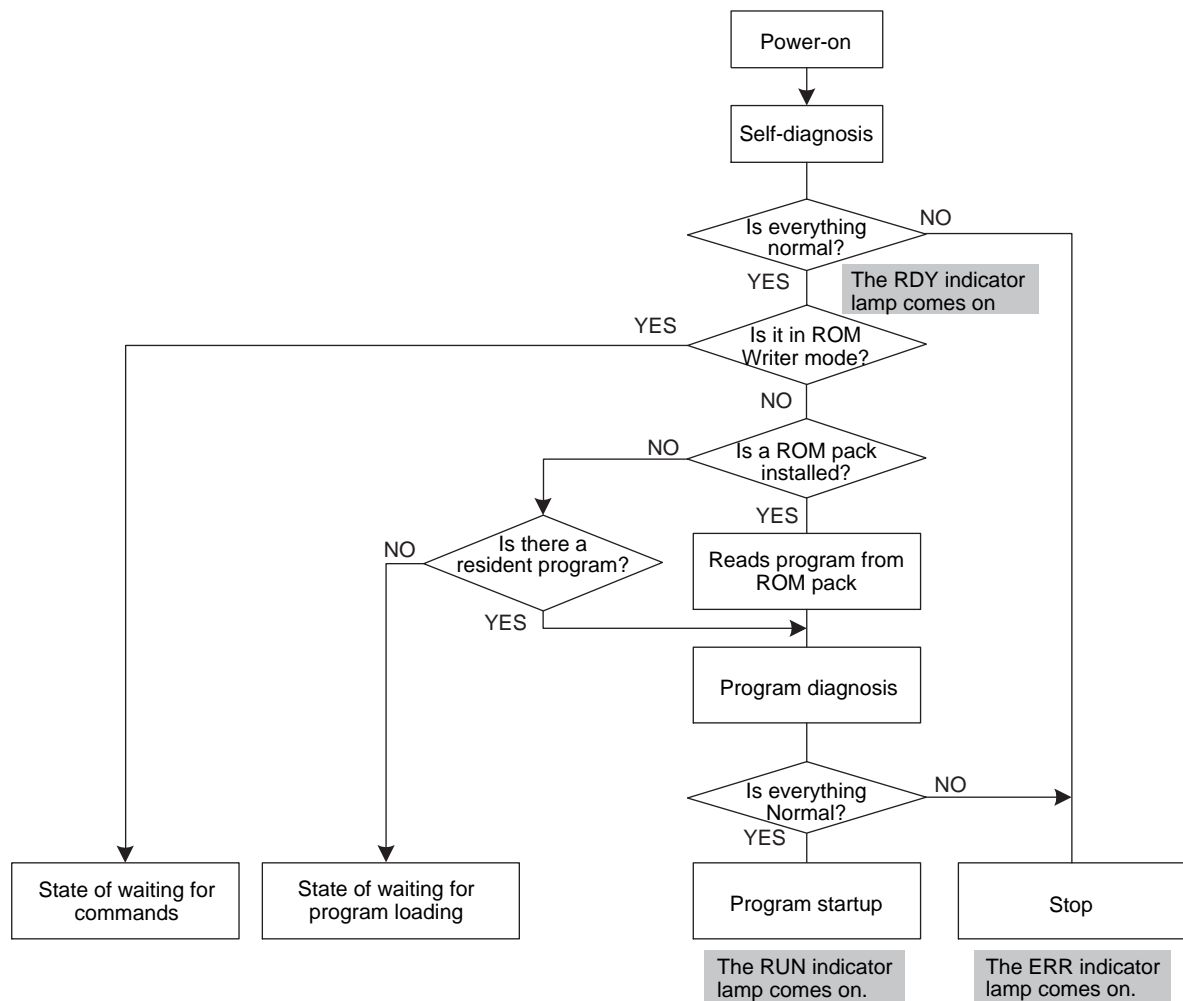
○: Lit; ●: Unlit

A3.2 Module Operation during Power-on/off Sequences

A3.2.1 Module Operation during Power-on Sequence

When the power is turned on, the BASIC CPU module initializes its settings to enable the program to be run. During initialization the module diagnoses itself for such faults as memory failure or CPU failure to verify that the CPU hardware is working normally. When no failure is identified and the program specified as a resident program is in storage, the module begins running the program from its starting point.

If a ROM pack is already installed, the module reads the program from the ROM pack to start operation. If in ROM Writer mode, the module does not read any program from the ROM pack and goes into a state of waiting for commands, such as a write-to-ROM command, without executing any programs.



FA030201.EPS

Figure A3.1 Module Operation during Power-on Sequence

A3.2.2 Module Operation during Power-off Sequence

When the power is turned off, the BASIC CPU module records the date and time to the error log file within the CPU and shuts down.

A3.3 Module Operation during Momentary or Total Power Failure

A3.3.1 Module Operation during Momentary Power Interruption

There are two modes for detecting a momentary power interruption: the standard mode and the immediate detection mode. The way the BASIC CPU module behaves in cases of momentary power interruption differs depending on the mode selected. Note that the immediate detection mode can only be selected from the CPU Configuration menu when an F3PU10-0N, F3PU20-0N or F3PU26-0N power supply module is used.

■ Standard Mode

If a momentary power interruption occurs, the BASIC CPU module records the date and time to the error log file within the CPU. The CPU stops processing until the module recovers from the momentary power interruption. This in turn causes a delay in timer updating. After recovery from the momentary power interruption, the CPU resumes operation from where it stopped processing.

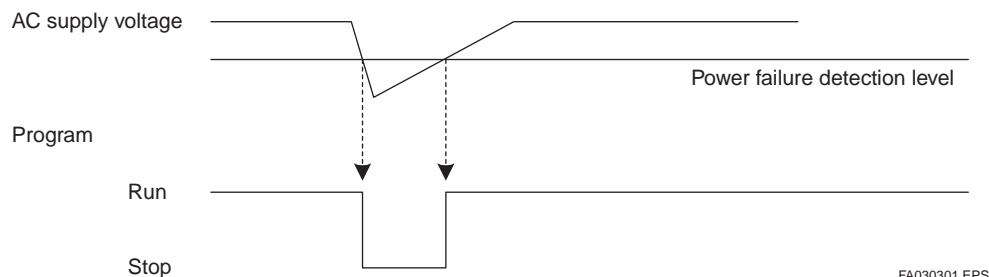


Figure A3.2 Module Operation during Momentary Power Interruption

FA030301.EPS

■ Immediate Detection Mode

If a momentary power interruption occurs, the BASIC CPU module records the date and time to the error log file within the CPU. The CPU stops processing until the module recovers from the momentary power interruption. At this point, the CPU turns off the external output to actuate the FAIL contact. After recovery from the momentary power interruption, the CPU undergoes a reset-start process to resume operation from the starting point of the program.

A3.3.2 Setting the Mode for Detecting Momentary Power Interruption

The BASIC CPU module can be set to either the standard or immediate detection mode. The module is initially set to the standard mode. For more details on each of these modes, see the "FA-M3 Hardware Manual" (IM 34M6C11-01E). Use the BASIC Programming Tool M3 for Windows to define the mode for detecting momentary power interruption. For more details on how to set the mode, see the "BASIC Programming Tool M3 for Windows" instruction manual (IM 34M6Q22-02E).



WARNING

If you carry out CPU configuration, all of the existing programs and their data stored in the CPU will be lost. It is therefore advisable that you save the programs as necessary before you carry out CPU configuration.

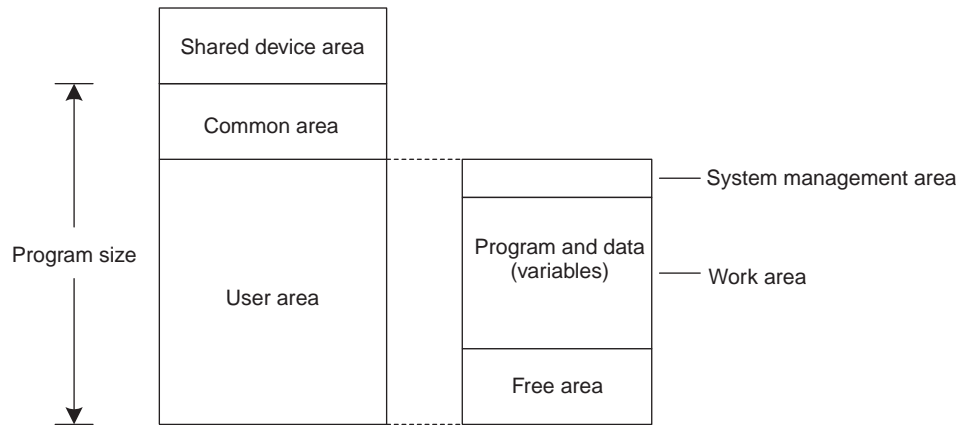
A3.3.3 Module Operation during Power Failure

If a power failure occurs, the BASIC CPU module behaves in the same way as when it is turned off.

A3.4 Configuration Function

A3.4.1 Setting the Sizes of User and Common Areas

The F3BP20 and F3BP30 BASIC CPU modules have the program sizes of 120 KB and 510 KB, respectively. Each program is separated into the user area and common area, and the size of each area can be defined or changed by the user.



FA030401.EPS

Figure A3.3 Areas of a BASIC Program

These areas are used in the ways described below.

● User Area

This area is used as a work area for a user-created program and variables (data) declared in the program, as well as for enabling the program to run. The user area is initialized with the NEW command.

- System Management Area

This area is allocated for BASIC management and occupies approximately 6 KB. If you execute the FREE command when there are no programs, the system shows the size of the free area. Thus, you can learn the maximum size of a program you can create.

- Program and Data (Variables)

This area is allocated to a user program and variables or array variables declared in the program. Since an area for variables is secured during program execution, the size of the free area differs between the start and end of program execution.

- Free Area

This area is used to run a BASIC program. Although the required size is dependent on the type of program, allocate a size of approximately 10 KB to this area.

● Common Area

This area is not initialized even when the BASIC CPU module is turned on or off. Use this area when you want to save your data or exchange data between the main program and subprogram. The common area is initialized by an INIT COM statement.

For more details on how to use the common area and on other relevant statements, see Section B5, "Common Variables," in Part B, "Description of YM-BASIC/FA," in this manual. Use the BASIC Programming Tool M3 for Windows to define or change the sizes of the user and common areas within the following ranges. For more details on this procedure, see the "BASIC Programming Tool M3 for Windows" instruction manual (IM 34M6Q22-02E).

Table A3.2 Sizes of User and Common Areas

Model	Area	Default	Configurable Range	Maximum Setpoint*
F3BP20	User area	64 KB	16 to 120 KB	120 KB
	Common area	4 KB	0 to 104 KB	
F3BP30	User area	64 KB	16 to 510 KB	510 KB
	Common area	4 KB	0 to 256 KB	

TA030401.EPS

* Means the total sum of the sizes of user and common areas.



WARNING

If you carry out CPU configuration, all of the existing programs and their data stored in the CPU will be lost. It is therefore advisable that you save the programs as necessary before you carry out CPU configuration.

A3.4.2 Configuring the Shared Devices

In a system comprising two or more CPU modules, the BASIC CPU module can share data with other CPUs. Data that can be shared is that of shared registers (up to 1024 units) defined in the data areas that are shared among the CPUs. It is not possible for the CPUs to share data, however, using the shared relays, extended shared relays or extended shared registers of any sequence CPU module or IBM PC-compatible CPU module. Each local CPU can read from/write to its own local shared register area only. Other CPUs can only read from that shared register area. Each shared register area is not initialized even if the CPU module is turned on or off. To initialize shared register areas, use the INICOMM3 standard library. Before each group of shared registers can be used, they must be allocated to the local and remote CPUs using the CPU Configuration menu.

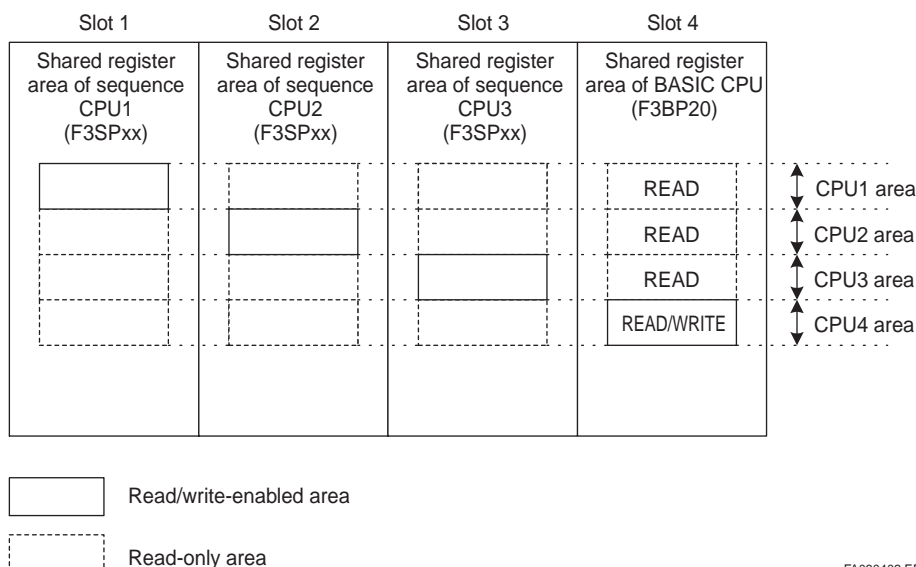


Figure A3.4 Shared Registers

The BASIC CPU module installed in slot 4 of the system shown in Figure A3.4 has read/write access to the CPU4 area only. The module has only read access to the CPU1, CPU2 and CPU3 areas.

For details on how the BASIC CPU module exchanges data with a sequence CPU module using shared relays and registers, see Chapter B6, "Data Exchange with Ladder Sequence Program," in Part B, "Description of YM-BASIC/FA," later in this manual.

Use the BASIC Programming Tool M3 for Windows to configure shared relays and registers within the following ranges. For more details on this procedure, see the "BASIC Programming Tool M3 for Windows" instruction manual (IM 34M6Q22-02E).

Table A3.3 Configurable Ranges of Shared Relays and Registers

Device Type	Configurable Range	Default
Shared relays*	0 to 2048 units (can be allocated freely to CPU1 to CPU4)	0
Shared registers	0 to 1024 units (can be allocated freely to CPU1 to CPU4)	0

TA030402.EPS

* The configuration of these relays should be set in compliance with that of other CPUs. Data cannot be shared with any sequence CPU using shared relays.



WARNING

If you carry out CPU configuration, all of the existing programs and their data stored in the CPU will be lost. It is therefore advisable that you save the programs as necessary before you carry out CPU configuration.



CAUTION

- Information on the allocation of shared relays and registers is managed separately by each individual CPU. For this reason every two CPUs that exchange data with each other must share the same information on the allocation of shared relays and registers. If the information differs between the two CPUs, data exchange may not be carried out correctly. The configuration of shared relays should be set in compliance with that of other CPUs.
- Shared devices in a sequence CPU are refreshed in asynchrony with scanning. For this reason the simultaneity of data is not guaranteed. If necessary, refresh them using an application program.

A3.5 Program Residence Function

The BASIC CPU module allows debugged or tuned programs to be specified as those resident within the module. If a program resides in the module's user area, it is executed automatically when:

- the module is turned on;
- the module undergoes a reset-start sequence; or
- the module is set in BYE&RUN mode and DEBUG mode is quit.

By default, the program is specified as a non-resident program. The program disappears from the main memory when the BASIC CPU module is turned off, subjected to a reset-start sequence, or the DEBUG mode is quit.

Use the BASIC Programming Tool M3 for Windows to specify the program as a resident program. For more details on this procedure, see the "BASIC Programming Tool M3 for Windows" instruction manual (IM 34M6Q22-02E) or the SETMD RES and SETMD RUN commands in Part C, "Syntax of YM-BASIC/FA," later in this manual.



CAUTION

- If no program is loaded in the user area, executing a command for specifying the program as a resident program results in an error.
- If any program is resident, any command or statement for loading other programs to the user area results in an error. Before loading other programs to the user area, cancel the program's resident status.
- If you create a program and specify it to be resident in the user area without having saved it at least once, it will be named '\$\$\$\$\$\$\$\$'. Note that programs cannot be saved under this name.

A3.6 ROM Writer Function

The BASIC CPU module can be equipped with a ROM pack (RK30-0N) to save programs or common-area data in the ROM pack, delete saved programs or data, or run programs stored in the ROM pack. Such tasks as writing programs to a ROM pack, which are usually done by a regular ROM writer, can also be performed using the BASIC CPU module. These functions equivalent to those of a ROM writer are referred to as ROM writer functions. The ROM writer functions are enabled in the ROM Writer mode, rather than the normal operating mode. Programs/data can be written to or deleted from the ROM pack when the BASIC CPU module is in ROM Writer mode. Since ROM Writer mode is retained even if the power is turned off, the BASIC CPU module does not read programs from the ROM pack when the power is turned back to on. Note that when in ROM Writer mode, the BASIC CPU does not come into normal operation.

There are two ROM writer functions:

● Writing to ROM

- This function writes the BASIC program in the BASIC CPU and data in the common area to the ROM. It is also possible to specify whether to have the common-area data reside in the ROM or not.
- This function can also be used to write the same program to two or more ROMs. This can be achieved by transferring the BASIC program to the ROM pack only once and then replacing the ROM pack with another.

● Deleting from ROM

- This function deletes the contents of the ROM pack.

Table A3.4 Available Models of ROM Packs

Model	RK10-0N	RK30-0N	RK50-0N
F3BP20	N/A	120 KB	N/A
F3BP30	N/A	N/A	510 KB

TA030601.EPS

Use the BASIC Programming Tool M3 for Windows to move to or cancel the ROM Writer mode and write/delete programs or data to and from the ROM. For more details on this procedure, see the "BASIC Programming Tool M3 for Windows" instruction manual (IM 34M6Q22-02E).



CAUTION

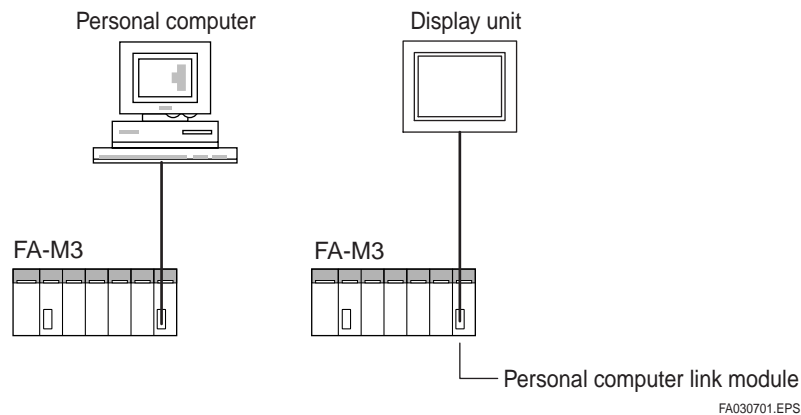
- A program residing in a ROM pack should have been completely debugged and tuned. It is not possible to edit any program or data residing in a ROM pack, for example, by making a partial correction to it.
- If equipped with a ROM pack and turned on without being set in the ROM Writer mode, the BASIC CPU module reads contents from the ROM pack even if the user program is specified as a resident program.
- The user program has already been specified as a resident program if it is read and executed from the ROM pack at power-on.
- If you write a program you have created to the ROM pack by setting the module in the ROM without having saved it at least once, it will be named '\$\$\$\$\$\$\$\$'. Note that programs cannot be saved under this name.
- When in ROM Writer mode, the BASIC CPU module cannot execute any program. In addition, any specified BYE&RUN mode is ignored.

A3.7 Access Using a Personal Computer Link

A3.7.1 Personal Computer Link System

You can have reference and access settings to common-area data and various other types of information, through such external equipment as a personal computer or display unit connected to a module (e.g., personal computer link module) with the personal computer link function. (However, unlike with a sequence CPU module, a program cannot be loaded, saved, started or stopped. In addition, you cannot access data other than that of the common-area.)

Note that there is no need to create a new program for the purpose of this communication.



FigureA3.5 Example of a Personal Computer Link System

For details on the commands for personal computer link and the responses to these commands, see the “Commands for Personal Computer Link” instruction manual (IM 34M6P41-01E).

A3.7.2 Accessing the Common Area

Access to the common area of the BASIC CPU module is obtained as specified below.

■ Word-based Access

To have word-based access to the common area, specify the device as D****. D00001 represents the starting point of the common area, where one device is equivalent to one word. For integer-type variables, one word is allocated to each integer variable. If a variable of any other type is used for the common area, the variable must be converted to the internal data format on the reading/writing side.

Note: For a sequence CPU module, the device "D" means a data register.

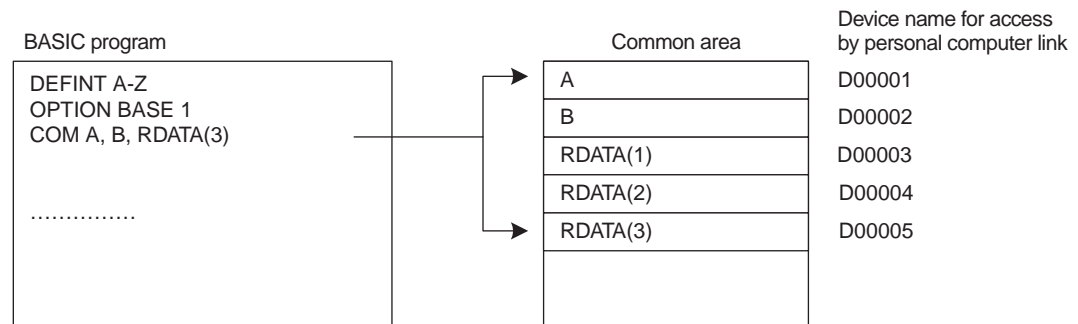


Figure A3.6 Word-based Access

FA030702.EPS

■ Bit-based Access

To have bit-based access to the common area, specify the device as `*****.I00001` represents the starting point of the common area, where one device is equivalent to one bit. For integer-type variables, 16 devices are allocated to each integer variable. Care must be taken on the reading/writing side to ensure that the internal data format is consistent.

Note: For a sequence CPU module, the device “I” means an internal relay.

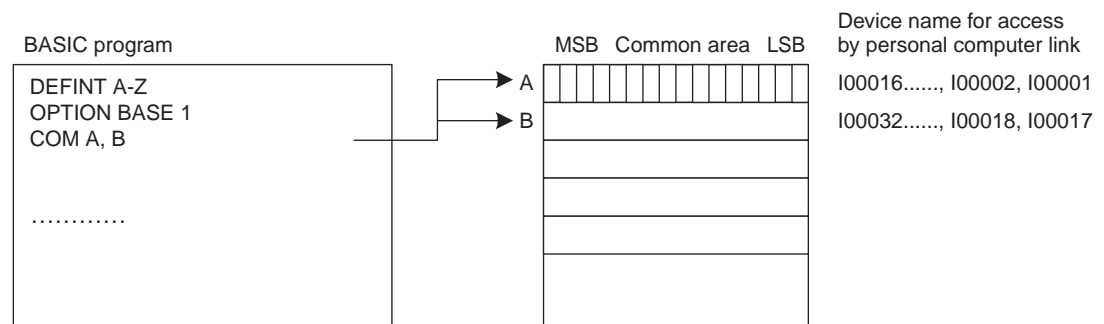


Figure A3.7 Bit-based Access

FA030703.EPS

■ Accessible Range of Common Area

In the case of word- or bit-based access to the common area, the accessible range of the common area differs depending on the type of module (e.g., a personal computer link module) having the personal computer link function. The accessible ranges are summarized in the following table.

Table A3.5 Accessible Range of the Common Area

	F3BP20		RK50-0N	
	Word-based Access	Bit-based Access	Word-based Access	Bit-based Access
Personal computer link function of sequence CPU	D00001 to D53248	I00001 to I99999 ^{*1}	D000001 to D131072	I00001 to I99999 ^{*1}
Ethernet interface module	D00001 to D53248	I00001 to I99999 ^{*1}	D000001 to D131072	I00001 to I99999 ^{*1}
Other modules (e.g., personal computer link module)	D00001 to D53248	I00001 to I99999 ^{*1}	D00001 to D99999 ^{*2}	I00001 to I99999 ^{*1}

TA030701.EPS

*1 12499 bytes (approximately 12 KB) from the starting point of the common area

*2 199998 bytes (approximately 195 KB) from the starting point of the common area

A4. Programming Tool

The BASIC CPU module allows programs to be developed and debugged using a programming tool that runs on a personal computer.

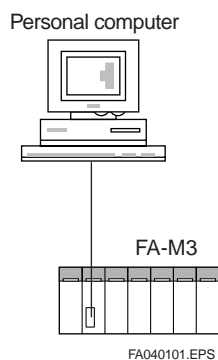


Figure A4.1 Program Development on a Personal Computer

For details on the programming tool that runs on a personal computer, see the “BASIC Programming Tool M3 for Windows” instruction manual (IM 34M6Q22-02E).

■ BASIC Programming Tool M3 for Windows

● Model Number and Suffix Code

Model Number	Suffix Code	Style Code	Option Code	Description
SF560	BASIC Programming Tool M3 for Windows
	-ECW	Microsoft Windows 98 Microsoft Windows 95 Microsoft Windows NT Workstation 4.0
		...	/ED	Microsoft Windows 98 Microsoft Windows 95 Microsoft Windows NT Workstation 4.0 Supplied with instruction manuals

TA040101.EPS

● Operating Environment

Item	Specifications
OS	Microsoft Windows 98 Microsoft Windows 95 Microsoft Windows NT Workstation 4.0 with Service Pack 3 or later
Computer model	IBM PC-compatible
Supply medium	CD-ROM
CPU	75 MHz or faster Pentium
Free memory space	16 MB minimum
Free hard disk space	30 MB minimum
Communication requirements	Interface: RS-232-C; Synchronization: start-stop; Baud rate: 9600/19200 bps
Supported CPU module	F3BP20-0N and F3BP30-0N

TA040102.EPS

In BASIC Programming Tool M3 for Windows, the commands, subcommands, and key operations [ESC], [CTRL]+[S], [CTRL]+[P], [CTRL]+[C], and the like, are operated from the menu bar, toolbar or edit window.

TIP

A 166-MHz or faster CPU is recommended. If the CPU is slower, the tool may operate very slowly.

If the BASIC CPU module fails, get a full understanding of the current situation, consider the relationship of the module to other equipment and the chance of the failure reoccurring. Take corrective actions in accordance with the flow chart shown in Figure A5.1.



B1. Standard Specifications and Features of YM-BASIC/FA

B1.1 Standard Specifications of YM-BASIC/FA

The standard specifications of the YM-BASIC/FA programming language are as follows.

- Method: Interpreter (with pre-run function)
- Number of tasks: 1
- Character code: See Appendix 1., "Table of Internal Codes," later in this manual.
- Data format: See Appendix 1.1, "Data Format," later in this manual.
- Program execution mode: DEBUG mode (command input mode) and REAL mode
- Size of user area:

Controller Model	CPU Model Number	Size of User Area
FA-M3	F3BP20	120 KB
	F3BP30	510 KB

TB010101.EPS

The sizes of user areas used with BASIC programs and common variables.

Area	Size	
	F3BP20	F3BP30
BASIC program	16 to 120 KB	16 to 510 KB
Common variable	0 to 104 KB	0 to 256 KB

TB010102.EPS

The total sum of each area must be smaller than the size of the user area.

- Structuring: Possible with subprograms (no limitation on their number)
- I/O module support: Contact inputs
Contact outputs
Analog inputs
Analog outputs
Communication control, etc.
See Chapter B7., "Accessing to I/O Modules," later in this manual for more details.

- Data types:

Integer: –32768 to 32767 (2 bytes internally)

Long integer: –2147483648 to 2147483647 (4 bytes internally)

Single-precision real number: An approximate range of 2.7×10^{-20} to 9.2×10^{18} for absolute values. The number of significant digits is approximately 7 (4 bytes internally).

Double-precision real number: An approximate range of 2.7×10^{-20} to 9.2×10^{18} for absolute values. The number of significant digits is approximately 16 (8 bytes internally).

Character string: 512 bytes maximum

- Arrays:

One-dimensional or two-dimensional arrays are available.

The number of elements allowed in a single array is:

32767 for one-dimensional arrays; and

32767×32767 for two-dimensional arrays.

- Operators:

(,), ^, +, –, NOT, *, /, =, <, >, >=, <=, <>, AND, OR, and EXOR

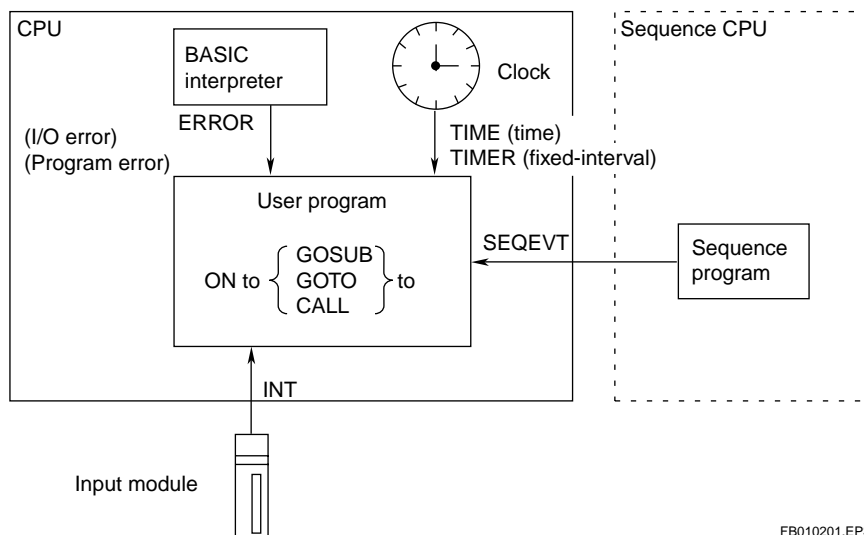
For more details on the data types, see Appendix 1.1, “Data Format,” later in this manual.

B1.2 Features of YM-BASIC/FA

The YM-BASIC/FA has the following features.

■ On-line Real-time Processing

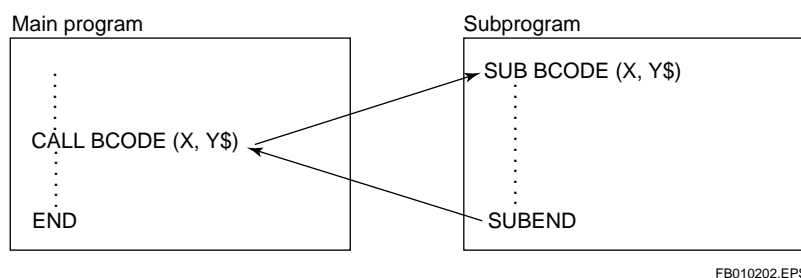
The YM-BASIC/FA supports a wide choice of interrupts. It is a high-speed BASIC language designed so that a program can promptly respond to external events. Using the YM-BASIC/FA, you can easily create on-line, real-time programs.



■ Block Structure of Programs

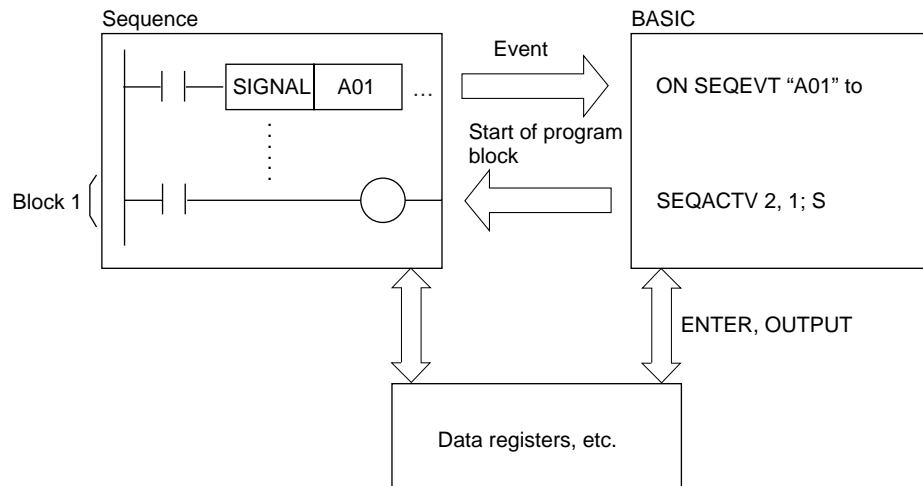
The YM-BASIC/FA allows subprograms to be used with the BASIC CPU module.

With subprograms, you can control variables, line numbers and labels separately. This feature further improves the developability, maintainability and recyclability of programs. It is also possible to develop a main program and a subprogram separately and then combine them using an APPEND command.



■ Simple Combination with a Sequence Program

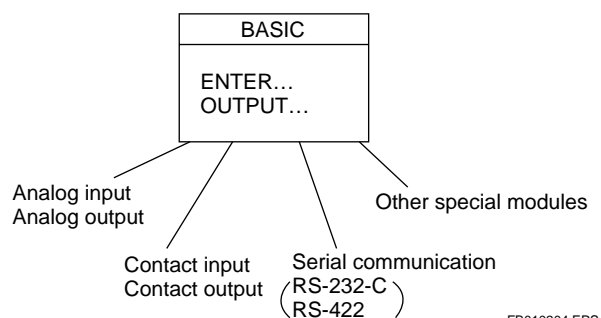
A ladder sequence program can be synchronized with a BASIC program using a SIGNAL command, or ON SEQV, ENTER or OUTPUT statement. In addition, data exchange can be easily achieved by reading from/writing to sequence devices using ENTER and OUTPUT statements. Shared registers can also be used for data exchange. For more details on data exchange, see Chapter B6., "Data Exchange with a Ladder Sequence Program," later in this manual. A sequence program block can be started using a SEQACTV statement.



FB010203.EPS

■ I/O Support

The BASIC language can be used to have access to a serial communication module, contact I/O module, analog I/O module, etc. Input and output can be achieved easily by using ENTER and OUTPUT statements, respectively.



FB010204.EPS

B2. Basic Syntax of YM-BASIC/FA

B2.1 Programs and Commands

If you type a statement according to the BASIC syntax and press the return key (↵) without adding a line number to the statement when BASIC is active (the BASIC prompt is on display), the statement is regarded as a command and executed immediately.

Example: BSC: PRINT "ABC" ← Command input
 ↑ Prompt
 ABC ← Result of execution

FB020101.EPS

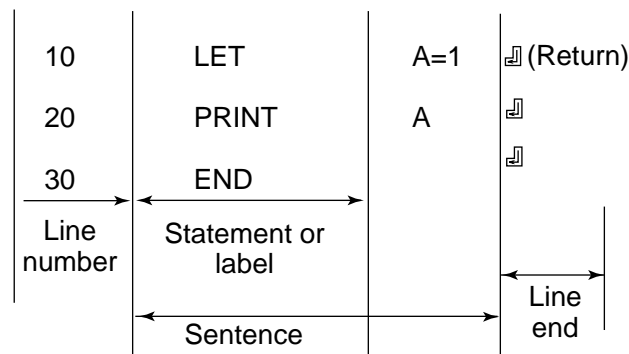
If lines are preceded by numbers (referred to as line numbers), the lines are stored as a program and executed by a RUN command.

BSC: 10 PRINT "ABC" } Program
 BSC: 20 END }
 BSC: RUN ← RUN command
 ABC ← Result of execution

FB020102.EPS

B2.2 Sentences and Lines

Each BASIC program consists of several lines. Each line consists of a line number, sentence that begins with a statement or label, and a line end.



FB020201.EPS

■ Length of a Line (Maximum Number of Characters That Can Be Typed)

The maximum number of characters that can be typed is:

Line number (5 digits maximum) + One space + Statement (246 characters maximum)

A line is terminated by pressing the return key or typing the allowable maximum number of characters.



CAUTION

— Precautions when Using a Commercially Available Editor —

Avoid exceeding the line number of 65535 when inputting a program using a commercially available editor. If a program contains line numbers greater than 65535, the extra lines are deleted when the BASIC Programming Tool M3 for Windows is used.

■ Line Numbers

Line numbers are whole numbers from 1 to 65535, and show the order in which lines in a program are stored in memory. A program must not include the same line number. If you type the same line number twice, the new line replaces the previous line with the same line number. Lines are executed in the ascending order of their line numbers. Line numbers are used in a sentence or command in order to branch the program or for editing purposes.

■ Labels

Labels can be used instead of line numbers in order to branch a program.

Example: When no labels are used

```
100 GOTO 200
:
200 PRINT "A"
```

Example: When labels are used

```
100 GOTO LP@
:
200 LP@ PRINT "A"
```

Labels are described using an upper-case alphanumeric character string (7 characters maximum), preceded by a letter and followed by an At sign (@).

A label follows a line number and is placed in the beginning of a sentence. The label is then followed by a line end or space, and then another statement. If it is followed by another statement, there is no need for appending a colon to the end of the label to form a compound sentence.

Example:

```
100 A@
110 END@
120 ABC@ PRINT A
```

■ Description of Sentences

Use upper-case letters to write statements, etc. in a sentence. If the sentence contains a series of statements or labels, as well as variables or constants, they must be separated from each other by a space.

Example: Incorrect: IFA=1THEN GOTO 100

Correct: IF _A=1 _THEN _GOTO _100(_ denotes a space character)

As an exception, space characters can be omitted for special characters described in Section B2.3.

Example: If the sentence is PRINT "Q", the quotation mark (") is a special character.

■ Compound Sentences

In each line, you can describe a series of sentences by separating them from each other with a colon (:). This series of sentences is referred to as a compound sentence.

```
10 A=1 : PRINT A
```

When writing a program with compound sentences, the following restrictions apply.

- (1) A sentence containing any of the following statements cannot be used in a compound sentence.

OPTION BASE, DEFINT, DEFLNG, DEFSNG, DEFDBL, COM, END, FOR, NEXT, DATA, IMAGE, ELSE, ENDIF, WHILE, END WHILE

- (2) A sentence described using any of the following statements must not be followed by a compound sentence.

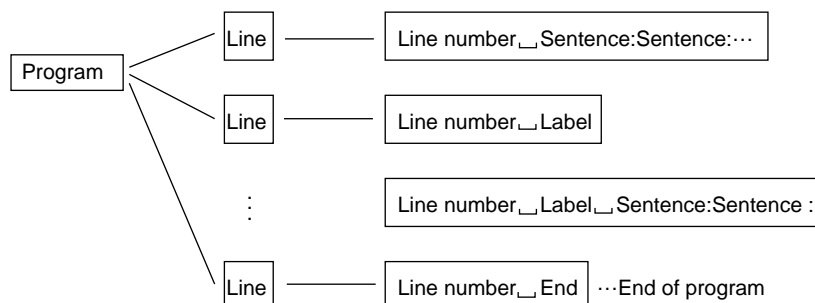
REM, STOP, GOTO, GOSUB, REURN, ON..., WAIT, IF..., CALLLIB

If any sentence that does not adhere to these restrictions is typed, an error will result when the program is input or executed.

Note 1: Despite the above-mentioned restrictions, a comment statement (one beginning with REM or !) can follow any statement, other than DATA and IMAGE, to form a compound sentence.

Note 2: For details on the "IF ..." statement, also see the item "IF ... THEN Statement" in Part C, "Syntax of YM-BASIC/FA," later in this manual.

The flow chart below summarizes the structure of a program. Use an END statement to indicate the end of a program.



FB020202.EPS

B2.3 Character Set

Characters that can be used are the English alphabet, numerals and special characters.

- Alphabet: ABCDEFGHIJKLMNOPQRSTUVWXYZ
 abcdefghijklmnopqrstuvwxyz
- Numerals: 0123456789
- Special characters: _ (space) ! " # \$ % ' () * + , - . / : ; < = > ? @ [¥] { } - |

B2.4 Data Types

There are five types of data that YM-BASIC/FA can deal with: integers, long integers, single-precision real numbers, double-precision real numbers and character strings.

■ Integer

The value of this data type is a whole number from -32768 to 32767 and is represented internally as 2-byte (16-bit) data.

■ Long Integer

The value of this data type is a whole number from -2147483648 to 2147483647 and is represented internally as 4-byte (32-bit) data.

■ Single-precision Real Number

The value of this data type is a real number and is represented internally as 4-byte (32-bit) data. The number of significant digits is approximately 7 in the decimal numbering system. The absolute value of this data type is within an approximate range of 2.7×10^{-20} to 9.2×10^{-18} .

■ Double-precision Real Number

The value of this data type is a real number and is represented internally as 8-byte (64-bit) data. The number of significant digits is approximately 16 in the decimal numbering system. The absolute value of this data type is within an approximate range of 2.7×10^{-20} to 9.2×10^{-18} .

■ Character String

The value of this data type is a character string. The maximum number of characters that can be handled as a single character string is 512 bytes.

B2.5 Constants

A constant is a data item that shows a value in itself. Constants are classified into numeric constants and character-string constants.

■ Numeric Constants

A numeric constant is represented either in one of the decimal-number formats shown in items 1 to 4 below, or in the hexadecimal format shown in item 5.

(1) Representation without a Decimal Point and Exponential Part

The numeric constant is simply a string of decimal numbers (0 to 9) having no decimal point and exponential part. A sign (+ or –) may be appended to the string's head.

Example: 10

–100

(2) Representation with a Decimal Point but without an Exponential Part

The numeric constant has a decimal point either at the start, in the middle, or at the end of a string of decimal numbers. A sign (+ or –) may or may not be appended.

Example: 0.1

–21.0

0.585

(3) Representation with both a Decimal Point and Exponential Part

The numeric constant is represented in the same way as in item 2 but followed by an exponential part. The exponential part is represented by the letter E followed by a decimal number containing no decimal point. A sign (+ or –) may or may not be appended.

Example: 1.2E10

–2.5E–4

0.367E8

(4) Representation without a Decimal Point but with an Exponential Part

The numeric constant is represented in the same way as in item 1 but followed by an exponential part. A sign (+ or –) may or may not be appended.

Example: 1E12

–4E–7

(5) Representation in Hexadecimal Format

The numeric constant is represented as a hexadecimal constant preceded by a dollar sign (\$). The constant is significant up to 8 digits. It is of integer type if consisting of 1 to 4 digits, or of long integer type if consisting of 5 to 8 digits. If 9 or more digits are used to specify the numeric constant, only the first 8 digits have any significance.

Example: \$5A (equivalent to \$005A)

\$1FF3A (equivalent to \$0001FF3A)

Note: If the numeric constant has 4 digits, any value from \$8000 to \$FFFF is interpreted as a negative (–) numeral. If the numeric constant has 8 digits, any value from \$80000000 to \$FFFFFFFF is interpreted as a negative (–) numeral.

\$FFFF=\$FFFFFFFF=–1

\$0FFFF=65535

\$07FFF=\$7FFF=32767

\$8000=–32768

\$08000=32768

■ Character-string Constants

A character-string constant is a character string enclosed with double quotes ("). The character string consists of characters from the alphabet, numerals and special characters. However, two consecutive double quotes are regarded as one character.

Example: "ABC" (equivalent to A&B" C)

 "X-3B2"

 "A&B""C"

B2.6 Variables

A variable is where a value or character string is stored and given a name consisting of upper-case alphanumeric characters. This name is referred to as a variable name.

Variables are classified into numeric variables to which a value can be assigned and character-string variables to which a character string can be assigned.

B2.6.1 Naming a Variable

■ Numeric Variables

A numeric variable is a combination of upper-case letters and numerals, beginning with a letter and comprising up to 8 characters.

Example: Z
 APPLE
 N1917A

■ Character-string Variables

A character-string variable is a combination of upper-case letters and numerals, beginning with a letter, comprising up to 7 characters and ending with a dollar sign (\$).

Example: F\$
 N3200A\$

Note that reserved words, such as statements and intrinsic functions, cannot be used as variable names. (See Appendix 2. later in this manual for more details on reserved words.) Nor can any character string beginning with FN be used as a variable name.

B2.6.2 Declaration of the Type of Variable

Numeric and character-string variables are distinguished from each other by the way they are named. Unless otherwise declared, a numeric variable is of single-precision real number type. The type of each numeric variable can be specified using a type-declaring statement (DEFINT, DEFLNG, DEFSNG or DEFDBL).

Example: DEFINT I All variables whose first letter is I are of integer type.
 DEFDBL L-P All variables whose first letter is one among L to P are of
 double-precision real number type.

Example of variable names after above-mentioned declaration:

ISLOT Integer type
RES1 Real number type
LDAT Double-precision real number type

B2.6.3 Declaration of Variables and Their Defaults

A simple variable can be stated in a program without declaring it in particular. The default of a variable is 0 (zero) for numeric variables and null (\$00) for character-string variables.



CAUTION

Unless expressly cleared using an INIT COM statement, a common variable still retains the previous value. For more details on this procedure, see Chapter B5., "Common Variables," in Part B, "Syntax of YM-BASIC/FA," later in this manual.

B2.6.4 Length of a Character-string Variable

A maximum of 18 characters in terms of the alphanumeric character set (represented by one bit) can be assigned to a character-string variable unconditionally. If you attempt to assign characters in excess of the maximum number, the extra characters are truncated without being assigned to the variable. The length of a character-string variable can be extended up to 512 bytes by declaration using a DIM statement. Since the system actually reserves memory space necessary for that length, for effective use of memory it is advisable that each character-string variable be kept only to the required length.

Example: DIM A\$100 100 bytes

 DIM Y\$8 8 bytes

- Specify the length of a character-string variable as an even number of bytes. Note that the terminator of a character-string variable is defined as null (\$00). Consequently, if the character-string variable contains any null among its characters, the character string is only valid up to the point immediately before the null.

B2.6.5 Array Variables

An array refers to a collection of data items having the same characteristics. The place where such an array is stored is referred to as an array variable. An array variable is declared using a DIM statement and its elements (respective data items) are sequenced using suffixes. For this reason array variables are sometimes referred to as suffixed variables.

Examples:

One-dimensional Array
DUN S(4)

S(0)
S(1)
S(2)
S(3)
S(4)

Number of elements = 5

Two-dimensional Array
DIM T\$(3, 2)

T\$(0, 0)	T\$(0, 1)	T\$(0, 2)
T\$(1, 0)	T\$(1, 1)	T\$(1, 2)
T\$(2, 0)	T\$(2, 1)	T\$(2, 2)
T\$(3, 0)	T\$(3, 1)	T\$(3, 2)

Number of elements = $4 \times 3 = 12$

TB020601.EPS

Since each series of suffixes begins with 0, the number of elements is calculated as the “maximum suffix number + 1” in the case of a one-dimensional array.

However, the first suffix can be changed to 1 using an OPTION BASE statement. In that case, the number of elements equals the maximum suffix number.

- YM-BASIC/FA can deal with arrays of up to two dimensions.
- The maximum suffix number is 32767.
- The size of array data that can be copied using a MOVE statement is smaller than 64 KB.

Table B2.1 Maximum Suffix Numbers That Can Be Declared

	One-dimensional Array	Two-dimensional Array
Numeric array	DIM S (32767)	DIM S2 (32767, 32767)
Character-string array	DIM S\$ (32767)	DIM S2\$ (32767, 32767)

TB020602.EPS

However, these suffix numbers are restricted by such factors as the actual memory size available.

- Calculation of arrays is carried out on an element-by-element basis.

Example:

```
20 FOR I=0 TO 4
30   A(I)=B(I).....Calculated on an element-by-element basis.
40 NEXT I
```

There are statements and functions, however, that deal with an array as a whole.

Example:

```
20 MOVE B(*),A(*)
```

As shown above an array is stated using a variable name (*).

Two-dimensional array variables, on the other hand, are stored in the following order internally.

Example: `DIM S3(5, 5)`

(where the first suffix is defined as 1 in an `OPTION BASE` statement)

S3(1, 1)
S3(1, 2)
S3(1, 3)
S3(1, 4)
S3(1, 5)
S3(2, 1)
⋮
S3(5, 4)
S3(5, 5)

TB020603.EPS

B2.7 Type Conversion

A numeric data item is automatically converted from its current type to another type, as necessary.

■ Constants

A constant is converted to one of the following types, depending on its current notation.

● Integer

- A value from –32768 to 32767 without a decimal point
- Described as a hexadecimal constant

● Long integer

- A value from -2147483648 to 2147483647 without a decimal point
- Described as a hexadecimal constant

● Real number

- A data item without a decimal point, having no more than 6 significant digits and a value that exceeds the range of integer-type data items

● Double-precision real number

- A data item without a decimal point, having no less than 7 significant digits
- A data item with a decimal point
- A data item with an exponential part

■ Type Conversion during Calculation

● Primary Rules

- If the types of two variables do not match during calculation, the system makes a calculation after converting the variables to the type that is the furthest on the right of the applicable types in the figure shown below.

Integer type < Long integer type < Single-precision real number type
< Double-precision real number type

FB020701.EPS

- The final result of calculation is given in the right-side member of the equation and conforms to the type that is the furthest on the right of the applicable types in the figure shown above.
- When substituted into the left-side member, the result of calculation in the right-side member is converted to the data type of the left-side member.
- Division between two integer or long-integer variables is carried out after converting them to single-precision real numbers. The result of calculation is also a single-precision real number.
- Multiplication between two integer variables is carried out as they are. If the result is out of the range from –32768 to 32767, both of them are converted to single-precision real numbers and re-calculated. The result of calculation is also a single-precision real number.

- Multiplication between two long-integer variables is carried out as they are. If the result is out of the range from -2147483648 to 2147483647, both of them are converted to single-precision real numbers and re-calculated. The result of calculation is also a single-precision real number.

● **Exercise (Assume the integer variable as I, single-precision real number variable as R, and double-precision real number variable as D)**

- $R1/R2*D1$
 - (1) Since both R1 and R2 are single-precision real numbers, the result of $R1/R2$ is also a single-precision real number.
 - (2) The result of step 1 is converted to a double-precision real number before operated on D1. The final result is also a double-precision real number.
- $R1/D1*R2$
 - (1) The calculation of $R1/R2$ is made by converting R1 to a double-precision real number, and the final result is also a single-precision real number.
 - (2) R2 is converted to a double-precision real number before the result of step 1 is operated on R2. The final result is also a double-precision real number.
- $R1/R2*R3+D1$
 - (1) The calculation of $R1/R2*R3$ is made using the single-precision real number variables as they are. The final result is also a single-precision real number.
 - (2) The result of step 1 is converted to a double-precision real number before D1 is added to the result. The final result is also a double-precision real number.
- $I1/I2*I3$
 - (1) The calculation of $I1/I2$ is made after converting both variables to single-precision real numbers. The final result is also a single-precision real number.
 - (2) Since the result of step 1 is a single-precision real number, it is multiplied by I3 after I3 is also converted to a single-precision real number. The final result is also a single-precision real number.

Example:

```

100 DEFINT I
110 DEFSNG R
120 DEFDBL D
130 R1=5:R2=7:R3=10
140 D1=4
150 I1=15:I2=24:I3=36
160 PRINT R1/R2*D1, R1/D1*R2, R1/R2*R3+D1, I1/I2*I3
170 END

```

The result of this calculation is as follows.

```

BSC:RUN
2.857142925262451 8.75 11.14285755157471 22.5

```

■ Logical Operations

Variables are first converted to integers before any logical operation is performed.

■ Conversion from Real Numbers to Integers

To convert real numbers to integers, all decimal places are truncated. If the resulting value exceeds the applicable range of integers, an overflow error will result.

Reference: Use an INT function when truncating the decimal places.

■ When a Double-precision Variable Is Substituted into a Single-precision Variable

The double-precision variable is rounded to a number comprising 7 significant digits. In the case of type conversion to a real number having a different level of precision, however, the binary data is rounded off (i.e., rounding down to 0 and rounding up to 1 in a binary number system) to its least significant digit. Consequently, an error may result when the internal value (binary number) of that data is viewed (decimal number) using a PRINT command.

■ Functions

Each function has a predetermined type of variable and any variable is automatically converted to that type.

■ Assignment Expressions

The result of calculation is converted according to the type of variables in the left-side member.

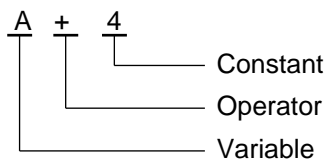
■ Other Cases of Type Conversion

Type conversion is carried out as necessary when data is input or output. The number of significant digits may change for reasons of read-out on a display or internal handling. Modules relevant to this discussion include:

- I/O modules
- Communication modules

B2.8 Expressions and Operations

An expression is a mathematical representation of constants and variables combined using an operator or operators.



FB020801.EPS

Since the result of calculating an expression is either a single numeral or character string, a mathematical representation consisting only of characters, numerals or variables is also regarded as an expression.

An expression by itself does not form any independent statement. Rather, it is quoted in a PRINT statement or an assignment statement (such as a LET statement).

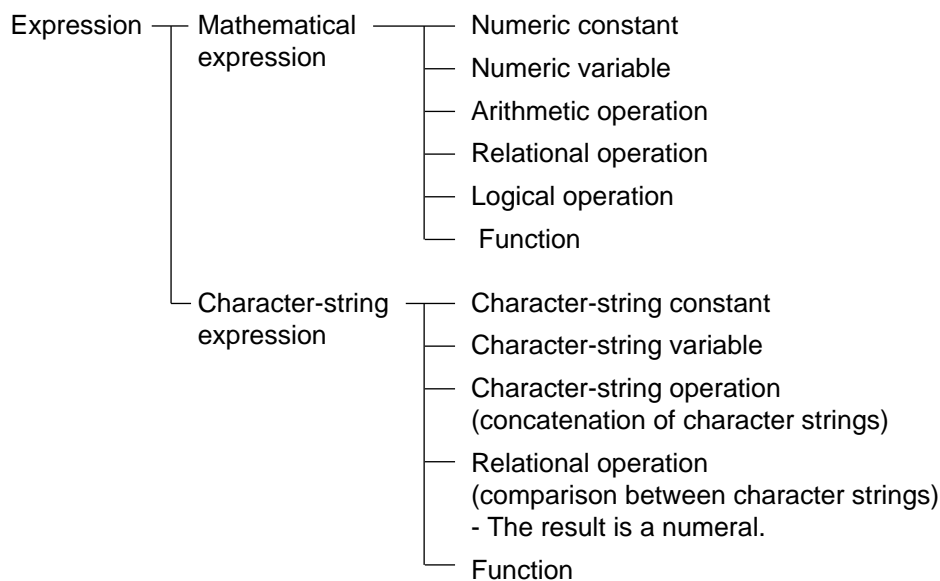
Example: "GRAPE"

1.56

4+3.14

A+B/C-D

SIN(X)



FB020802.EPS

Use parentheses to change the order in which operation is carried out. Any expression within parentheses precedes others during operation. Note that parentheses can be nested into multiple layers.

B2.8.1 Arithmetic Operation

Table B2.2 Arithmetic Symbols and Priority

Arithmetic	Symbol	Priority
Exponentiation	\wedge	High
Multiplication and division	$*, /$	\updownarrow
Addition and subtraction	$+, -$	Low

TB020801.EPS

Example:	Algebraic Representation	BASIC's Representation
	$2x+y$	$2*X+Y$
	$\frac{x-y}{2}$	$(X-Y)/2$
	z^2+y^2	X^2+Y^2
	$y(-x)$	$Y*(-X)$

■ Signs and Monadic Operators

In the case of ordinary operators, operation is carried out on the numerals or variables on both sides of an operator. This type of operation is referred to as dyadic operation. On the other hand, such operation as one using the minus sign as a symbol is referred to as monadic operation.

$X + Y$ Dyadic operation

$-X$ Monadic operation

For example, " $- -4$ " is regarded as " $-(-4)$ " and results in "4".

■ Division between Integers and Remainders

The result of division between integers is a real number. To evaluate the division as an integer, use a function.

$DIV(X, Y)$

$MOD(X, Y)$ Remainder

■ Division by Zero

If division by zero is carried out in calculating an expression, the error "Division by Zero" occurs.

$Y = X/0$

■ Exponentiation of Zero and Zero to the Power of Zero

The exponentiation of zero are defined as shown below.

$0 \wedge (\text{Positive value}) = 0$

$0 \wedge 0 = 1$

$0 \wedge (\text{Negative value}) = \text{Arithmetic overflow}$

B2.8.2 Relational Operation

A relational operator is used to compare two values. The dimensional relationship between the two values is checked to determine whether it:

- matches (= true [i.e., 1]) the significance of the relational operator; or
- mismatches (= false [i.e., 0]) the significance.

Thus, the system returns a value representing either “true” or “false.”

Table B2.3 Relational Operators

Operator	Example	Significance
<	A<B	A is smaller than B.
>	A>B	A is greater than B.
=	A=B	A equals B.
<>	A<>B	A does not equal B.
<=	A<=B	A is equal to or smaller than B.
>=	A>=B	A is equal to or greater than B.

TB020802.EPS

Relational operators are often used in an IF statement.

Example: IF X=0 THEN GOTO 1000

The equal sign (=) is also used in an assignment statement.

Example: A=(B=C)

This statement means B is compared with C and, if B equals C, a value of 1 is substituted into A; otherwise, a value of 0 is substituted into A.

B2.8.3 Logical Operation

Table B2.4 Types of Logical Operation and Their Results

X	Y	X AND Y	X OR Y	X EXOR Y	NOT X
True (1)	True (1)	1	1	0	0
True (1)	False (0)	0	1	1	0
False (0)	True (1)	0	1	1	1
False (0)	False (0)	0	0	0	1

TB020803.EPS

Both the left-side and right-side members are converted to integers (by rounding off the fractions), logical operation is carried out on them by regarding values other than 0 as 1, and a value of 0 or 1 is returned.

Example:

(1) IF X AND Y THEN 800

If both X and Y are 1, execution is moved to the line numbered 800.

(2) IF NOT (A = 0) THEN X = 30

If A is not 0, a value of 30 is substituted into X.

B2.9 Character-string Operation

B2.9.1 Concatenation of Character Strings

Character strings can be concatenated using the + operator.

Example:

```
10 F$="FA"
20 C$="Computer"
30 A$=F$+" "+C$
40 PRINT A$
50 END
BSC:RUN
FA Computer
```

B2.9.2 Comparison between Character Strings

Like comparison between values, character strings can also be compared using one of the following relational operators.

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

In the case of character strings, characters in the left-side member are compared with characters in the right-side member, one at a time, beginning with the leftmost one. If the character strings in both members are completely the same, they are equal to each other. If any two characters under comparison differ, the character whose internal code is larger is judged to be greater.

Example:

```
"APPLE"="APPLE"
"AA"<"AB"
"pen">"PEN"
```

If a character string on either side is shorter than the other one and comparison finishes halfway, then the shorter string is judged to be smaller. Note that a white space also has significance in this comparison.

Example:

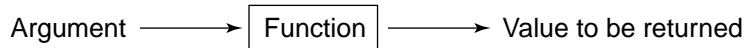
```
"WTD"<"WTDATA"
"PMK"<"PMK "
```

For more details on the internal codes, see Appendix 1 later in this manual.

Order of JIS 7-bit codes (alphabet, numerals and symbols)

B2.10 Functions

A function carries out a predetermined operation on the given parameter (or argument) and returns the result as a value.



FB021001.EPS

The data type function conforms to the type of data that the function returns.

Example:

SIN(X) Numeric type
 CHR\$(\$41) Character type

A function can be quoted by describing it in an expression after appending an appropriate actual argument to the function's end. As an actual argument, an expression can also be used. In that case however, the number of units and the type must match between dummy arguments and actual arguments. Functions are classified into intrinsic functions that are previously defined by the YM-BASIC/FA and functions defined by the user. Some functions do not have any argument.

Example:

DATE\$

B2.10.1 Intrinsic Functions

Intrinsic functions can be used by the user without having to define them. They include arithmetic functions, functions that deal with bits, and functions that deal with characters. For more details on the intrinsic functions, see Section C2.3, "Functions," in Part C, "Syntax of YM-BASIC/FA," later in this manual.

Example:

DIV(X, Y)
 HEX\$(A)
 LEN(TEXT\$)

B2.10.2 User-defined Functions

User-defined functions are defined by the user before they can be put in use. The name of a user-defined function must always begin with FN. See the DEF FN statement in Part C, "Syntax of YM-BASIC/FA," later in this manual for more details.

Example:


```

10 DEF FN RAD(X)=X*PI/180
20 PRINT FN RAD(180)
30 END
  
```

B2.11 Priority of Operations

Operations are carried out in the following order of priority. If operators are at the same priority level, they are processed in the order from left to right in the following table.

Table B2.5 Priority of Operations

Type of Operation	Symbol (Operator)	Priority
Operation on expression within parentheses	Parentheses ()	Highest priority
Operation on function	Function	
Monadic operation	+, −, NOT	
Exponentiation	^	
Multiplication and division	*, /	
Addition and subtraction	+, −	
Relational operation	=, <, >, >=, <=, <>	
Logical operation	AND	
Logical operation	OR, EXOR	Lowest priority

TB021101.EPS

An equal-sign operator (=) in any assignment statement is even lower than the lowest priority in the table above.

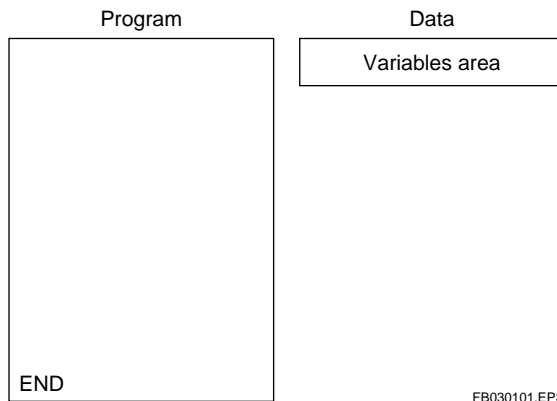
In the YM-BASIC/FA, monadic operation has priority over exponentiation in processing so that the speed of operation is increased. When using the exponentiation operator, enclose it with parentheses. This will make the expression identical to an algebraic representation.

−X² −(X²)

B3. Subprograms

B3.1 Structure of a Program

A program basically consists of two or more lines classified by the line number, where the last line must end up with an END statement. This program is referred to as a main program. Data for variables used in a main program are stored in the variables area.



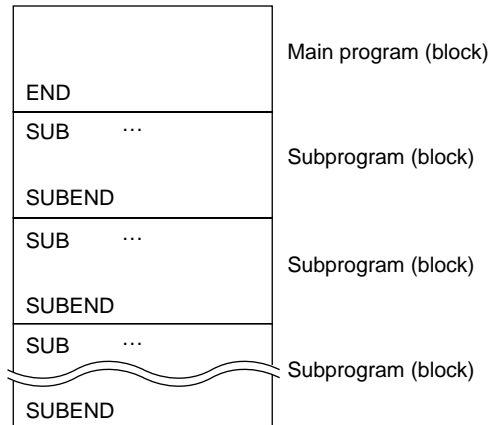
FB030101.EPS

Note that the variables area discussed here is formed within the user area. Variables in this area are also referred to as local variables, as they are used with this particular program only. A regular BASIC program has this structure.

B3.2 Subprograms

Subprograms are programs separately created on a function-by-function basis. Their variables and line numbers can be managed separately as well. They resemble a subroutine represented by “GOSUB ... RETURN” in terms of the flow of a program.

Using a main program structured as explained in Section B3.1 of Part B, “Description of YM-BASIC/FA,” in combination with multiple subprograms, you can create an even larger program. Such a program is configured as shown in the following figure. Note that variable areas are reserved separately in their respective program blocks.



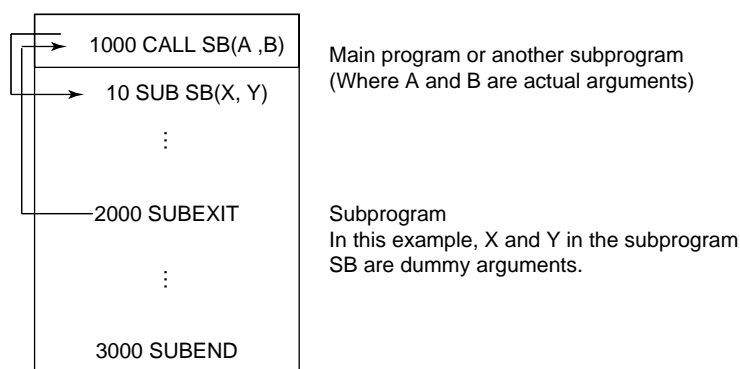
FB030201.EPS

Any single BASIC program consists of one main program and several subprograms, though subprograms are not always required. Each main program and subprograms is also referred to as a program block. There is no particular rule that a statement be written in the beginning of a main program; however, an END statement must always be placed at the end of the program. In addition, a subprogram must begin with a SUB statement and end up with a SUBEND statement.

B3.3 Call of Subprograms

Any subprogram can be called from a main program or another subprogram. To call a subprogram, use a CALL statement.

Example:



FB030301.EPS

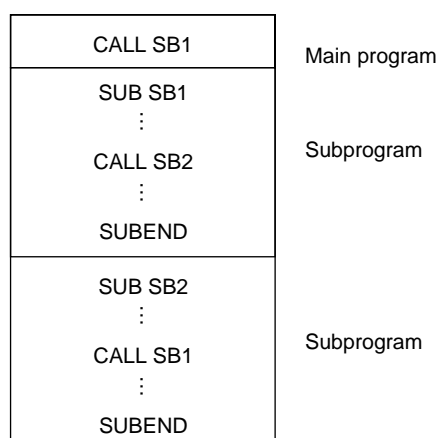
Declare the name and dummy arguments using a SUB statement in the beginning of a subprogram. The subprogram name must be a string of no more than 8 alphanumeric characters and begin with a letter. Define the type and quantity of dummy arguments (parameters), to allow other blocks to be able to call the subprogram. If the definition of a dummy argument is unnecessary it may be skipped.

Use a CALL statement, as shown in the figure above, in another program block from which you call the subprogram in question. In that case, the type and quantity of the actual arguments must match those of the dummy arguments defined using a SUB statement. Upon the end of execution, the program that was called returns to the program block from which it was called. At this point, a SUBEXIT statement is used. If the SUBEXIT statement is omitted, a SUBEND statement serves the same purpose.

Note that END and STOP statements are only valid when used within a main program. If used within any subprogram, the statements will result in an error when the program attempts to execute them.

When the execution of the SUBEXIT statement is complete, program execution moves to the statement that immediately follows the CALL statement of the program that called the subprogram in question. Note that a subprogram cannot be used in such a recursive way as calling itself. Nor can any subprogram call itself indirectly through another subprogram.

Example: Incorrect use of subprograms

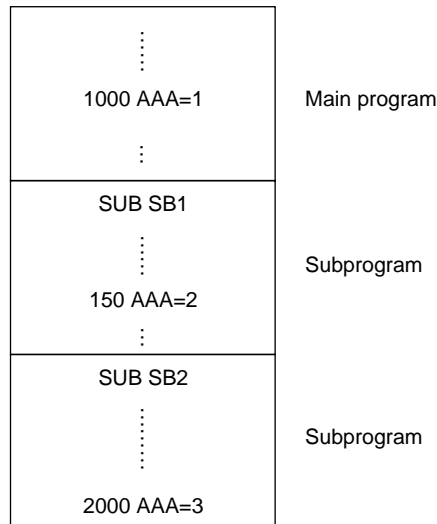


FB030302.EPS

B3.4 Independency of Programs

■ Variables and Line Numbers (Labels)

One of the advantages of using subprograms is that you can separately use variable names and line numbers within a particular subprogram only, irrespective of other programs. For example, you can use subprograms as shown in the following figure.



FB030401.EPS

In the figure above, the main program and two subprograms deal with variables having the same name AAA. In terms of the characteristics of a program, however, these variables are processed as totally different ones. Likewise, line numbers and labels can also be used in an independent manner among the subprograms involved.

■ Statements Relevant to Various Declarations

Statements that make various declarations, such as interrupt definitions and type declarations, are available with the YM-BASIC/FA. This item explains how widely these statements apply to program blocks.

(a) Statements That Define Interrupts

An interrupt-defining statement beginning with "ON ..." is only valid within the program block where it is defined or within a subprogram that is called from the program block. The ON ... declarative statement is automatically turned into the OFF ... statement when exiting the subprogram with SUBEXIT, nullifying the interrupt definition. If you declare the same interrupt factor in two places within the same program block, only the later declaration is valid. If you declare the same interrupt factor in different program blocks, the later declaration results in an error. You must therefore be careful when making declarations within subprograms. The ON ERROR statement is exceptional, however, as it is only valid within the program block where it is declared.

Statements that comply with the statements discussed here are as follows.

ON TIME

ON TIMER, ON SEQVET,

ON EOT, ON INT, ON TIMEOUT

(b) Statements That Cancel Interrupts

An interrupt-canceling statement that begins with OFF ... should be used within the program block where it was defined. Be careful since any OFF ... statement used within program blocks other than the one where it was defined becomes invalid.

(c) Declarative Statements

The following declarative statements relevant to variables are only valid within their respective own program blocks.

DEFINT/DEFLNG/DEFSNG/DEFDBL, OPTION BASE, DIM, ALLOCATE, DEF FN, COM, RECOM

(d) I/O-related Declarative Statements

I/O-related declarative statements are common to (and valid for) all program blocks.

I/Os are valid within a program block where an ASSIGN statement that defines module configuration is made, or within a subprogram that is called from the program block. As discussed in the preceding item, any I/O-related declaration within a subprogram is released from the ASSIGN state when you exit that subprogram with SUBEXIT. The ASSIGN declarative statement remains valid if it is executed once. It is therefore advisable that you make an ASSIGN declaration within a main program.

B3.5 Arguments Transferable to Subprograms

Arguments (parameters) can be defined when they are called between a main program and a subprogram. The same quantity of arguments must be stated both on the caller side (CALL statement) and recipient side (SUB statement).

■ Statements Where No Arguments Can Be Stated

No arguments can be stated in statements written in the format ON xxxx CALL ... Use a common variable to pass data to a subprogram.

■ Items Statable As Arguments

The following six items can be stated as arguments.

(a) Variable

This argument can pass a character-string variable and a numeric variable. A whole array can also be stated.

(b) [Variable]

This argument is for reference only within a subprogram and no data can be written into it. The rest of the characteristics are the same as item (a).

(c) Arithmetic Expression

This argument is for reference only within a subprogram and no data can be written into it.

(d) Character-string Expression

This argument is the same as item (c).

(e) Constant

This argument allows either a numeric constant or a character-string constant to be stated. It is for reference only within a subprogram.

(f) Common Variable

This argument allows a common variable to be stated. Common variables can also be stated within a subprogram.

The following is an example of how these arguments are stated.

CALL S1 (A (*), [B\$], C+100, "DD", F\$), where F\$ is a common variable.

SUB S1 (P (*), Q\$, R, S\$, T\$)

- In this example, arguments that are for reference only are Q\$, R and S\$.
- P (*) denotes a whole array.

B3.6 Subprograms and Subroutines

A subprogram should be interpreted as a subroutine and executed as a relatively large size of program. A subroutine (GOSUB) should be interpreted as a relatively small size of program.

A subprogram, when called, is placed into pre-run so that a necessary variable area is reserved, for example. If at this point the necessary variable area is not reserved, the error message "Insufficient Area" is given.

Since a subprogram reserves and cancels a variable area each time it is executed, the efficiency of memory is increased. Its speed of execution is lower, however, compared with a GOSUB subroutine. It is therefore advisable that a GOSUB subroutine be used in parts of a program that need to be processed at higher speeds. Use subprograms for a program that will take several hundred milliseconds to complete execution or has more than several hundred steps. Use GOSUB subroutines for a program that will take only several tens of milliseconds to complete execution or has only several tens of steps.

B3.7 Variables and Labels

Restrictions apply to the total sum of variables and labels allowed within a single program, as shown below.

Total sum of variables and labels ≤ 1637

Exceeding this limit results in the error code (ERRC) 80, "Insufficient Area."

TIP

A program consisting of a main block only has only one program block. If a program has a subprogram or subprograms, one subprogram equals one program block.

B4. Real-time Statements

The YM-BASIC/FA is real-time BASIC. The programming language has an interrupt handling function that, when a specific factor (event) occurs, stops the current execution and runs a process appropriate for that factor, irrespective of the flow of a program.

B4.1 Execution Modes

The mode of regular system operation is referred to as the REAL mode. In contrast, the mode in which programs are created, or debugged while being executed, is referred to as the DEBUG (command input) mode.

To start a program in the DEBUG mode, use a RUN command. In the REAL mode, specify the program as being resident in the FA-M3 controller so that it is executed when the power is turned on.

B4.2 Wait for Events (WAIT)

Use a WAIT statement when bringing the program to a stop (wait state) or postponing the execution of the program.

■ WAIT Statement

A WAIT statement is used to wait for an external interrupt to occur. It is when a certain event occurs that the wait state due to a WAIT statement is cancelled. The wait state can be cancelled by pressing the ESC key on the keyboard when the command-input mode is active. Consequently, the program comes to a pause. When returning from a subroutine branching off during a wait state or from a subprogram, the CPU moves to the line number that immediately follows the WAIT statement. If you use a GOTO statement instead of a WAIT statement and state as

```
160 GOTO 160,
```

then the GOTO statement is executed over and over again until an interrupt occurs. This will increase the CPU load. For this reason, use a WAIT statement to wait for an interrupt.

■ WAIT statement for Delay Time

A WAIT statement for delay time causes the CPU to wait a specified time and then goes to the next line number. A branch immediately takes place if any event occurs during a wait state. When returning from a subroutine to which a branch was caused or from a subprogram, the CPU waits until the remaining delay time expires and then goes to the next line number.

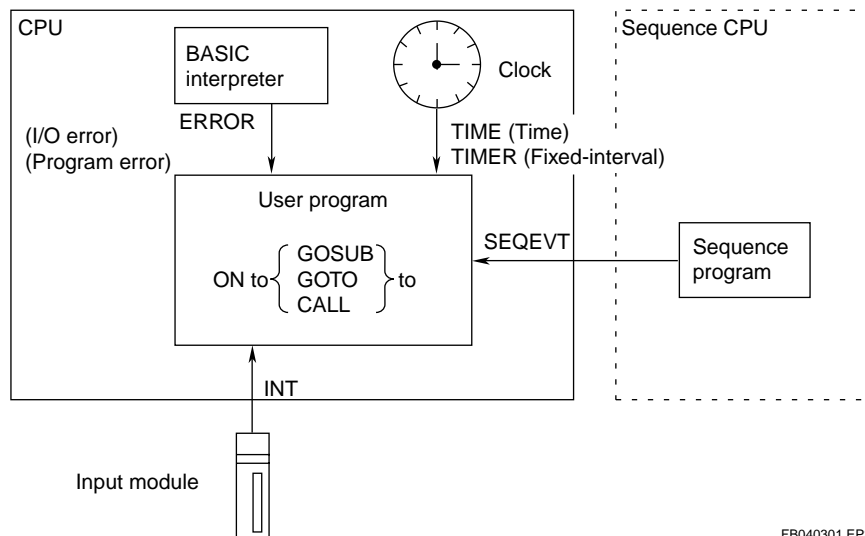
B4.3 Interrupt

When the flow of a program needs to be changed by means of an external factor, use an interrupt. This feature enables real-time programming that carries out processes appropriate for irregular events.

(1)Types of Interrupt

The types of interrupt are shown in Figure B4.1. The factors of interrupt include time, time length, event notification from a ladder sequence program, and event notification* from an I/O module, interpreter (in case of an error) or other BASIC programs.

Depending on the type, some interrupts cause a branch an indefinite number of times if stated once while others cause a branch only once.



FB040301.EPS

Figure B4.1 Types of Interrupt

Interrupts from a ladder sequence program are explained in Chapter B6 of Part B, "Description of YM-BASIC/FA," later in this manual.

(2) Interrupt Handling for Branches to Subroutines (ON Interrupt factor GOSUB)

The following is a statement for declaring acceptance of interrupts.

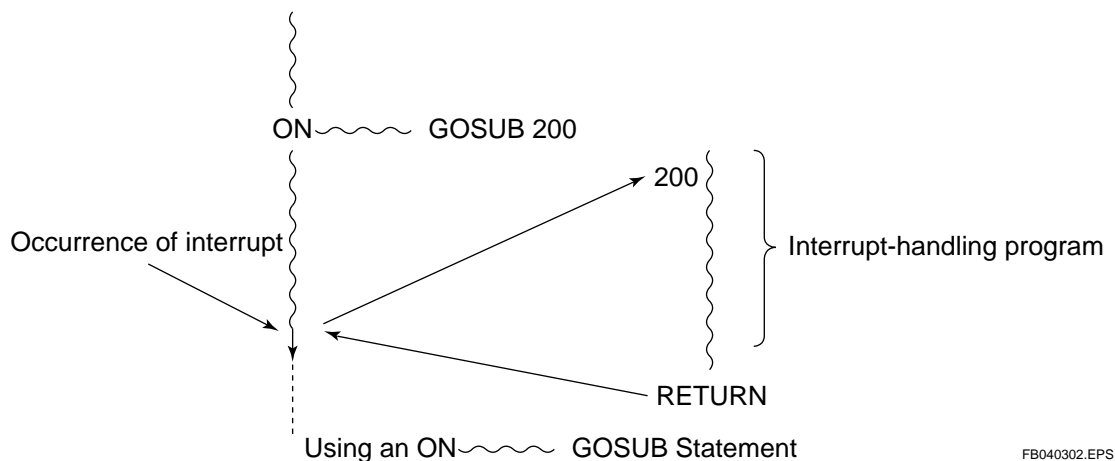
```
ON  Interrupt factor  GOSUB { Line number }
                             { Label }
```

In this statement, you define the interrupt factor and the line number to which a branch is caused upon interrupt. The subsequent lines of the program are then executed in sequence. If an interrupt occurs after the execution of a statement for declaring acceptance of interrupts, a subroutine jump takes place to the declared line number.

Upon the jump to the interrupt-handling subroutine, the system automatically places the CPU in an interrupt-disabled (masked) state until a RETURN statement is executed. In this state, no jump occurs, though interrupts are still accepted.

If the RETURN statement is executed in the subroutine, the system automatically places the CPU in an interrupt-enabled (unmasked) state. If there is an interrupt while the interrupt-handling subroutine is being executed (in a masked state), a jump takes place at that time.

An indefinite wait using a WAIT statement while interrupt handling is in process in a masked state will result in a syntax error.



FB040302.EPS

(3) Interrupt Handling for Changing the Program Flow (ON Interrupt factor GOTO)

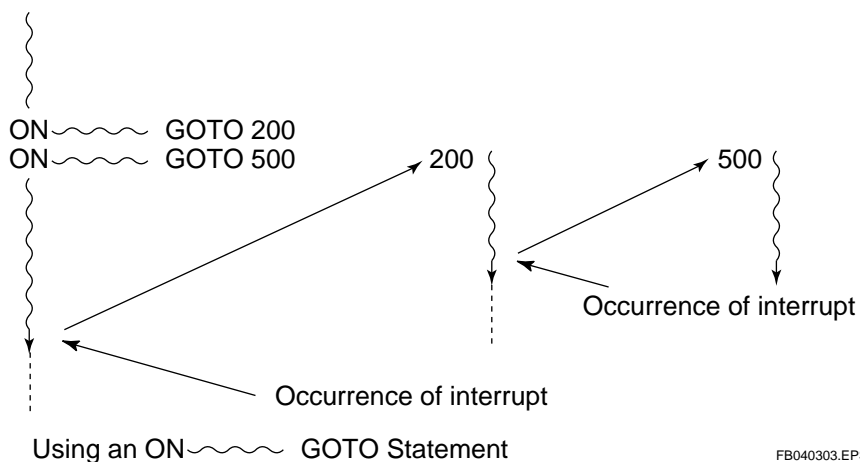
Similar to the statement in the preceding item, the following is a statement for declaring acceptance of interrupts.

```
ON  Interrupt factor  GOTO  { Line number }
                             { Label      }
```

When an interrupt is accepted, a jump is caused to the line number specified in this statement. The CPU however, cannot return to the line number where it was before the occurrence of the interrupt. It is therefore advisable not to use the statement except in exceptional cases. Create a program that executes at least one statement after a jump.

Since the CPU is still in an interrupt-unmasked state after a jump to the specified line number, it takes another jump if another interrupt occurs. In that case, the user can control the interrupt-masked/unmasked states using DISABLE/ENABLE statements.

If any ON ... GOTO branch is defined in a subroutine, the RETURN information is cleared when the ON ... GOTO branch is executed. Consequently, executing a RETURN statement after the jump will result in the error "No Destination of Return."



FB040303.EPS

(4) Interrupt Handling for Subprogram Jumps (ON Interrupt Factor CALL)

The following is a statement for declaring acceptance of interrupts.

```
ON  Interrupt factor  CALL  Subprogram name
```

In this statement, you define the interrupt factor and the name of a subprogram to which a jump is caused upon interrupt. Not arguments can be passed at this point. The rest of the characteristics is the same as those of the subroutine jump discussed in item (2).

(5)Interrupt Prohibition (OFF Interrupt Factor)

The following is a statement for declaring cancellation of accepting interrupts based on an ON ... statement.

OFF Interrupt factor

All interrupts relevant to this factor which follow this statement are ignored. If the ON ... statement is executed once again, interrupts are also accepted once again. Thus, you can control the acceptance/prohibition of interrupts at your option according to the sequence of processes in your program.

All interrupts are prohibited in the beginning of a program, as a matter of course. An interpreter error is regarded as a system interrupt in the absence of an ON ERROR statement, causing the program to stop. Be careful since an OFF ... statement is only valid within a program block where interrupts have been defined using an ON ... statement.

(6)Interrupt Statements

● Interrupt by Time

ON__TIME__ #Timer number, Time [, Time interval] $\left\{ \begin{array}{l} \text{GOTO} \\ \text{GOSUB} \\ \text{CALL} \end{array} \right\} \text{---} \left\{ \begin{array}{l} \text{Line number} \\ \text{Label} \\ \text{Subprogram name} \end{array} \right\}$

1 to 8

OFF__TIME__ # Timer number

FB040304.EPS

With these statements, an interrupt occurs at the specified time. If only the time is specified, a branch takes place only once. When a time length is specified, an interrupt occurs periodically after the specified time.

● Interrupt by Timer

ON__TIMER__ # Timer number, Time interval__ $\left\{ \begin{array}{l} \text{GOTO} \\ \text{GOSUB} \\ \text{CALL} \end{array} \right\} \text{---} \left\{ \begin{array}{l} \text{Line number} \\ \text{Label} \\ \text{Subprogram name} \end{array} \right\}$

1 to 8

OFF__TIMER__ # Timer number (The timer number may be the same as the one in the ON __TIME statement.)

FB040305.EPS

With these statements, an interrupt occurs at an interval specified in the Time Length field. The timer numbers in the ON TIME and ON TIMER statements can be set separately; therefore, the same number may be used for both.

- **Interrupt from an Input Module**

ON__INT__Slot number, Device number [, Terminal number] _ {
 {GOTO } _ {Line number}
 {GOSUB } _ {Label}
 {CALL } _ {Subprogram name}

OFF_ INT_ Slot number, Device number [, Terminal number]

FB040306.EPS

With these statements, it is possible to handle interrupt input from any external device.

Note: The following input (or I/O) modules have no interrupt feature.
F3XD64-□□, F3WD64-□□

● Interrupt from the Interpreter

ON_ERROR_	{	GOTO	}	— {	Line number	}
		GOSUB			Label	
		CALL			Subprogram name	

OFF_ERROR

FB040307.EPS

These statements are instructions that cause the interpreter to generate an interrupt without taking any action against an error resulting from faulty programs or data. Thus, error handling is carried out by the user program. The ON ERROR statement is regarded as an exceptional interrupt instruction, a branch takes place upon occurrence of an interrupt even when a process based on another interrupt factor is in progress.

(a) Branching by ON ERROR GOTO Statement

Example:

```

50    ON ERROR GOTO A@
    :
100   OUTPUT XXX... ← Occurrence of error
    :
200   A@ DP ERRL, ERRC, HEX$(ERRCE)
    :
250   OUTPUT XXX... ← Occurrence of error

```

Any error occurring during the execution of an ON ERROR GOTO branch will result in an indefinite loop. After the ON ERROR GOTO branch, be sure to execute an OFF ERROR statement.

(b) Branching by ON ERROR GOSUB Statement

If another error occurs during the execution of a GOSUB branch, the program stops because it is interrupted by the system. Therefore during the execution of an ON ERROR branch, it is safer to avoid using such a statement as an error may occur therein in an online state.

If any error occurs in a statement of a compound sentence, a branch takes place immediately. Then, according to a RETURN statement, the CPU returns to the line that follows the one from which it branched. A RETURN RETRY statement, when executed, causes the CPU to return to the first field of the line number in question. For this reason it is advisable that the statement be made using a simple sentence for operation involving I/O-related actions.

Example:

```
100 A=1:OUTPUT 8,1;A$:C=1
      Error
110 D=1
```

If an error occurs in an OUTPUT statement, an error branch is caused without executing C = 1. With a RETURN statement, the CPU returns to line number 110. With a RETURN RETRY statement, the CPU resumes execution from the first field A=1 of line number 100.

● Interrupt from Sequence Program

ON__SEQEVT__ Signal name [, Variable name]__ { GOTO } { Line number }
 { GOSUB } { Label }
 { CALL } { Subprogram name }

OFF__SEQEVT__ Signal name

FR040308 FPS

FB040308.EPS

With these statements, an interrupt by a SIGNAL command in a sequence program is accepted. This makes it possible to synchronize the sequence program with a BASIC program. For example, these statements can be used to determine the time when the data of sequence devices are read from the BASIC program using an ENTER statement. The data can also be transferred in the form of a signal. Data transfer can be carried out at a rate of one cycle per variable (integer-type).

● Interrupt at the End of Data Transfer

ON_EOT Slot number, Device number { GOTO } — { Line number }
 { GOSUB } — { Label }
 { CALL } — { Subprogram name }

OFF_EOT Slot number, Device number

FR040309.FPS

FB040309.EPS

With these statements, an interrupt occurs at the end of data transfer by a TRANSFER statement. These statements allow an action to be taken following the end of data transfer to a communication module by a TRANSFER statement.

- **Interrupt Generated when an I/O Action Does Not Finish within Specified Time Length**

ON__TIMEOUT__Slot number, Device number__
 OFF__TIMEOUT__Slot number, Device number

{ GOTO } — { Line number }
 { GOSUB } — { Label }
 { CALL } — { Subprogram name }

FR040310.FPS

FB040310.FPS

These statements allow an action to be taken when an I/O action does not finish within the length of time specified by a SET TIMEOUT statement.

(7) Temporary Prohibition of Interrupts and Its Cancellation

The temporary prohibition and its cancellation of all interrupts declared by an ON ... statement can be controlled using DISABLE/ENABLE statements. Although in an ON ... declaration, DISABLE/ENABLE statements in a branch to a subroutine or subprogram are valid, the statements do not make sense when the line is at the high level. Rather, the state in question arises when the line returns to the low level.

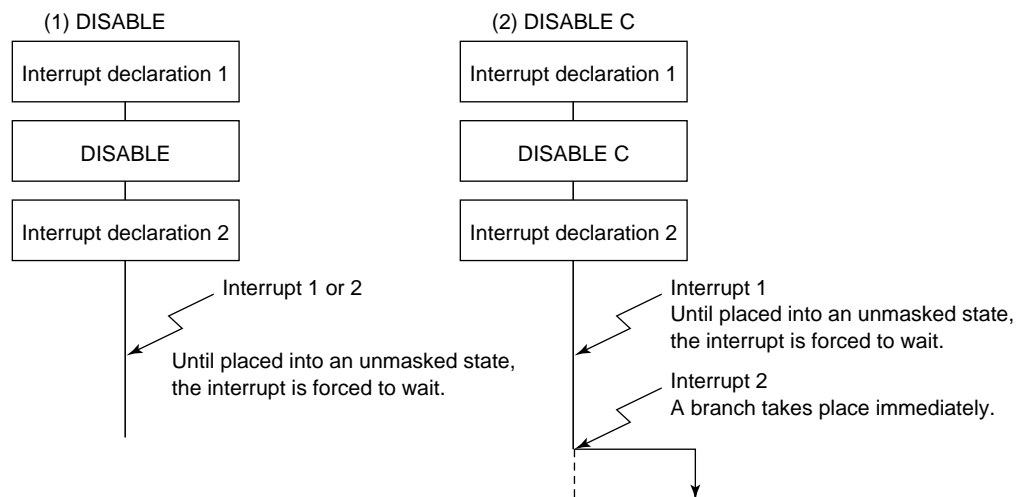
● DISABLE [C] Statement

This statement temporarily prohibits all interrupts, except those set by an ON ERROR statement. A DISABLE statement places all declared interrupts into a disabled (masked) state. If you declare an interrupt using an ON ... statement after the execution of a DISABLE statement, the declaration is registered with the system with the interrupts kept in the masked state. A DISABLE C statement, on the other hand, places interrupts declared before the DISABLE C statement is executed into a masked state. Any interrupt declaration implemented after the execution of the DISABLE C statement is placed into an enabled (unmasked) state.

Interrupts are accepted even if they are in a masked state, and a branch takes place when they are placed into an unmasked state.

Use a DISABLE C statement when you want to temporarily prohibit interrupts from occurring.

Example:



FB040311.EPS

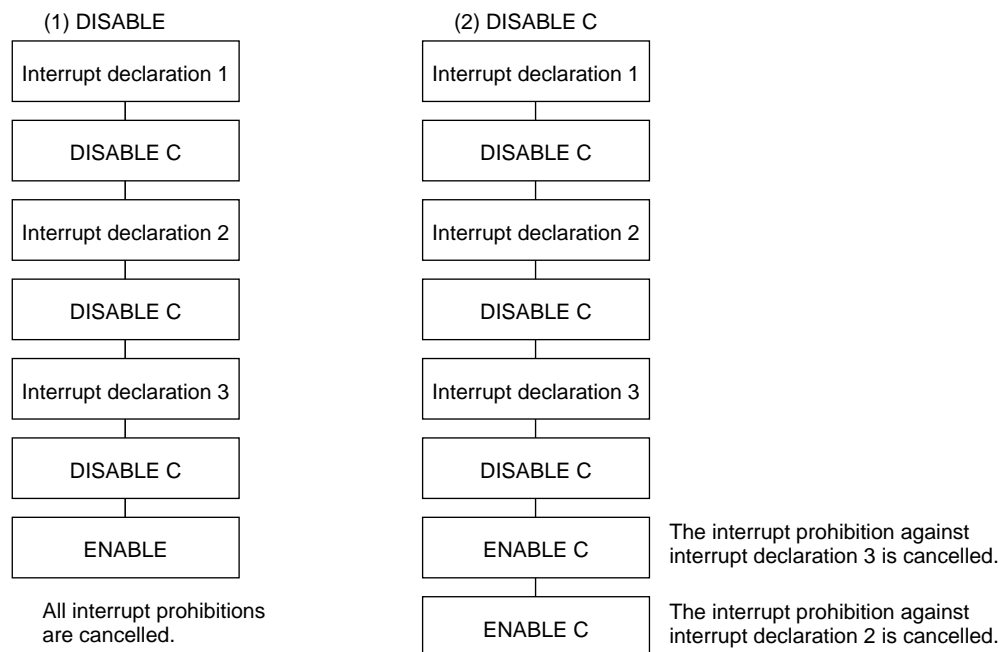
Interrupts are placed into an unmasked state automatically if you execute a WAIT statement in a masked state set by a DISABLE statement. As an exception, the status of control on the masked state set by a DISABLE C statement does not change, however.

● ENABLE [C] Statement

This statement places interrupts prohibited by a DISABLE statement into an unmasked state (interruptible again). If an interrupt occurs in a masked state, a branch takes place when this statement is executed.

An ENABLE statement cancels all the masked, prohibitive states of interrupts set by DISABLE [C] statements executed earlier. An ENABLE C statement places an interrupt prohibited by the immediately preceding DISABLE C statement into an unmasked state (interruptible).

Example:

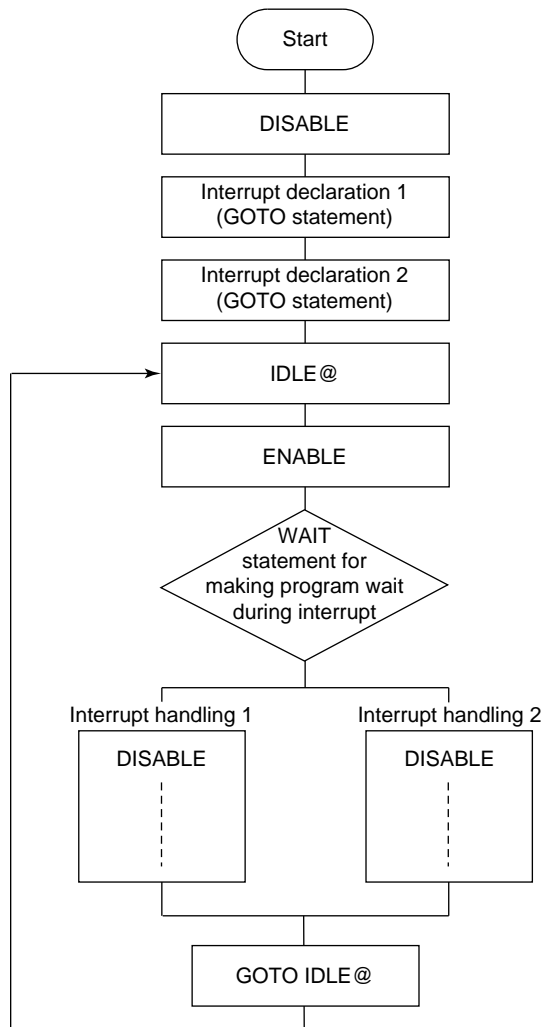


FB040312.EPS

● Using DISABLE/ENABLE Statements

As a rule, DISABLE/ENABLE statements are used as shown below.

Example (1)

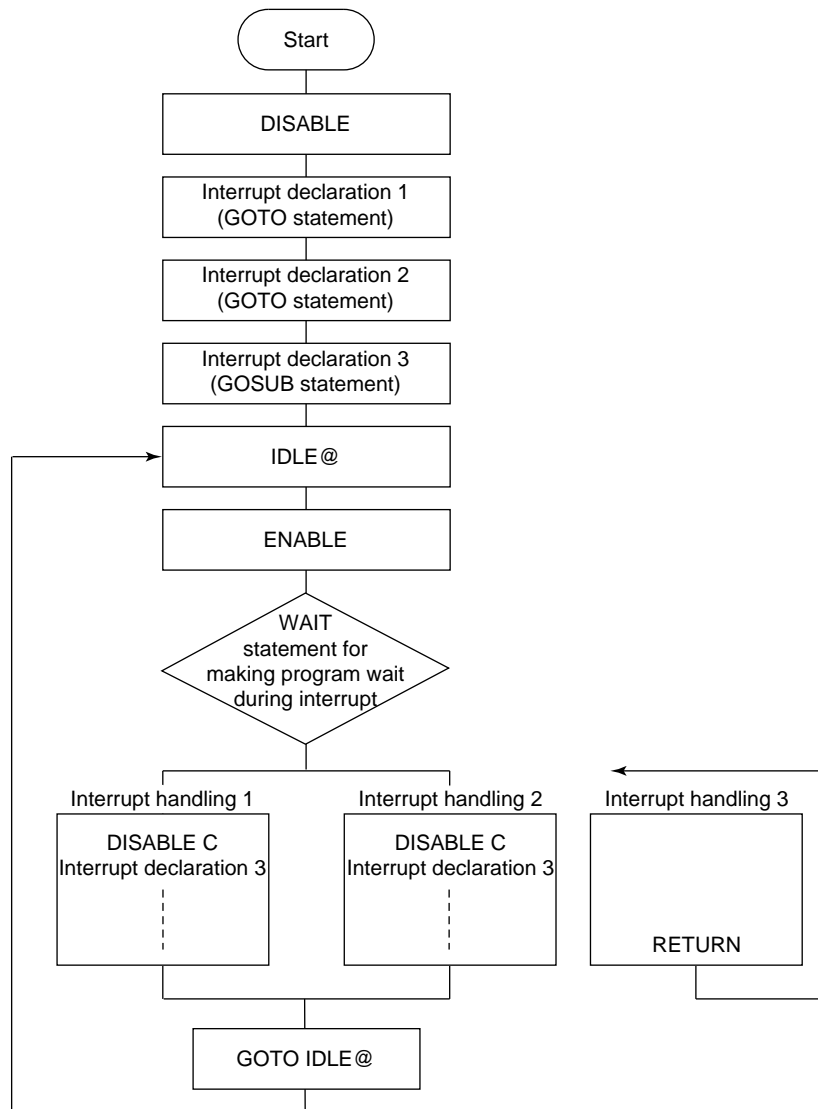


FB040313.EPS

This way of using the statements enables you to handle interrupts without having the program flow disturbed by other interrupts during interrupt handling based on a GOSUB statement.

Such programming as shown below is possible by combining DISABLE and ENABLE statements with DISABLE C and ENABLE C statements.

Example (2)



FB040314.EPS

Interrupt handling 3 is always feasible even when interrupt handling 1 and 2 is in process. This way of writing a program is convenient when immediate processing is always needed in communication sessions, for example.

In the case of interrupt handling 1 and 2 in Examples (1) and (2) discussed earlier, a DISABLE statement must be made in the line that is first executed after the occurrence of an interrupt branch. The YM-BASIC/FA is designed so that interrupt factors are checked for at the end of each line number. If there is any unprocessed interrupt factor, an interrupt branch takes place. For this reason, make a DISABLE statement in the line to which the interrupt branch is caused, as shown in Example (3-1) below. In either case in Example (3-2), if there are any unprocessed interrupt factors upon completion of line 190, a branch is caused to the interrupt handling corresponding to that factor.

Example (3-1) Correct Use**(Branch to Line Number)**

```
50 ON SEQEVN "EVENT1" GOTO 190
:
180 ! Interrupt handling 1
190 DISABLE
200 :
```

(Branch to Label)

```
50 ON SEQEVN "EVENT2" GOTO EVT2@
:
180 ! Interrupt handling 2
190 EVT2@ DISABLE
200 :
```

Example (3-2) Incorrect Use**(Branch to Line Number)**

```
50 ON SEQEVN "EVENT1" GOTO 190
:
190 ! Interrupt handling 1
200 DISABLE
```

(Branch to Label)

```
50 ON SEQEVN "EVENT2" GOTO EVT2@
:
180 ! Interrupt handling 2
190 EVT2@
200 DISABLE
```

(8)Timing for Interrupts

The YM-BASIC/FA is designed so that an interrupt branch takes place after the completion of the line number currently under execution. This is still true even if the interrupt is stated using a compound sentence; thus, the branch takes place after all statements included within the line number have been executed.

- Branch due to occurrence of error

A branch immediately takes place upon occurrence of an error. In a line where statements are made with a compound sentence, those subsequent to the statement where the error occurred are excluded from the execution. Statements relevant to error handling include ON ERROR and SET STATUS.

- WAIT statement

In the case of a WAIT statement, interrupts are accepted as an exception even while the statement is being executed.

(9)Priority of Interrupts

This item explains how interrupts are accepted in a case where there is a number of interrupt factors.

- Order of declaration

Acceptance of interrupts are declared using an ON ... statement. There are no particular restrictions on the order and number of interrupts. An acceptance table is created within the user area for each ON ... statement. If you declare exactly the same interrupt factor twice, however, the later declaration takes precedence.

- Order of acceptance

Interrupts are accepted in the order of their occurrence. For each interrupt, a branch takes place to an interrupt-handling subroutine created by the user. The order of accepting interrupts that have occurred exactly at the same time depends on the hardware and system. Interrupts are never ignored since the system still accepts them even when any user interrupt is being processed.

- Two or more interrupts caused by the same interrupt factor

The CPU remembers only one interrupt for each interrupt factor. The second and subsequent interrupts are ignored. The CPU clears its memory when a branch is caused to an interrupt-handling subroutine, however. Consequently, interrupts due to the same interrupt factor as above are accepted when the interrupt-handling subroutine is in progress. A branch takes place to the same interrupt-handling subroutine once again when the CPU returns from that subroutine.

(10) Interrupt-handling Programs

Interrupt-handling subroutines, subprograms and DISABLE statements must be processed as quickly as possible since they are executed with other interrupts placed into a masked state. An indefinite wait using a WAIT statement while interrupt handling is in process in a masked state will result in a syntax error.

(11) Quantity of Interrupts That Are Accepted

If too many interrupts come in at the same time, the system may not accept some of them even if they are of different types. If more than 40 interrupts come in while any single line is being executed, all subsequent interrupts are ignored unconditionally.



CAUTION

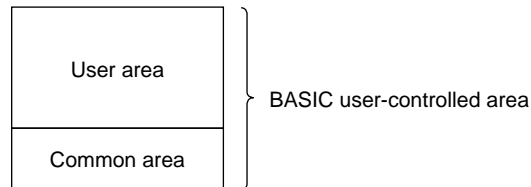
If more than 40 interrupts come in while an interrupt-handling subroutine based on an ON Interrupt factor GOSUB statement is being executed, all subsequent interrupts are ignored unconditionally.

B5. Common Variables

B5.1 Common Area

The BASIC user-controlled area consists of a user area (program area) and a common area. The common area is not initialized even if the BASIC CPU module is turned on or off. To initialize the common area, use an INIT COM statement. The size of the common area can be determined in advance by means of software configuration.

Areas within BASIC CPU module



FB050101.EPS

B5.2 Basics of How to Use Common Variables

B5.2.1 Functions of COM Statement

A COM statement is used to define variable names in the common area. Like a DIM statement, you can also declare arrays and specify the length of a character string. A DIM statement places definitions in the program area, while a COM statement sets definitions in the common area. Variables declared by a COM statement are referred to as common variables.

State as shown below to secure a common area within the local user area.

COM _Variable name [, Variable name,]

where the description within the brackets [] is for a case when there is more than one variable.

Variables declared by COM statements are allocated to the common area in the order in which they are stated in a program. Declared variables are not referenced correctly unless their data type, quantity (number of elements if in an array), and order are consistent between programs. In addition, if the variables are of character-string type, the length of any particular element must be the same between programs.

Matching as to whether the variable is an array variable or a simple variable is not required as long as the above-mentioned conditions are satisfied. A variable name need not necessarily be the same between programs.



CAUTION

When declaring any character-string variable in the common area by a COM or COM # statement, be sure to specify it using an even number of bytes. If you specify the variable using an odd number of bytes, an error may occur in a library or the program performance (execution speed) may be degraded.

B5.2.2 Clearing the Common Area

The common area exists separately from the program area and is not cleared even by a NEW command. To initialize the common area, use an INIT COM statement. The common area that is initialized at this point is that of a user program where the INIT COM statement has been executed.

B5.2.3 Restrictions on the Use of Common Variables

It is not possible to use a common variable for any of the following parameters of statements.

- A device number in an I/O statement, such as ENTER/OUTPUT statements
- An I/O buffer variable in a transfer action
- A variable representative of specifications in an IMAGE statement
- A counter variable in a FOR ... NEXT statement

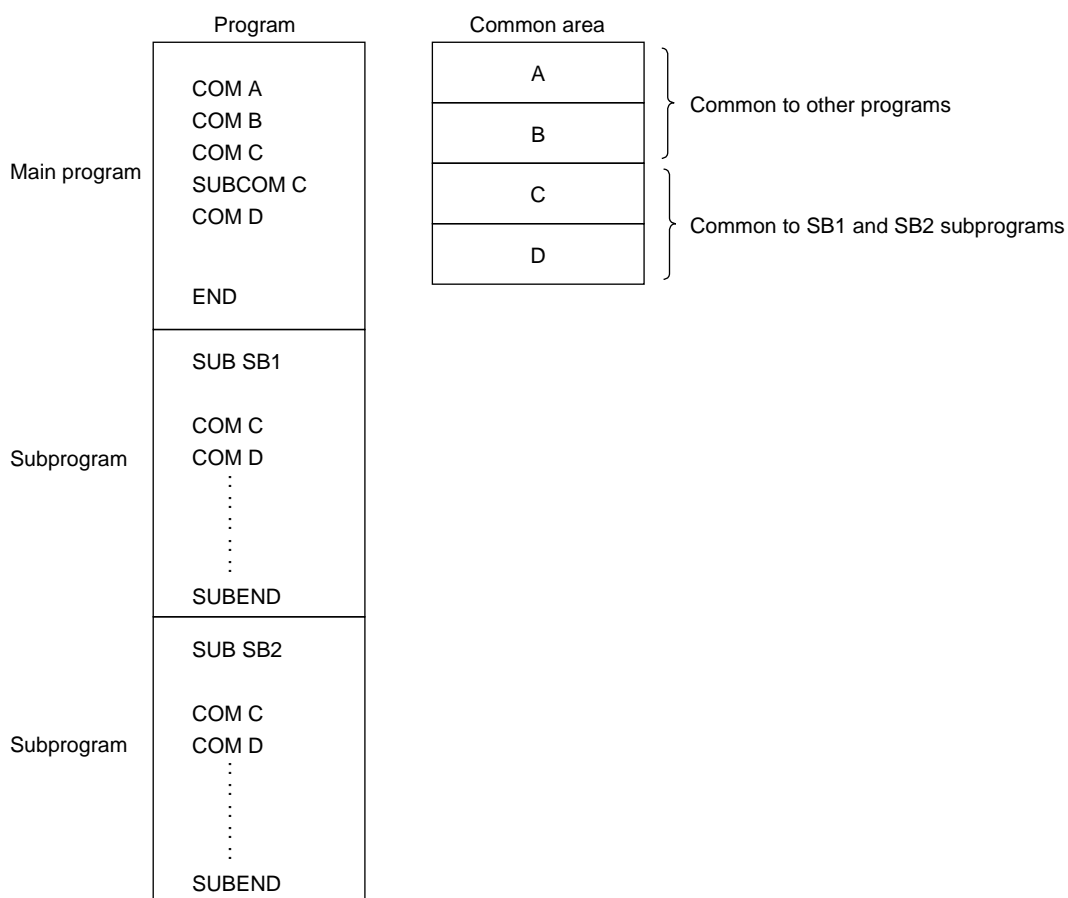
B5.3 Statements Related to COM Statement

B5.3.1 SUBCOM Statement

A SUBCOM statement is used when the common area needs to be separated into one shared with other programs (main programs) and one shared by the main program and subprograms. As shown in Example 1 below, common areas for subprograms are secured in sequence, beginning with the common area specified by the SUBCOM statement that immediately precedes it.

A SUBCOM statement is designed to be used within a main program. If you use more than one SUBCOM statement at the same time, the program may fall into an uncertainty state of operation for such reasons as interrupts. Do not use more than one SUBCOM statement at the same time (see Example 2).

Example 1:



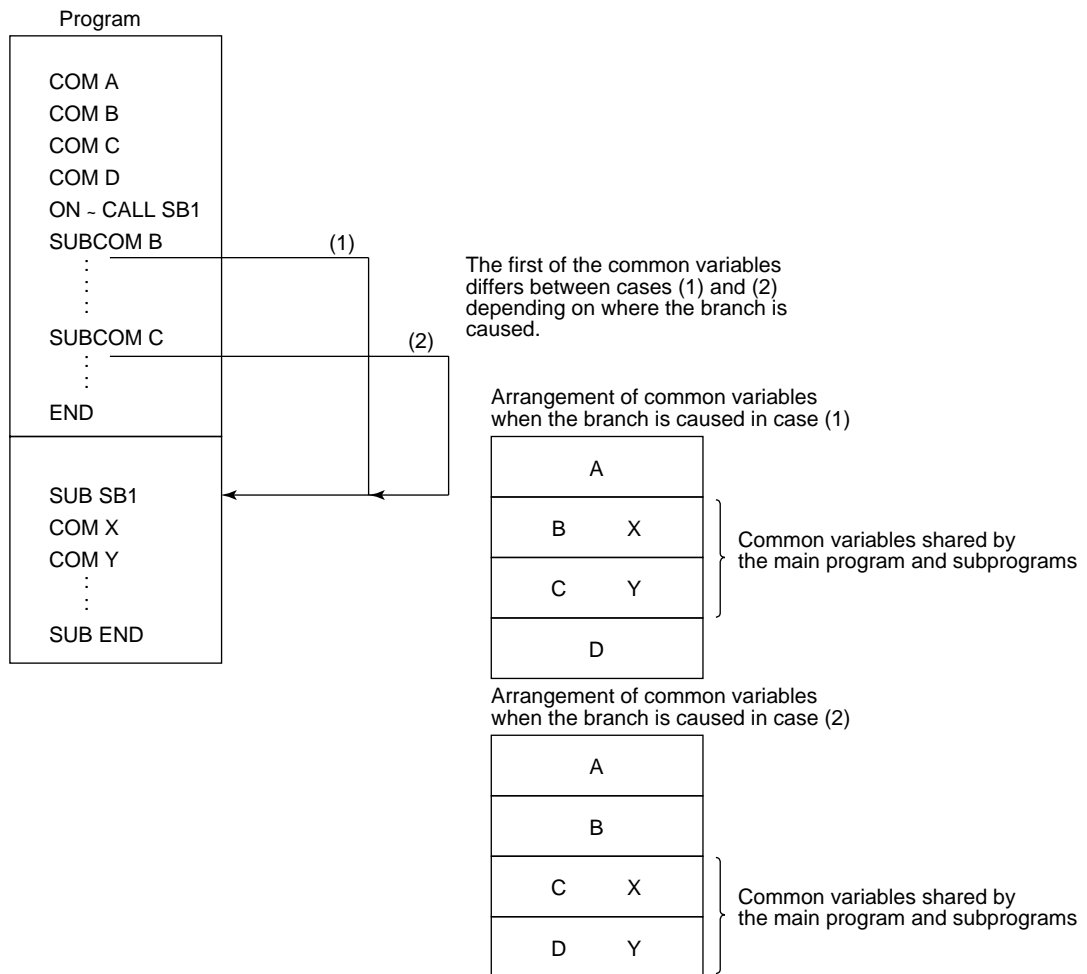
FB050301.EPS



CAUTION

It is not possible to use a SUBCOM statement together with a COM # statement that defines a common area for other programs.

Example 2:



FB050302.EPS

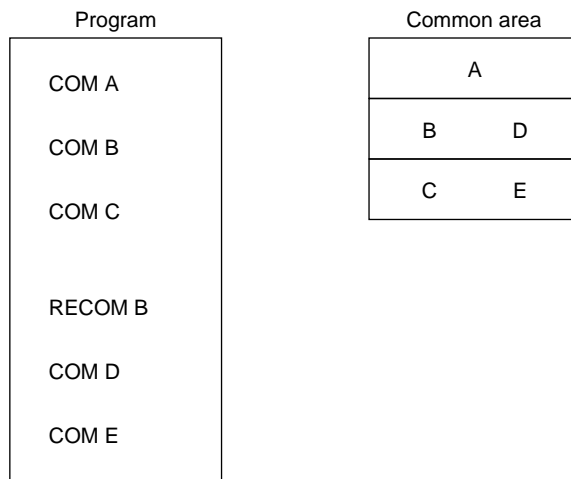
In Example 2 above:

- (1) The branch in case (1) results in the variable names B and X sharing the same data and the variable names C and Y sharing the same data, respectively.
- (2) The branch in case (2) results in the variable names C and X sharing the same data and the variable names D and Y sharing the same data, respectively.

B5.3.2 RECOM Statement

A RECOM statement is used when you specify the same location of a common variable under a different variable name. In the example shown below, COM statements that follow the RECOM B statement allocate variables, beginning with the variable B. In other words, a RECOM statement registers the same common variable under more than one name within the program's local common area. If a RECOM statement having no parameters is used, allocation begins with the first of variables in the common area. Any variable name declared by a COM statement can also be used after making a RECOM statement. In the example, the variables B and D share the same data.

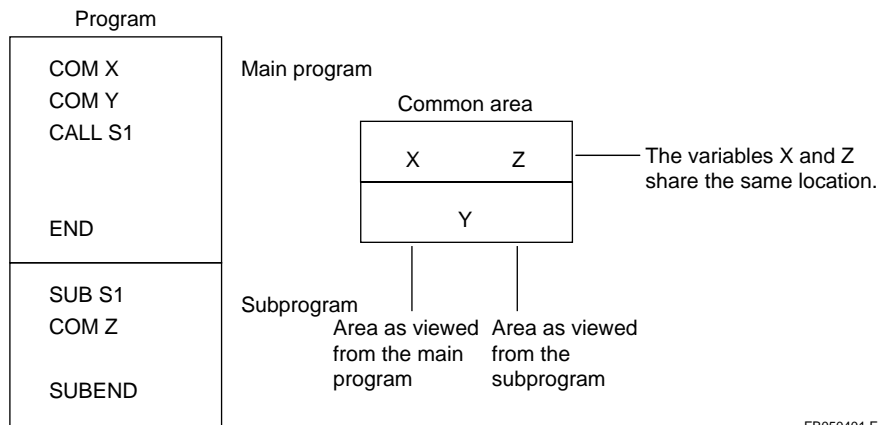
Although this example is given for a main program, a RECOM statement can also be used in exactly the same way for subprograms. Note that a RECOM statement can be used together with a COM # statement that defines common areas to be shared with other programs.



FB050303.EPS

B5.4 Data Exchange with Subprograms

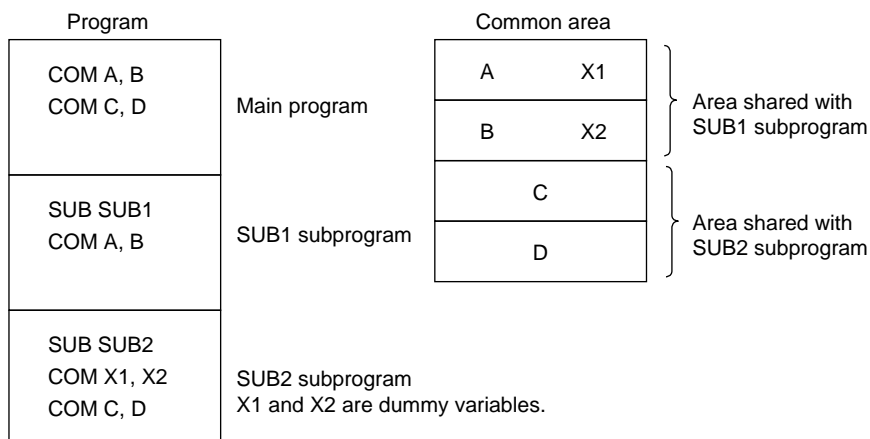
In data exchange between a main program and subprograms, each COM statement allocates common variables in sequence in the same area, as shown below.



FB050401.EPS

Common Area Shared with Two or More Subprograms

If you want the common area shared by more than one subprogram, allocate dummy variables before using the common area.



FB050402.EPS

B6. Data Exchange with a Ladder Sequence Program

B6.1 Data Exchange between CPU Modules

The FA-M3 multi-controller has the following functions necessary for a BASIC program and a ladder sequence program to exchange data with each other.

(1) Data Exchange

Data exchange between a BASIC program and a ladder sequence program can be achieved 1) by common variables and shared registers or 2) by using an ENTER or OUTPUT statement.

Data exchange using common variables and shared registers

- Statements in the BASIC program are executed irrespective of how the ladder sequence program is scanned.
- Since actual data exchange is carried out asynchronously, you must consider the timing at which data is transferred.
- This method has superior program readability and maintainability since data access can be achieved using variable names.
- Any change in data allocation can be coped with by simply changing the declarative statement.

Data exchange using an ENTER or OUTPUT statement

- It is possible to directly gain read/write access to all devices from a BASIC program.
- Since actual data exchange is carried out synchronously, you need not consider the timing at which data is transferred.
- The execution time of statements in the BASIC program depends on how the ladder sequence program is scanned.
- Since sequence devices are directly accessed from the BASIC program using device addresses, it is relatively difficult to change the program or use the program for other purposes. (In other words, the program maintainability is poor.)

(2) Synchronization between Programs (Interrupts)

Synchronization can be achieved between the BASIC program and the ladder sequence program in two ways:

- Using an ENTER or OUTPUT statement, read/write binary-digit data from/to the internal relays of sequence devices; or
- Apply interrupts from the ladder sequence program to the BASIC program.

In this case, execute an ON SEQVLT statement in advance in the BASIC program to declare acceptance of interrupts. To apply an interrupt from the ladder sequence program, use a SIGNAL command.

(3) Running/Stopping the Ladder Sequence Program

This function runs or stops the ladder sequence program.

(4) Status Reading

This function reads the operating status (RUN/STOP) of the ladder sequence program.

This chapter explains the functions discussed in items (1) to (4) above.

Table B6.1 lists the statements in a BASIC program that are used to exchange data between the BASIC program and ladder sequence program.

Statement in BASIC Program	Function Description
ASSIGN sequence ID=Sc	Declares use of a sequence CPU module from the BASIC program.
ENTER Sc, Device name's character-string expression; Input variable	Inputs data from sequence devices.
OUTPUT Sc, Device name's character-string expression; Output variable	Outputs data to sequence devices.
CONTROL Sc, 1 ; I	Starts or stops the ladder sequence program on the CPU module whose use has been declared.
STATUS Sc, 1 ; I	Read the operating and error status of the CPU module whose use has been declared.
COM #S Sc Common variable's name	Declares data access to be carried out between the BASIC program and the ladder sequence program using common variables.
ON SEQEVTV ...	Declares acceptance of interrupts from the ladder sequence program.
OFF SEQEVTV ...	Cancels the declaration of acceptance of interrupts from the ladder sequence program.
SEQACTV Sc, Block number; I	Starts or stops the ladder sequence program that has been divided into program blocks.

TB060101.EPS

Sc: Slot number
I: Start/stop setting.

B6.1.1 Data Exchange Using Common Variables

As explained earlier, data can be exchanged between a BASIC program and a ladder sequence program 1) by using common variables and shared registers or 2) by using an ENTER or OUTPUT statement. This subsection explains how to use common variables and shared registers for data exchange. For details on how to use an ENTER or OUTPUT statement for data exchange, see subsection B6.1.2, "Data Exchange Using an ENTER or OUTPUT Statement," later in Part B, "Description of YM-BASIC/FA."

B6.1.1.1 Sharing of Sequence Devices

The YM-BASIC/FA programming language is designed so that data can be shared between the BASIC CPU module and a sequence CPU module.

The data that can be shared are those of shared registers (up to 1024 units) defined in the data area shared by the CPUs. In a BASIC program, shared registers can be referenced/configured as BASIC variables.

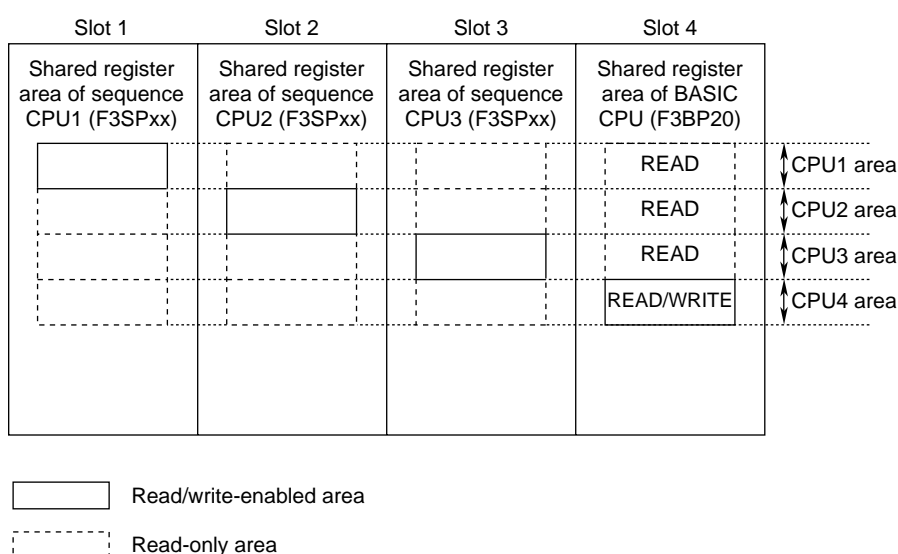


Figure B6.1 Shared Registers

The BASIC CPU module installed in slot 4 in such a system as shown in Figure B6.1 has read/write access to the CPU4 area only. It has only read access to the CPU1 to CPU3 areas.

TIP

A shared register is basically equivalent to a sequence common register installed in an FA500 controller. Be careful however, as they differ in the following points.

- The data area that can be shared differs between them, as shown below.

FA500 = B register; FA-M3 = R register

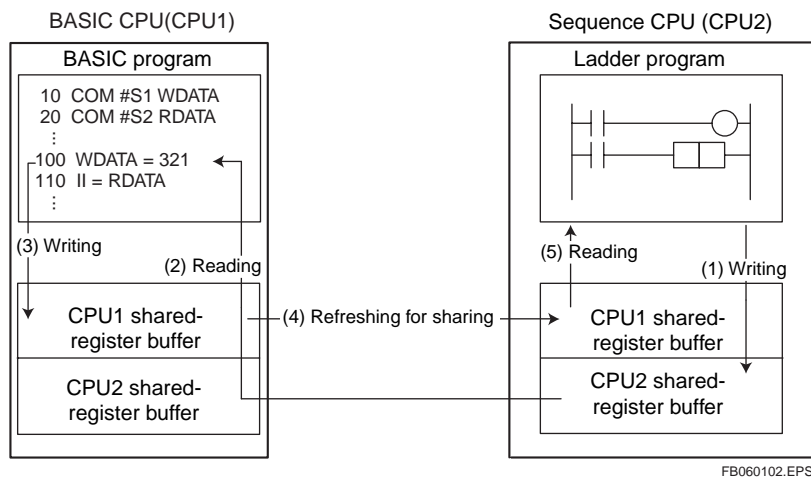
The size of shared registers is 1024 words, irrespective of the number of CPUs. The configurable area of each CPU is defined using the configuration function.

- Restrictions apply to the area to which data can be written, as noted below.

Each CPU has read/write access to its own local area. The CPU has only read access to the area of the other CPU.

B6.1.1.2 BASIC Common Variables and Sequence Devices

Carry out data exchange using the shared registers (R registers) within the sequence CPU and the BASIC common area. (Before data exchange can be carried out, an area [shared register area] for data exchange with the sequence CPU must be created within the BASIC CPU module using the configuration function.)



FB060102.EPS

Figure B6.2 Common Variables and Sequence Devices

Data flow from sequence CPU to BASIC CPU

- (1) The sequence CPU writes arithmetic results to the CPU2 shared-register buffer within its own CPU at the end of a scan.
- (2) The BASIC CPU reads data from the CPU2 shared-register buffer within the sequence CPU when executing a statement.

Data flow from BASIC CPU to sequence CPU

- (3) The BASIC CPU writes data to the CPU1 shared-register buffer within its own CPU when executing a statement.
- (4) The sequence CPU reads the CPU1 shared-register buffer within the BASIC CPU into its own CPU1 shared-register buffer, asynchronously, irrespective of how it is scanned (refreshing for sharing).
- (5) When refreshing for sharing is complete, the sequence CPU reads data in the CPU1 shared-register buffer into the shared registers, at a scan break, so that the data can be used for computing.

TIP

- If in a ladder program you have written values to the shared registers and send an interrupt telling the BASIC CPU that data has been written to the BASIC program, do so after having waited one scan after writing. This is because the BASIC CPU reads the result of the previous scan when a scan of the written data is being executed.

**CAUTION**

- Information on the allocation of shared relays and registers is managed separately by each individual CPU. For this reason every two CPUs that exchange data with each other must share the same information on the allocation of shared relays and registers. If the information differs between the two CPUs, data exchange may not be carried out correctly. The configuration of shared relays should be set in compliance with that of other CPUs.
- Shared devices in a sequence CPU are refreshed in asynchrony with scanning in units of one shared register (16 bits). For this reason the simultaneity of data exceeding the size of one shared register (16 bits) is not guaranteed. Be especially careful when using a long-integer variable (32 bits) or specifying the data using a whole array. If necessary, declare common variables in your application program.
- It is only possible to write to the local CPU's own area. When exchanging data with a sequence CPU, define write-only and read-only common variables separately.
- Sequence devices that can be used in a COM #S statement are shared registers only.

B6.1.1.3 COM #S Statement

Use a COM #S statement to declare data exchange based on common variables to be carried out with a specified sequence CPU. A COM #S statement has the following format. For more details on the format, see "COM Statement" in Part C, "Syntax of YM-BASIC/FA," later in this manual.

COM #S Slot number Common variable.....

Slot number: Slot where the CPU is installed

Common variable: Integer variable or long-integer numeric variable. Multiple common variables and common variables in an array are also acceptable.

Like other declarations of common variables, common variables declared by a COM #S statement are allocated to the shared registers of a specified sequence CPU, in the order in which they are stated in the program (see Figure B6.2). Variables that can be declared by a COM #S statement are integer and long-integer variables only. Using a variable of other types will result in an incorrect value. To be able to use an integer variable (16 bits), the variable must be declared to be of integer type in advance by a DEFINT statement. One 16-bit integer variable is equivalent to one shared register.

Likewise, to be able to use a long-integer variable (32 bits), the variable must be declared to be of long-integer type in advance by a DEFLNG statement. One 32-bit long-integer variable is equivalent to two shared registers.

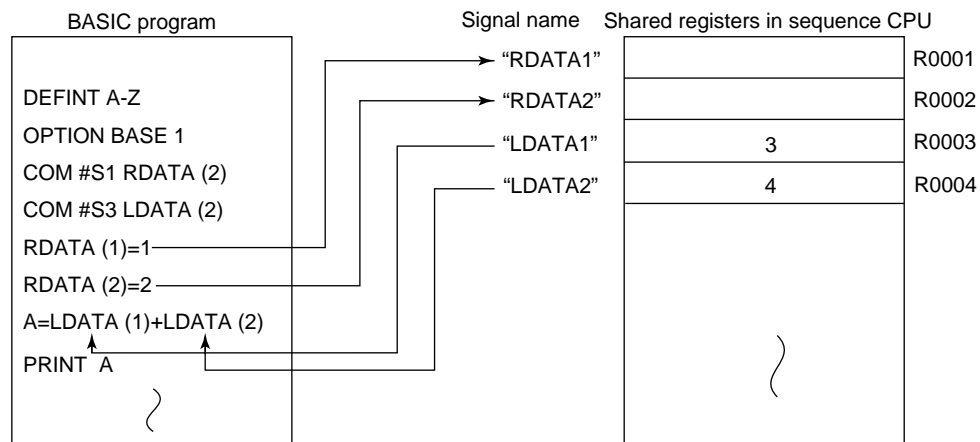
When using a COM #S statement, carry out shared-device configuration in advance using the BASIC Programming Tool M3 for Windows.

**CAUTION**

- Shared devices in a sequence CPU are refreshed in asynchrony with scanning in units of one shared register (16 bits). For this reason the simultaneity of data exceeding that of one shared register (16 bits) is not guaranteed. Be especially careful when using a long-integer variable (32 bits) or specifying the data using a whole array. If necessary, declare common variables in your application program.

Example 1:

From a BASIC program that runs on a CPU in slot 1, you declare data exchange with an add-on sequence CPU installed in slot 3. Any name may be used for common variables in the BASIC program. For example, you can use the signal names of shared registers in the ladder sequence program for the names of common variables in the BASIC program.



The allocation of shared registers in the example above is as follows.

R0001 and R0002: BASIC CPU

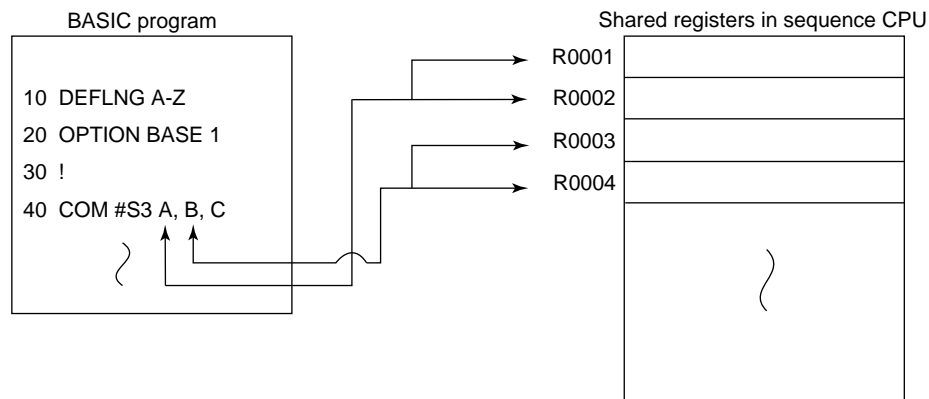
R0003 and R0004: Sequence CPU

FB060103.EPS

Figure B6.3 Allocation of Common Variables (1 of 2)

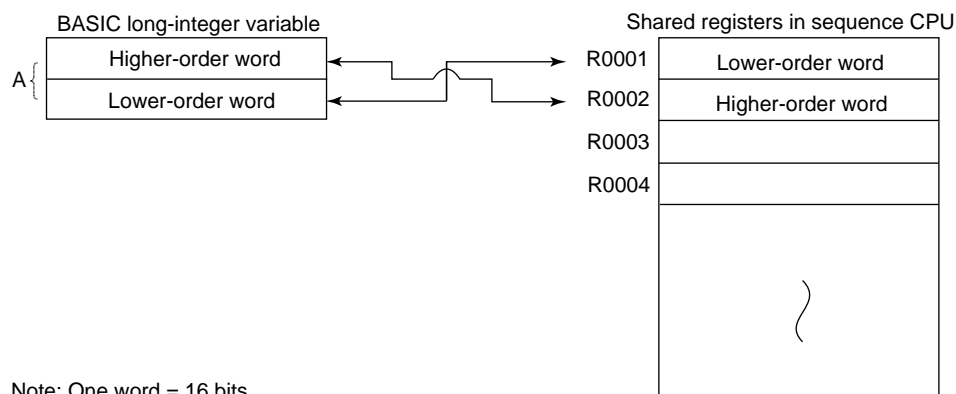
Example 2:

The correlation of long-integer variables with the shared registers is as follows.



FB060104.EPS

In addition, the correlation of the higher-order and lower-order words of a long-integer variable with the higher-order and lower-order words of a shared register is as follows.



Note: One word = 16 bits

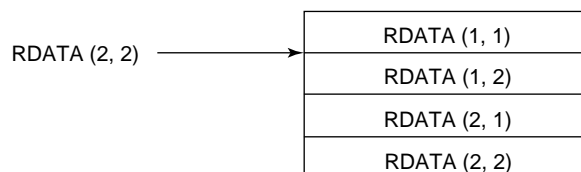
FB060105.EPS

Figure B6.4 Allocation of Common Variables (2 of 2)

**CAUTION**

- If any long-integer variable is used, the simultaneity of data in the higher- and lower-order words is not guaranteed. If necessary, declare common variables in your application program.

If a two-dimensional array of integer variables is used, the variables are allocated as shown below.



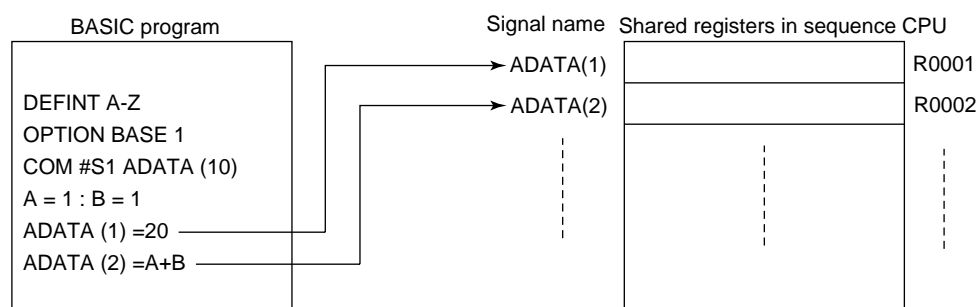
FB060106.EPS

Figure B6.5

After declaration of common variables, data is exchanged within the BASIC program as described below.

■ Data Output from BASIC Program to Sequence Program

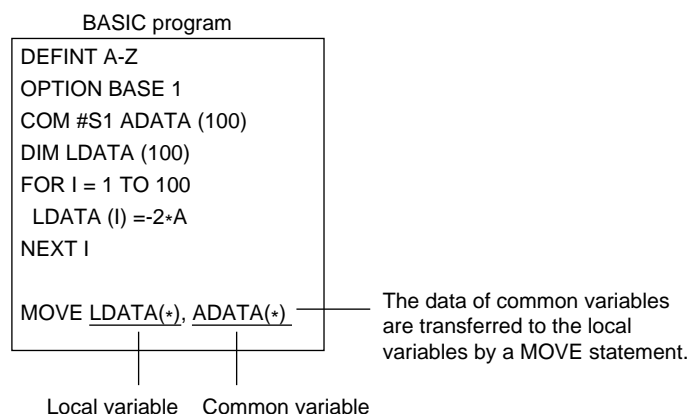
You can pass data to a ladder sequence program by simply inputting the data to a common variable.



FB060107.EPS

Figure B6.6

If the frequency of access to common variables in a BASIC program is extremely high, transfer the data of the common variables to the local variables of the BASIC program before processing the data. This strategy saves time consumed by gaining access to the common variables.

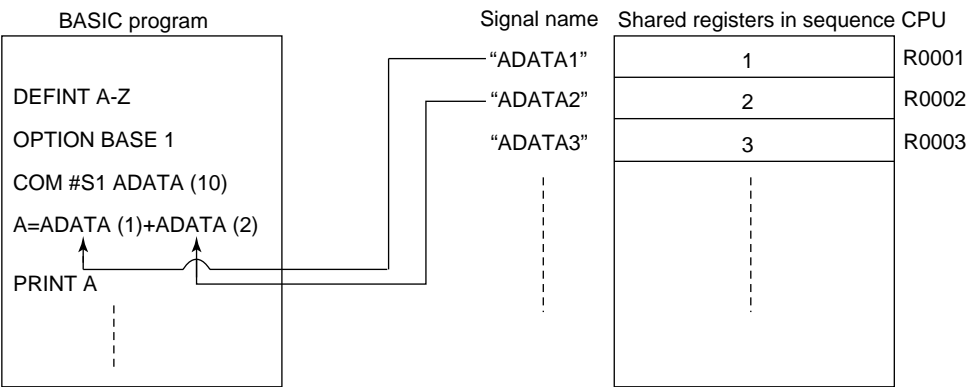


FB060108.EPS

Figure B6.7

■ Data Input from Sequence Program to BASIC Program

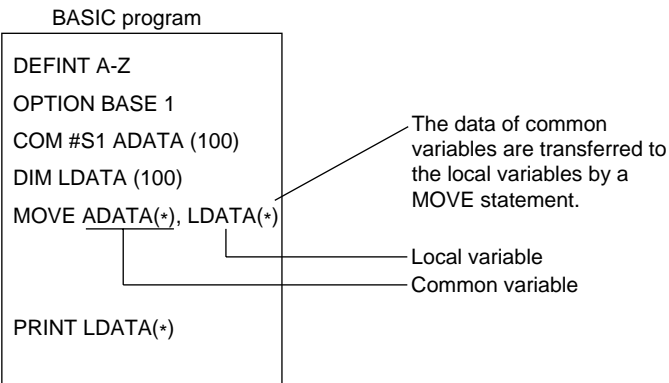
You can receive data from a ladder sequence program by simply referencing the given common variable.



FB060109.EPS

Figure B6.8

If the frequency of access to common variables in a BASIC program is extremely high, transfer the data of the common variables to the local variables of the BASIC program before processing the data. This strategy saves time by gaining access to the common variables.



FB060110.EPS

Figure B6.9

TIP

A local variable refers to a variable that is used within a particular single main program or subprogram. If you use a common variable in your program, you gain access to the ladder sequence program each time you use the variable. Too frequent access therefore involves a decrease in the processing speed of the BASIC program. Transfer the data of common variables to local variables when making a program so that the frequency of access to the common variables is minimized.

Common variables can be used as variables or elements in an array. When specifying the common variables using a whole array, pass them to local variables using a MOVE statement. This strategy enables the whole array to be transferred in a very efficient manner.

■ Data Exchange between Multiple Subprograms and a Ladder Sequence Program

To exchange data between a BASIC program having multiple subprograms and a ladder sequence program, use a COM #S statement for each program block. This method causes common variables to be allocated so that the first line of each program block corresponds to the first line of shared registers, as shown in Figure B6.10.

Note that common variables that are declared by a COM #S statement cannot be used with a SUBCOM statement. If the area needs to be shared among subprograms, allocate dummy variables to the COM #S statement so that the common variables are not used (see Figure B6.10).

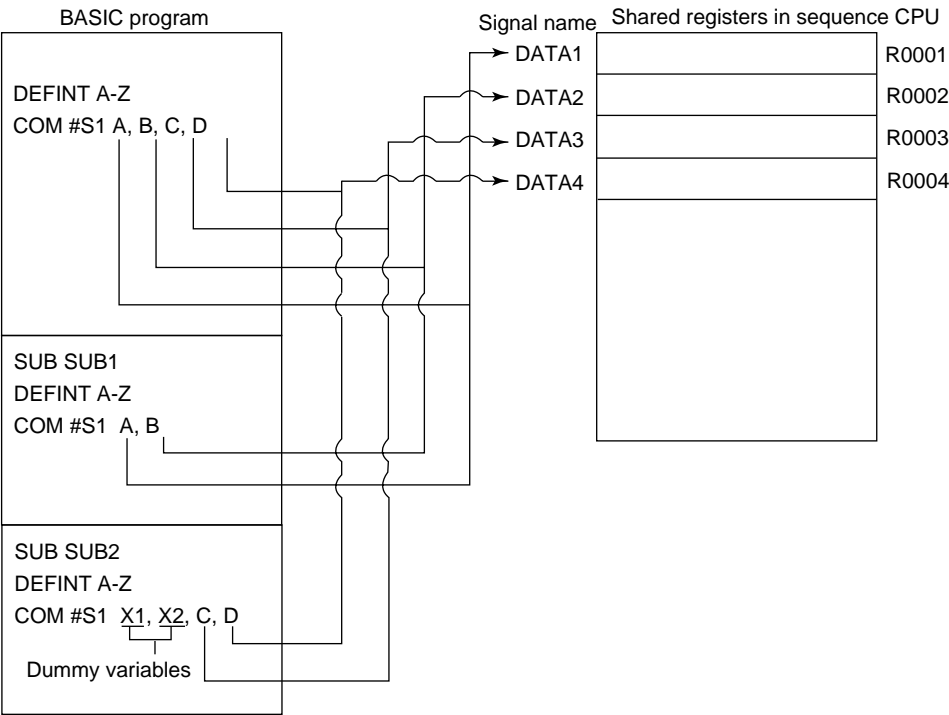
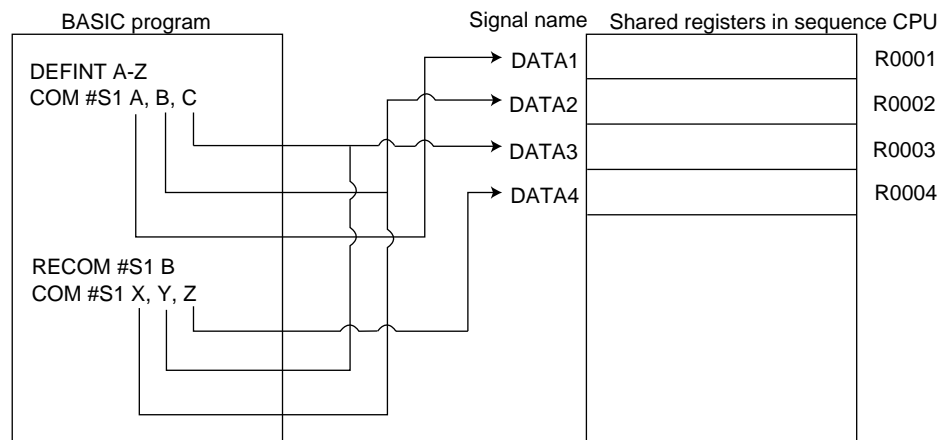


Figure B6.10

FB060111.EPS

■ RECOM #S Statement

A RECOM #S statement is used when you specify the same location for a common variable under a different variable name. Common variables thus stated can be used just like common variables declared by other COM statements. For more details on this statement, see “RECOM Statement” in Part C, “Syntax of YM-BASIC/FA,” later in this manual.



FB060112.EPS

Figure B6.11

■ Clearing Shared Registers

Shared registers are not cleared even if an INIT COM statement is executed. To initialize shared registers, use an INICOMM3 standard library or write 0 to the registers in the BASIC program or ladder sequence program.

■ Restrictions on the Use of Variables Declared by a COM #S Statement

Like other common variables, it is not possible to use a common variable declared by a COM #S statement as any of the following parameters of statements.

- A device number in an I/O statement, such as ENTER/OUTPUT statements
- An I/O buffer variable in a transfer action
- A variable representative of specifications in an IMAGE statement
- A counter variable in a FOR ... NEXT statement

■ Arguments of a Library

Any variable declared by a COM #S statement cannot be used as an argument of a library.

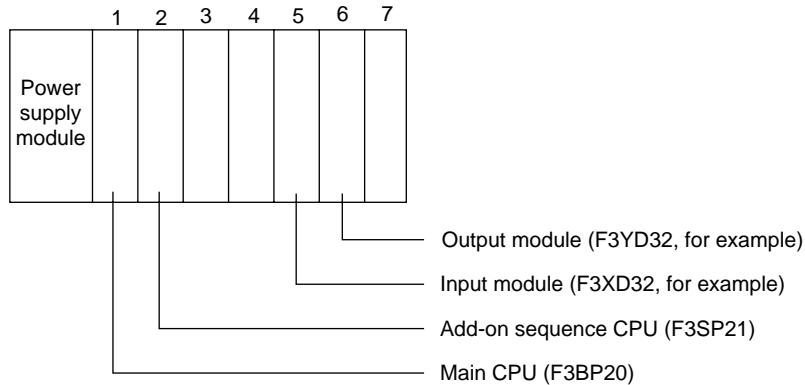
■ COM IS Statement

A COM IS statement cannot be used together with a COM #S statement.

B6.1.1.4 Example of Data Exchange

This item shows an example of how data is exchanged between a BASIC program and a ladder sequence program. In the following example, information on input relays acquired by the ladder sequence program is computed (the square root is evaluated) by the BASIC program. Then, the result is output externally from output relays by the ladder sequence program.

● Hardware Configuration



FB060113.EPS

● Programs

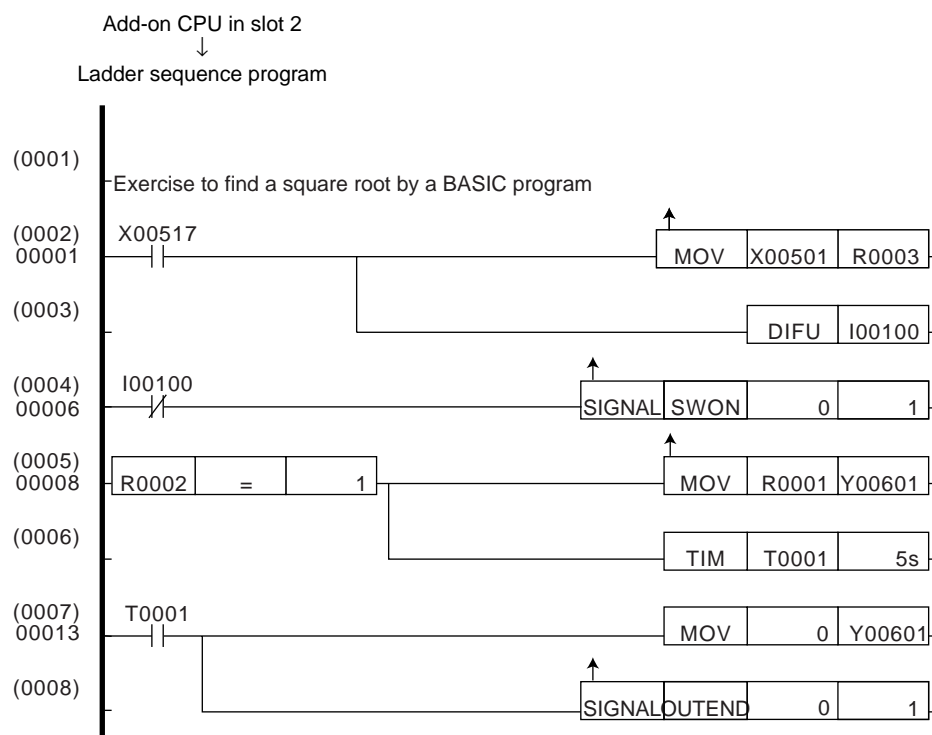
Main CPU in slot 1
↓
BASIC program

The allocation of shared registers in the example above is as follows.
R0001 and R0002: BASIC CPU
R0003 and R0004: Sequence CPU

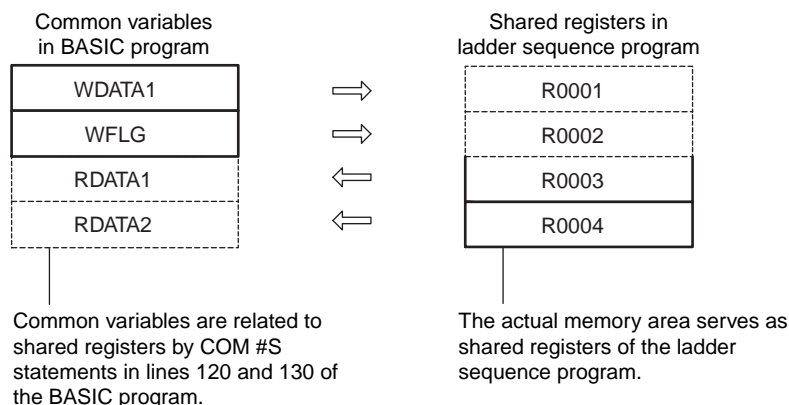
```

100 REM Exercise to Find a Square Root
110 DEFINT A-Z
120 COM #S1 WDATA1,WFLG
130 COM #S2 RDATA1,RDATA2
140 ASSIGN SP21 = 2
150 WFLG = 0
160 ON SEQEV "OUTEND" GOSUB OUTEND@
170 ON SEQEV "SWON" GOSUB SWON@
180 !
190 LOOP@
200 WAIT
210 GOTO LOOP@
220 STOP
300 SWON@
310 WDATA1 = INT(SQR(RDATA1))
320 WFLG = 1
330 RETURN
400 OUTEND@
410 WFLG = 0
420 RETURN
500 !
510 END

```

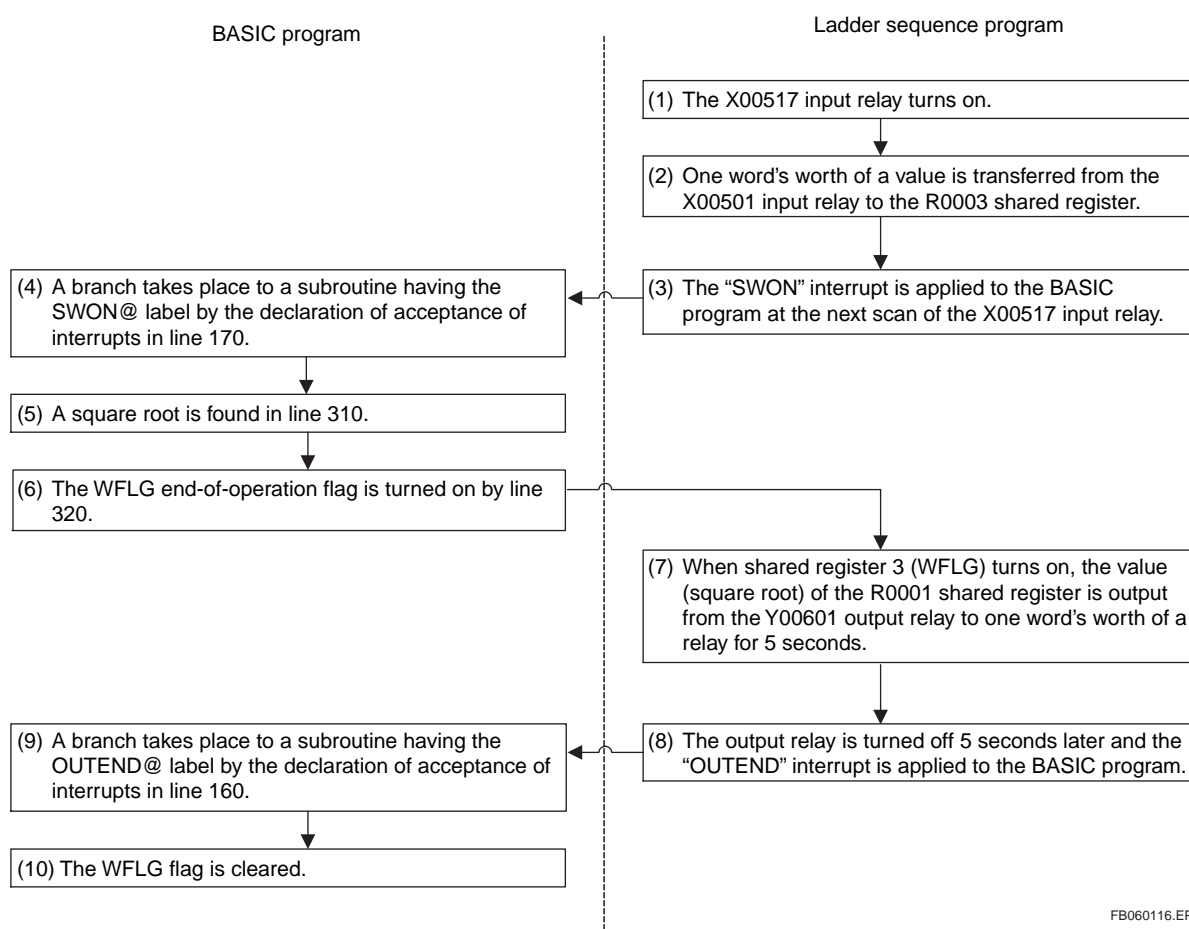


● Relationship between Common Variables and Shared Registers



FB060115.EPS

● Program Flow



FB060116.EPS

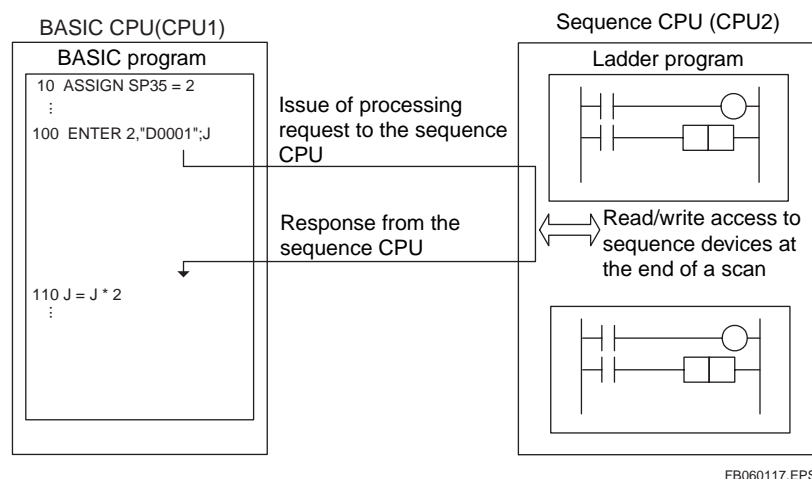
B6.1.2 Data Exchange Using an ENTER or OUTPUT Statement

From the BASIC program, you can read data from sequence devices in a sequence CPU or write data to the sequence devices, using an ENTER or OUTPUT statement. To be able to use these functions, you must use an ASSIGN statement to declare the use of a CPU module to which you gain read/write access in the first line of your program. By taking advantage of the function for writing data to sequence devices, you can achieve synchronization between the BASIC program and the ladder sequence program.

Another method of carrying out data exchange between the BASIC program and the ladder sequence program is to use common variables and shared registers. For details on this method, see subsection B6.1.1, "Data Exchange Using Common Variables," earlier in Part B, "Description of YM-BASIC/FA."

ENTER and OUTPUT statements are handled according to the following procedure.

- (1) When executing an ENTER (OUTPUT) statement, the BASIC CPU makes a processing request to the sequence CPU.
- (2) The sequence CPU accepts the processing request as a CPU service of the peripheral processing system and reads from (writes to) sequence devices at the end of a scan.
- (3) The sequence CPU returns a response to the BASIC CPU.
- (4) Upon receipt of the response from the sequence CPU, the BASIC CPU goes to the next statement.



Data Exchange based on an ENTER or OUTPUT Statement

B6.1.2.1 ENTER Statement

Using an ENTER statement, you can read the data of sequence devices in a ladder sequence program from a BASIC program. For details on the sequence devices from which data can be read, see item B6.1.2.3 later in Part B, "Description of YM-BASIC/FA."

An ENTER statement is in the following format.

ENTER Slot number, device name's character-string expression [NOFORMAT]; input variable

Slot number: Integer-type expression

Device name's character-string expression:

Character-string expression

This expression represents the device address of a sequence device.

Input variable: Simple numeric variable or array numeric variable

This variable is of integer, long-integer or character-string type, to which data of a sequence device is input.



CAUTION

Note the following when reading data from sequence devices using an ENTER statement.

- I/O relays (X and Y relays) from which values can be read using an ENTER statement are only those relays* whose use has been declared in the specified sequence CPU module. If the use of selected I/O relays has not yet been declared in the specified sequence CPU module, values that are read become uncertain.
- If reading/writing data using a 16-bit address (XmmnnW), specify nn as 01, 17, 33 or 49 (the first of devices names given at intervals of 16 bits). If reading/writing data using a 32-bit address (YmmnnL), specify nn as 01 or 33 (the first of devices names given at intervals of 32 bits).
- When reading data using an ENTER statement, the simultaneity of the data is guaranteed in units of:
 - statements, if data of any single sequence device is read;
 - statements also, if data of more than one sequence device is read; and
 - 32 devices,

if data of sequence devices that are consecutive from a specified device is read collectively. If data needs to be exchanged in units of any larger amounts, specify accordingly in your application program.

* Denote the I/O relays set to the BIN or BCD option in the DI/O Setting box of the Configuration menu of the Ladder Diagram Support Program M3.

The following examples are given to explain how to specify a device name's character-string expression and input variable in an ENTER statement.

■ Reading Data of Any Single Sequence Device

- (1) ENTER 1, "I0001"; I

This statement reads the ON/OFF status of the I0001 sequence device (internal relay) of the CPU installed in slot 1 into the variable I.

- (2) ENTER 2, "Z001"; D1

This statement reads the value of the Z001 sequence device (special register storing the latest scan time) of the CPU installed in slot 2 into the variable D1.

■ Reading Data of More than One Sequence Device

- (1) ENTER 1, "I0001 TP001 D0001"; I, J, K

This statement reads:

- the ON/OFF status of the I0001 sequence device (internal relay) into the variable I,
- the value of TP001 (timer's current value) into the variable J, and
- the value of D0001 (data register) into the variable K,

among the sequence devices of the CPU installed in slot 1.

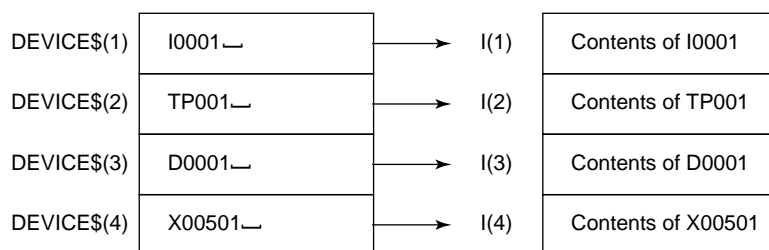
When specifying more than one sequence device at the same time, separate them with a space character () or comma (,). Specify each sequence device using its device address; you cannot use its signal name.

When reading more than one sequence device, you can also specify them as an array, as shown below. For variables in any single array, you can specify up to 32 sequence devices.

```

10  DEFINT AA|Z
20  OPTION BASE 1
30  !
40  DIM DEVICE$(8)
50  DIM I(4)
60  ASSIGN SP21=1
70  DEVICE$(1)="I0001 "
80  DEVICE$(2)="TP001 "
90  DEVICE$(3)="D0001 "
100 DEVICE$(4)="X00501 "
    :
    :
500 ENTER 1,DEVICE$(*);I(*)
    :
    :
800 END

```



FB060118.EPS

■ Collectively Reading Data of Sequence Devices That Are Consecutive from a Specified Device

(1) ENTER 1, "D0001*512" NOFORMAT ; I (*)

This statement collectively reads the values of 512 sequence devices that are consecutive from D0001 (data register), i.e., from D0001 to D0512 among the sequence devices of the CPU installed in slot 1, into the array I (*). If I is a one-dimensional array, the values are read into I (1) to I (512) (when an OPTION BASE 1 statement is used).

TIP

- Collective read-out must be carried out in units of 16 bits (one word) or 32 bits (two words). To do so, add *111 (where, 111 is the number of devices to be collectively read) to the device name's character string with which values are read in units of 16 or 32 bits using an ENTER ... NOFORMAT statement. For more details on the unit of read-out values and the device names' character strings, see item B6.1.2.3 later in Part B, "Description of YM-BASIC/FA."
- The number of devices that can be read in collective read-out is as follows.
512 maximum if the unit of read-out is 16 bits
256 maximum if the unit of read-out is 32 bits

B6.1.2.2 OUTPUT Statement

Using an OUTPUT statement, you can write data to sequence devices in a ladder sequence program from a BASIC program. For details on the sequence devices to which data can be written, see item B6.1.2.3 later in Part B, "Description of YM-BASIC/FA."

An OUTPUT statement is in the following format.

OUTPUT Slot number, Device name's character-string expression [NOFORMAT];
output variable

Slot number: Integer-type expression

Device name's character-string expression:

Character-string expression

This expression represents the device address of a sequence device.

Output variable: Simple numeric variable or array numeric variable

This variable is of integer, long-integer or character-string type, with which data is output to a sequence device.



CAUTION

Note the following when writing data to sequence devices using an OUTPUT statement.

- Do not use an OUTPUT statement to write data to the same sequence device as the one to which data is output using an OUT command in a ladder sequence program.
- A BASIC program is slower in processing than a ladder sequence program. For this reason data obtained by reading from a sequence device to which the data has been written using an OUTPUT statement may not immediately reflect the original data when the data is read into a ladder sequence program. If necessary, synchronize the BASIC program with the ladder sequence program using such devices as internal relays (see item B6.1.2.4 later in Part B, "Description of YM-BASIC/FA.")
- If reading/writing data using a 16-bit address (XmmmnW), specify nn as 01, 17, 33 or 49 (the first of devices names given at intervals of 16 bits). If reading/writing data using a 32-bit address (YmmmnL), specify nn as 01 or 33 (the first of devices names given at intervals of 32 bits).
- When writing data using an OUTPUT statement, the simultaneity of the data is guaranteed in units of:
 - statements, if data is written to any single sequence device;
 - statements also, if data is written to more than one sequence device; and
 - 32 devices,

if data is collectively written to sequence devices that are consecutive from a specified device. If data needs to be exchanged in units of any larger amounts, specify accordingly in your application program.

The following examples are given to explain how to specify a device name's character-string expression and output variable in an OUPUT statement.

■ Writing Data to Any Single Sequence Device

(1) OUTPUT 1, "I0001"; 1

This statement turns on the I0001 sequence device (internal relay) of the CPU installed in slot 1.

(2) OUTPUT 2, "D0001"; D1

This statement writes the value of the variable D1 to the D0001 sequence device (data register) of the CPU installed in slot 2.

■ Writing Data to More than One Sequence Device

OUTPUT 1, "I0001 TP001 D0001"; I, J, K

This statement writes:

- the value of the variable I into I0001 (internal relay),
- the value of the variable J into TP001 (timer's current value), and
- the value of the variable K into D0001 (data register),

among the sequence devices of the CPU installed in slot 1.

When specifying more than one sequence device at the same time, separate them with a space character () or comma (.). Specify each sequence device using its device address; you cannot use its signal name.

When writing to more than one sequence device, you can also specify them as an array, as shown below. For variables in any single array, you can specify up to 32 sequence devices.

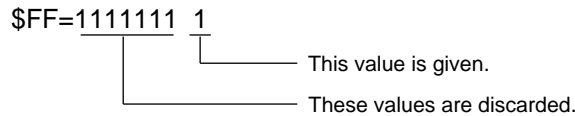
```

10  DEFINIT A-Z
20  OPTION BASE 1
30  !
40  DIM DEVICE$(4)
50  DIM I(4)
60  ASSIGN SP21=1
70  DEVICE$(1)="I0001 "
80  DEVICE$(2)="TP001 "
90  DEVICE$(3)="D0001 "
100 DEVICE$(4)="Y00601 "
    :
    :
500 OUTPUT 1,DEVICE$(*);I(*)
    :
    :
800 END

```

If data having two or more values are given to a sequence device using an OUTPUT statement, only the value of the data's least significant bit (LSB) is given; the values of other bits are discarded.

Example: OUTPUT 1, "I0001", \$FF



FB060119.EPS

■ Collectively Writing Data to Sequence Devices That Are Consecutive from a Specified Device

(1) OUTPUT 1, "D0001*512" 1NOFORMAT ; I (*)

This statement collectively writes the values of the array I (*) into 512 sequence devices that are consecutive from D0001 (data register), i.e., from D0001 to D0512 among the sequence devices of the CPU installed in slot 1.

TIP

- Collective writing must be carried out in units of 16 bits (one word) or 32 bits (two words). To do so, add *111 (where, 111 is the number of devices to be collectively written) to the device name's character string with which values are written in units of 16 or 32 bits using an ENTER ... NOFORMAT statement. For more details on the unit of written values and the device names' character strings, see item B6.1.2.3 later in Part B, "Description of YM-BASIC/FA."
- The number of devices that can be written in collective writing is as follows.
512 maximum if the unit of writing is 16 bits
256 maximum if the unit of writing is 32 bits

B6.1.2.3 Selectable Sequence Devices

Table B6.3 lists the sequence devices that can be specified in an ENTER or OUTPUT statement. Set a device name in the format shown under "Device name's Character-string Expression" in Table B6.3 in each device name's character-string expression (see B6.1.2.1 and B6.1.2.2 in Part B, "Description of YM-BASIC/FA) in an ENTER or OUTPUT statement. Be careful because some sequence devices have names identical to their device addresses, while others have names different from their device addresses. A device name's character string may also be stated in the following manner.

- **Alphabetic characters contained in a device name's character string may be either upper-case or lower-case.**

Example: In the following list, both sides result in the same designation.

I0001	÷	i0001
D0001	÷	d0001
D0001L	÷	d0001l
D0001*512	÷	d0001*512
TP001	÷	tp0001

- **Non-significant zero(es) in the numeric part of a device name's character string are omissible if the part is short of the given number of digits.**

Example: In the following list, both sides result in the same designation.

I0001	÷	I1
D0200	÷	D200
TP0005	÷	TP5
D0001*064	÷	D0001*64

Table B6.3 Sequence Devices That Can be Specified in an ENTER or OUTPUT Statement (1 of 3)

Sequence Device	Device Name (Address) and Range	Availability		Device Name's Character String	Unit of Reading/ Writing	Range of (Numeric) Data	Remarks
		ENTER	OUTPUT				
Input relay	F3SP21/25/35 X00201 to X71664	○	—	XmmmmnnW*III	16 bits	–32768 to 32767	mmm = Slot number nn = Terminal number III = Number of collectively accessed devices ● Xmmmmnn, Ymmmmnn One bit from the specified relay ● XmmmmnnW, YmmmmnnW One word (16 bits) from the specified relay ● XmmmmnnL, YmmmmnnL Two words (32 bits) from the specified relay The range of available relays depends on how the installed input module, output module, and sequence CPU are configured.
				XmmmmnnL			
				XmmmmnnL*III	32 bits	–2147483648 to 2147483647	
Output relay	F3SP21/25/35 Y00201 to Y71664	○	○	YmmmmnnW*III	16 bits	–32768 to 32767	
				YmmmmnnL			
				YmmmmnnL*III	32 bits	–2147483648 to 2147483647	
Internal relay	F3SP21 I0001 to I4096	○	○	Innnnn	1 bit	0:OFF, 1:ON	nnnn = Internal relay's number III = Device number up to which collective access applies The range of available devices depends on how the sequence CPU is configured.
	InnnnnW			16 bits	–32768 to 32767		
	F3SP25 I0001 to I8192			InnnnnW*III			
	F3SP35 I0001 to I16384			InnnnnL	32 bits	–2147483648 to 2147483647	
Shared (extended shared) relay	F3SP21 E0001 to E2048	○	○	Ennnnn	1 bit	0:OFF, 1:ON	nnnn = Shared relay's number III = Device number up to which collective access applies The range of available devices depends on how the sequence CPU is configured.
	EnnnnnW			16 bits	–32768 to 32767		
	F3SP25/35 E0001 to E4096			EnnnnnW*III			
	EnnnnnL			32 bits	–2147483648 to 2147483647		
Link relay	F3SP21 L00001 to L11024	○	○	LnnnnnW*III	16 bits	–32768 to 32767	nnnn = Link relay number III = Device number up to which collective access applies The range of available devices depends on how the sequence CPU is configured.
	LnnnnnL						
	F3SP25/35 L00001 to L71024			LnnnnnL*III	32 bits	–2147483648 to 2147483647	
Special relay	F3SP21 M0001 to M2048	○	○	MnnnnnW*III	16 bits	–32768 to 32767	nnnn = Special relay's number III = Device number up to which collective access applies The relays that can be used in an OUTPUT statement are only the special writable relays.
	MnnnnnL						
	F3SP25/35 M0001 to M9984			MnnnnnL*III	32 bits	–2147483648 to 2147483647	
Timer Continuous timer	F3SP21 T001 to T512	○	○	TUnnnn	1 bit	0:OFF, 1:ON	nnnn = Timer number III = Device number up to which collective access applies ● TUnnnn Time-out relay for the specified timer ● TSnnnn Setpoint of the specified timer ● TPnnnn Current value of the specified timer The range of available devices depends on how the sequence CPU is configured.
	F3SP25 T0001 to T2048		—	TSnnnn*III			
	F3SP35 T0001 to T3072		○	TPnnnn			
				TPnnnn*III	16 bits	0 to 32767	

TB060102.EPS

Table B6.3 Sequence Devices That Can be Specified in an ENTER or OUTPUT Statement (2 of 3)

Sequence Device	Device Name (Address) and Range	Availability		Device Name's Character String	Unit of Reading/ Writing	Range of (Numeric) Data	Remarks
		ENTER	OUTPUT				
Counter	F3SP21 C001 to C512	○	○	CUnnnn	1 bit	0:OFF, 1:ON	nnnn = Counter number III = Device number up to which collective access applies ● CUnnnn Count-up relay for the specified counter ● CSnnnn Setpoint of the specified counter ● CPnnnn Current value of the specified counter The range of available devices depends on how the sequence CPU is configured.
	F3SP25 C0001 to C2048		—	CSnnnn*III	16 bits	0 to 32767	
	F3SP35 C0001 to C3072		○	CPnnnn			
				CPnnnn*III			
Data register	F3SP21 D0001 to D5120	○	○	Dnnnn	16 bits	−32768 to 32767	nnnn = Data register number III = Device number up to which collective access applies ● Dnnnn One word (16 bits) from the specified register ● DnnnnL Two words (32 bits) from the specified register The range of available devices depends on how the sequence CPU is configured.
	Dnnnn*III						
	F3SP25/35 D0001 to D8192			DnnnnL	32 bits	−2147483648 to 2147483647	
				DnnnnL*III			
Link register	F3SP21 W00001 to W11024	○	○	Wnnnn*III	16 bits	−32768 to 32767	nnnn = Link register number III = Device number up to which collective access applies ● Wnnnn One word (16 bits) from the specified register ● WnnnnL Two words (32 bits) from the specified register The range of available devices depends on how the sequence CPU is configured.
	WnnnnL			32 bits	−2147483648 to 2147483647		
	WnnnnL*III						
Special register	F3SP21/25/35 Z001 to Z512	○	○	ZnnnnL	32 bits	−2147483648 to 2147483647	nnnn = Special register's number III = Device number up to which collective access applies ● Znnnn One word (16 bits) from the specified register ● ZnnnnL Two words (32 bits) from the specified register The devices that can be used in an OUTPUT statement are only the writable special registers.
	ZnnnnL*III						
Index register	F3SP21/25/35 V01 to V32	○	○	VnnL	32 bits	−2147483648 to 2147483647	nn = Index register number III = Device number up to which collective access applies ● Vnn One word (16 bits) from the specified register ● VnnL Two words (32 bits) from the specified register
				VnnL*III			

TB060103.EPS

Table B6.3 Sequence Devices That Can be Specified in an ENTER or OUTPUT Statement (3 of 3)

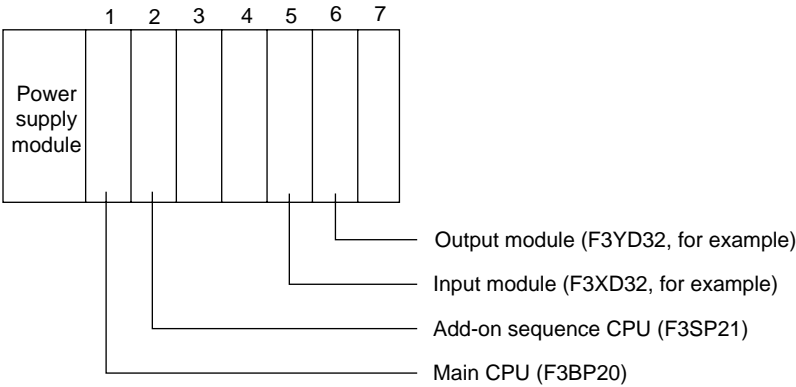
Sequence Device	Device Name (Address) and Range	Availability		Device Name's Character String	Unit of Reading/ Writing	Range of (Numeric) Data	Remarks
		ENTER	OUTPUT				
Shared (extended shared) register	F3SP21 R0001 to R1024	○	○	Rnnnn	16 bits	–32768 to 32767	nnnn = Shared register's number III = Device number up to which collective access applies ● Rnnnn One word (16 bits) from the specified register ● RnnnnL Two words (32 bits) from the specified register The range of available devices depends on how the sequence CPU is configured.
	Rnnnn*III						
	F3SP25/35 R0001 to R4096			RnnnnL	32 bits	–2147483648 to 2147483647	
	RnnnnL*III						
File register	F3SP25/35 B00001 to B32768	○	○	Bnnnnn	16 bits	–32768 to 32767	nnnn = File register number III = Device number up to which collective access applies ● Bnnnnn One word (16 bits) from the specified register ● BnnnnnL Two words (32 bits) from the specified register
				Bnnnnn*III			
				BnnnnnL	32 bits	–2147483648 to 2147483647	
				BnnnnnL*III			

TB060104.EPS

B6.1.2.4 Example of Data Exchange

This item shows an example of how data is exchanged between a BASIC program and a ladder sequence program. In the following example, information on input relays acquired by the ladder sequence program is computed (the square root is evaluated) by the BASIC program. Then, the result is output externally from output relays by the ladder sequence program.

● Hardware Configuration



FB060120.EPS

● Programs

Main CPU in slot 1

↓

BASIC program

```
100 REM Exercise to Find a Square Root
110 DEFINT A-Z
120 ASSIGN SP21=2
130 ON SEQEV T "SWON" GOSUB 190
140 !
150 WAIT
160 GOTO 150
170 STOP
180 !
190 ENTER 2, "D0003W";RDATA1
200 WDATA1=INT(SQR(RDATA1))
210 OUTPUT 2, "D0001 W";WDATA1
220 OUTPUT 2, "I0001";1
230 RETURN
240 !
250 END
```

Add-on sequence CPU in slot 2

↓

Ladder sequence program

(0001) Exercise to find a square root by a BASIC program

(0002) X00517 | MOV X00501 D0003

(0003) | SIGNAL SWON 0 1

(0004) I00001 | MOV D0001 Y00601

(0005) | TIM T0001 5s

(0006) T0001 | MOV 0 Y00601

(0007) | RST I00001

FB060121.EPS

B6.1.3 Synchronization between Programs

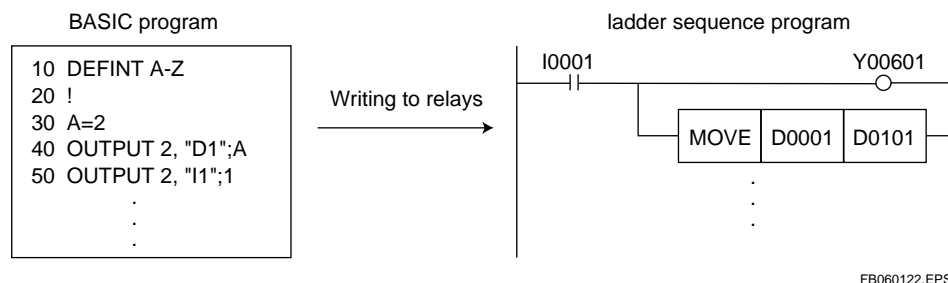
■ Synchronization Using Sequence Devices

Between CPUs in the same unit, you can gain read/write access to sequence devices from a BASIC program using an ENTER or OUTPUT statement (see B6.1.2.1 and B6.1.2.2 in Part B, "Description of YM-BASIC/FA"). By taking advantage of this function, you can achieve synchronization between the BASIC program and a ladder sequence program.

When synchronizing one program with another, internal relays among the sequence devices are normally used.

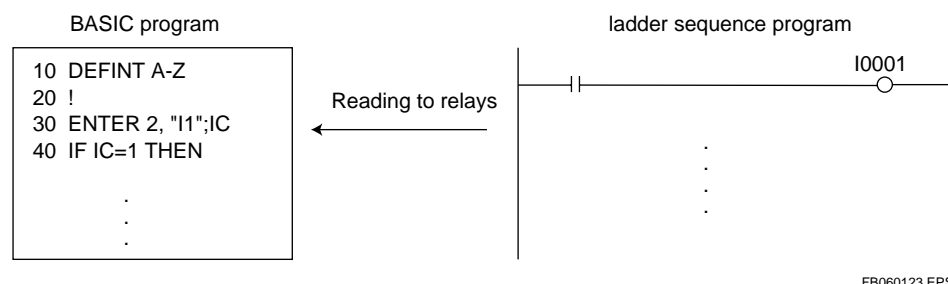
For example, consider a case where data is written to data registers using an OUTPUT statement and data exchange is carried out between the BASIC program and ladder sequence program. In that case, internal relays among the sequence devices are used to notify the timing of data exchange with each other. The following examples show how the internal relays are used.

Example 1:



In line 40 of the BASIC program, data is written to the data register D1 (D0001) by an OUTPUT statement. Then, the internal relay I1 (I0001) is set to 1 (ON) by an OUTPUT statement. From the fact that the internal relay I0001 has turned on, the ladder sequence program learns that data has been passed to the data register D0001. Thus it runs the processes that follow.

Example 2:



In line 30, the data of the internal relay I0001 is read from the add-on sequence CPU installed in slot 2 into the variable IC. In line 40 and subsequent lines, the program runs processes according to the content of the variable IC.

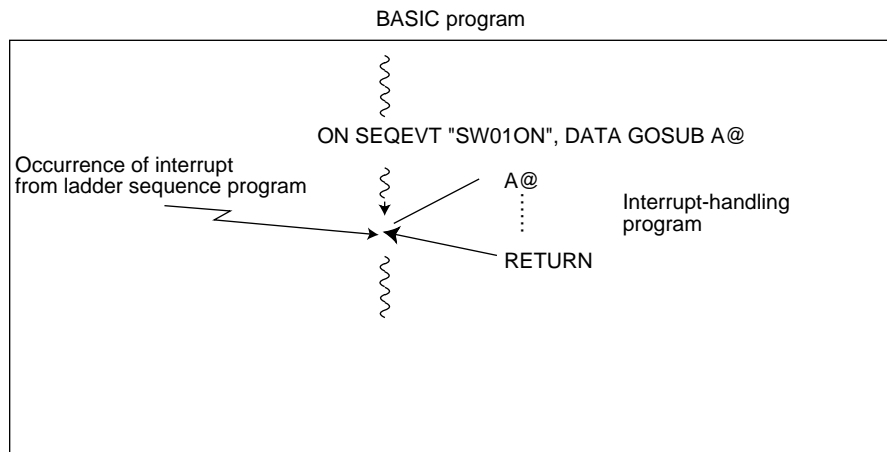
■ Synchronization Using Interrupts from a Ladder Sequence Program

In the FA-M3 multi-controller, information on the occurrence of an event can be sent from a ladder sequence program to a BASIC program to carry out interrupt handling. This enables synchronization to be achieved between the ladder sequence program and BASIC program. To be able to receive a message of the occurrence of an event from the ladder sequence program, acceptance of interrupts must be declared in the BASIC program in advance using an ON SEQVST statement. An ON SEQVST statement is in the following format. For more details on the format, see “ON SEQVST Statement” in Part C, “Syntax of YM-BASIC/FA,” later in this manual.

ON ___ SEQEVN___ Signal name [, Variable name]___ {
 GOTO } ___ { Line number }
 GOSUB } ___ { Label }
 CALL } ___ { Subprogram name }

FB060124.EPS

With the previous execution of the above-mentioned statement, the BASIC program is notified of an interrupt after the ladder sequence program has executed an instruction of interrupt. Consequently, a branch takes place to a process appropriate to the specified signal name. Note that the BASIC program can also receive integer-type values as data.



FB060125.EPS

To interrupt the BASIC program from the ladder sequence program, use a SIGNAL command. For more details on the command, see the Sequence CPU -Commands instruction manual (IM 34M6P12-03E).

B6.1.4 Precautions with Data Exchange

■ Measures When Data of Shared Registers Cannot Be Read Correctly

If the data of shared registers cannot be read correctly or is read with their values shifted, check the following.

- Configuration of shared registers and relays

Information on the allocation of shared registers and relays is managed separately by each individual CPU. For this reason every two CPUs that exchange data with each other must share the same information on the allocation of shared registers and relays. If the information differs between the two CPUs, data exchange may not be carried out correctly. The shared relays should also be configured in compliance with that of other CPUs.

- Declaration of common variables

When carrying out data exchange using common variables and shared registers, the common variables must be declared in advance using a COM #S statement. The common variables are allocated to the shared registers of a specified sequence CPU, in the order in which they are stated in the BASIC program.

- Data types of common variables

Variables that can be declared by a COM #S statement are integer and long-integer variables only. Using a variable of other types will result in an incorrect value.

- Starting number of suffix of array variables

Unless otherwise specified by an OPTION BASE statement, the suffixes of array variables begin with 0.

■ Refreshing of Shared Registers

Shared devices in a sequence CPU are refreshed in asynchrony with scanning. For this reason the simultaneity of data is not guaranteed. If necessary, refresh the shared devices in your application program.

■ Simultaneity of Data

● Data Exchange Using Common Variables and Shared Registers

Shared devices in a sequence CPU are refreshed in asynchrony with scanning in units of one shared register (16 bits). For this reason the simultaneity of data exceeding the size of one shared register (16 bits) is not guaranteed. Be especially careful when using a long-integer variable (32 bits) or specifying the data using a whole array. If necessary, declare common variables in your application program.

● Data Exchange Using an ENTER or OUTPUT Statement

When exchanging data using an ENTER or OUTPUT statement, the simultaneity of the data is guaranteed in units of:

- statements, if data is passed to any single sequence device;
- statements also, if data is passed to more than one sequence device; and
- 32 devices,

if data is collectively passed to sequence devices that are consecutive from a specified device. If data needs to be exchanged in units of larger amounts, specify accordingly in your application program.

■ Passing Data of Single-precision Real Number Type

The floating-point data of single-precision real number type differs between the IEEE representation of floating-point data used by a sequence CPU and the representation of floating-point data internal to the YM-BASIC/FA. When exchanging data of single-precision real number type, convert the IEEE representation to the YM-BASIC/FA internal representation or vice versa using an IFPCNV standard library.

The following program reads two of floating-point data items from a sequence CPU.

```

10  DEFINIT I
20  DEFLNG S,B
30  OPTION BASE 1
40  COM #S1 SP(2)
50  COM BP(2)
60  RECOM
70  COM FPIN(2),FPOUT(2)
80  !
90  MOVE SP(*),BP(*)
100 ICMD =1
110 FORM$ = "2F4"
120 IFPCNV(ICMD,FPIN(*),FORM$,FPOUT(*),IERR)
130 IF IERR <> 0 THEN STOP
140 DP FPOUT(1),FPOUT(2)
150 !
160 END

```

B6.2 Starting/Stopping a Ladder Sequence Program

A ladder sequence program can be started or stopped from a BASIC program in two ways, as explained below. To be able to use these functions, you must use an ASSIGN statement to declare the use of a CPU module where the ladder sequence program is started or stopped.

■ Starting/Stopping a Ladder Sequence Program

If a ladder sequence program is already loaded to the sequence CPU module, you can use a CONTROL statement to start the whole program. You can also stop the whole program by using a CONTROL statement.

■ Starting/Stopping a Ladder Sequence Program Block

If a ladder sequence program is already loaded to the sequence CPU module and the program is active, you can use a SEQACTV statement to start the program on a block-by-block basis. You can also stop the program on a block-by-block basis by using a CONTROL statement.

This section explains the two types of statements discussed above.

B6.2.1 Starting/Stopping a Ladder Sequence Program

When starting or stopping a ladder sequence program in a sequence CPU module specified from a BASIC program with an ASSIGN statement, use a CONTROL statement. The ladder sequence program to be started must have been loaded to the sequence CPU before this statement is executed.

The CONTROL statement has the following format. For more details on this statement, see "CONTROL Statement" in Part C, "Syntax of YM-BASIC/FA," later in this manual.

CONTROL Slot number, 1 ; Start/Stop setting

Select from the following values to specify "Start" or "Stop."

1 = Stop

2 = Start

Slot number: Slot where the CPU module whose use is declared in advance using an ASSIGN statement is installed

B6.2.2 Starting/Stopping a Ladder Sequence Program Block

When starting or stopping a ladder sequence program in a sequence CPU module on a block-by-block basis from a BASIC program, use a SEQACTV statement.

The SEQACTV statement has the following format. For more details on this statement, see "SEQACTV Statement" in Part C, "Syntax of YM-BASIC/FA," later in this manual.

SEQACTV Slot number, Block number ; Start/Stop setting

Specify either the Start or Stop setting by selecting a character from the following two options.

E = Stop

S = Start

Slot number: Slot where the CPU module whose use is declared in advance using an ASSIGN statement is installed

Block number: Integer-type expression

A program block refers to each individual ladder sequence program that is made using the Ladder Diagram Support Program M3 before the program's executable file is created.

B6.3 Reading the Operating Status of a Ladder Sequence Program

If you want to know whether a ladder sequence program is running or at a stop or whether there are any errors, read the operating status of the program using a STATUS statement. Before any STATUS statement can be used, you must declare the use of the sequence CPU in question using an ASSIGN statement. The sequence CPU for which a ladder sequence program is checked is the CPU whose use has been declared by an ASSIGN statement.

A STATUS statement has the following format. For more details on the STATUS statement, see "STATUS Statement" in Part C, "Syntax of YM-BASIC/FA," later in this manual.

STATUS Slot number, 1 ; Variable

Variable: Either of the following values is returned to the variable as the RUN or STOP status.

1: Stop

2: Run

Slot number: Slot where the sequence CPU module whose use has been declared by an ASSIGN statement is installed

B6.4 Error Codes

82-xx error codes that may appear during data exchange with a ladder sequence program have the meanings summarized in the following table.

Error Code (Hexadecimal)	Meaning	Cause	Corrective Actions
\$9D	No sequence CPU module installed yet	No sequence CPU module is recognized.	(1) Reset the CPU module. (2) Turn on and off the power.
\$E1	Device not ready	The sequence CPU module in question is defective.	(1) Check the way the module is mounted on the base unit. (2) Check the ALM and ERR lamps. Reset the CPU module. Replace the CPU module.
\$E2	Device busy	The sequence CPU module cannot accept statements from the BASIC program.	(1) Reset the sequence CPU module. (2) Turn on and off the power.
\$E6	Time-out error	The accepted statement cannot be coped with within the given length of time (2 seconds).	(1) Check the ALM and ERR lamps. (2) Reset the CPU module. Turn on and off the power.
\$F1	Error in statement execution check	The sequence CPU module is not ready to execute statements.	(1) If the ladder sequence program is not yet downloaded, download it. (2) Place the sequence CPU in the RUN mode. (3) Reset the sequence CPU module. Turn on and off the power.
\$F2	Error in statement execution result	The sequence CPU module is not in the specified state even if a statement is executed.	(1) Check the ALM and ERR lamps. (2) Reset the CPU module. Turn on and off the power.

TB060401.EPS

**CAUTION****About Time-out Error**

If the size a ladder sequence program in a sequence CPU module is relatively large, a time-out error may occur when:

- the power is turned on;
- online editing with the sequence CPU module is completed; or
- the ladder sequence program is downloaded.

Consequently, the BASIC program may come to a stop if access is made from the BASIC program to the sequence CPU.

A time-out error may be encountered during the handling of an ON ERROR statement when access is made from the BASIC program to the sequence CPU module. In that case, add a process that retries the statement in question. This strategy can prevent the BASIC program from coming to a stop. (One method for avoiding any time-out error during a power-on sequence is to use a WAIT statement in a BASIC program to have the program wait for a specific time length.)

The following program is an example of how a time-out error is avoided when the power is turned on.

```
10  DEFINIT A-Z
20  SLOT=1
30  ASSIGN SP21=SLOT
40  ON ERROR GOTO SEQWAIT@
50  ENTER SLOT,"M131";I
60  !
70  GOTO SEQST@
80  SEQWAIT@
90  IF ERRC=82 AND ERRCE=$E6 THEN GOTO 50
100 OFF ERROR
110 GOTO 50
120 !
130 SEQST@
140 OFF ERROR
150 !The original starting point of the program
```

B7. Methods of Access to I/O Modules

Input-output access to I/O modules installed in the FA-M3 versatile-range multi-controller can be made using a ladder sequence program or a BASIC program.

Be careful since you can gain only one-way access to some I/O modules, though this depends on the module type. The following table lists examples of such I/O modules. For details on modules that are not listed here or modules other than contact I/O modules, refer to the instruction manual of the respective modules.

Module Name	FA-M3 Multi-controller	
	BASIC Program*1	Ladder Sequence Program*2
Contact input module	Yes	Yes
Contact output module	Yes	Yes
Contact I/O module	Yes	Yes
Analog input module	Yes	Yes
Analog output module	Yes	Yes
Personal computer link module	—	— *3
RS-232-C communication module	Yes	—
RS-422 communication module	Yes	—
FA link module	— *4	Yes
High-speed counter module	Yes	Yes
Positioning module	Yes	Yes

TB070001.EPS

*1 "Yes" in this column indicates that the module can be accessed from a BASIC program.

*2 "Yes" in this column indicates that the module can be accessed from a ladder sequence program.

*3 Indicates that the module can access to a ladder sequence program.

*4 Access from a BASIC program can only be made to link relays and link registers among the ladder sequence devices. With a BASIC program alone, you cannot have direct access to any FA link module.

This chapter explains how access can be made from a BASIC program to I/O modules.

B7.1 Means of Access to I/O Modules

You can have input-output access to I/O modules using the following statements.

Statement	Action
ENTER	Input
OUTPUT	Output

TB070101.EPS

The following additional statements are available for some modules in order to make a declaration of use or set parameters.

(1) **ASSIGN**

Use of I/O modules must first be declared with this statement and then their slots must be defined.

(2) **RESET**

With this statement, the setup parameters of an I/O module are initialized.

(3) **CONTROL**

With this statement, parameters for such purposes as interrupt settings and behaviors specific to the module in question are configured.

(4) **ON INT and ENABLE INTR**

When interrupt functions are used, acceptance of interrupts must be declared with an ON INT statement.

For communication modules, an ENABLE INTR statement is also used to control interrupts.

(5) **ENTER and OUTPUT, and TRANSFER INTO and TRANSFER FROM**

With ENTER and OUTPUT statements, data is input to or output from an I/O module. For communications modules, TRANSFER INTO and TRANSFER FROM statements are also used to enable input/output access.

(6) **STATUS**

With this statement, the status of an I/O module or parameters configured by a CONTROL statement are referenced.

B7.2 Slot Number and Terminal Number

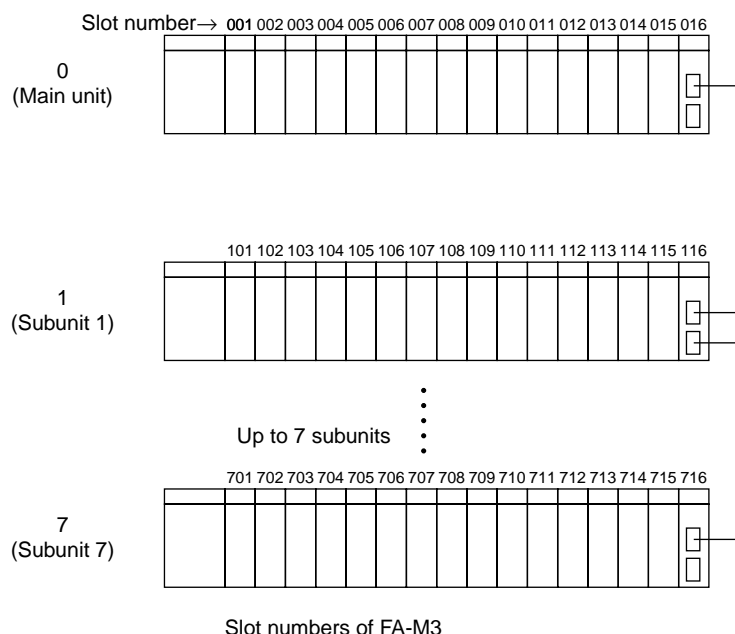
■ Slot Number

When access is made to each I/O module, its slot number and device number (comprising the port number, channel number, terminal number)* identify each interface.

- * Each I/O module can deal with more than one signal. To discriminate between these multiple signals, the port, channel and terminal numbers are used depending on the type of module, as shown below.
- Contact I/O modules: Terminal number
 - Analog I/O modules: Channel number
 - Serial communication modules: Port number

All these numbers are collectively referred to as a device number.

A slot number is a 3-digit integer and allocated as shown below.



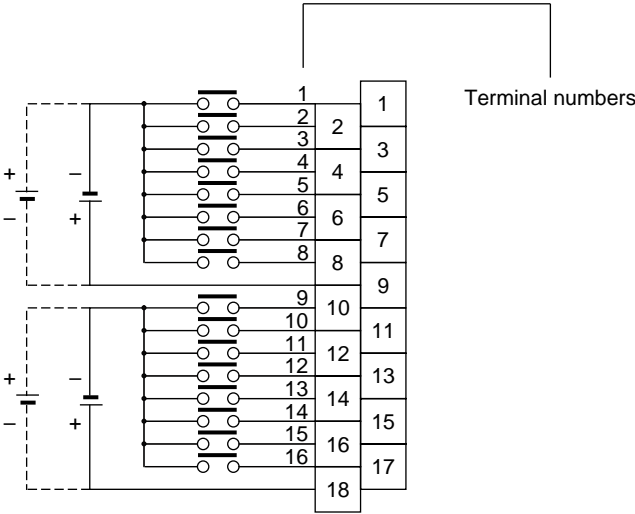
FB070201.EPS

In the case of a BASIC program, any non-significant zeroes preceding the actual slot number may be omitted.

■ Terminal Number

A terminal number is given to each terminal of an I/O module, as shown below. For more details on the terminal number, see the “FA-M3 Hardware Manual” (IM 34M6C11-01E).

Example: Terminal numbers of F3XD16-3N module



FB070202.EPS

B7.3 Declaring Use of I/O Modules

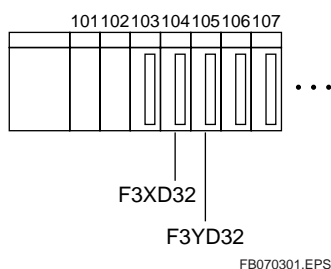
When using I/O modules in a BASIC program, you must first execute an ASSIGN statement to state which module is installed in which slot. The ASSIGN statement has the following format.

ASSIGN Module ID=n, Module ID2=n,

Module ID: Character string representative of the type of module

n: Slot number

Example: ASSIGN XD32=104, YD32=105



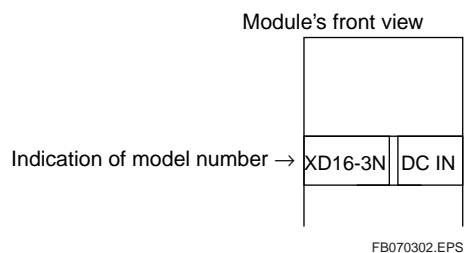
The module ID and sequence ID are character strings that represent the type of module and are used as part of the model number of each module. The following table lists some examples of module and sequence IDs. A sequence ID denotes a CPU module where a ladder sequence program can run, while a module ID identifies an I/O module.

FA-M3 Multi-controller	
Module's Model Number	Module ID or Sequence ID
F3SP21-0N	SP21
F3SP25-2N	SP25
F3SP35-5N	SP35
F3XD16-3N	XD16
F3YD32-1A	YD32
F3AD04-0N	AD04
F3XP02-0H	XP02
F3RS22-0N	RS22

TB070301.EPS

For each module, the first four characters of a model number shown on the module identifies the module ID or sequence ID.

Example: F3XD16-3N module



CAUTION

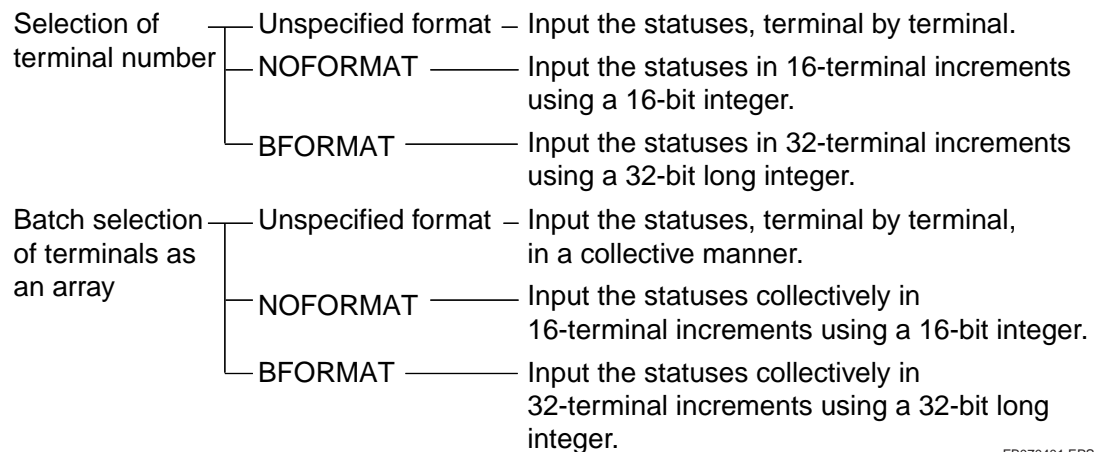
In the case of multiple-CPU system configuration, any single output module (and any advanced module having Y□□□□ output relays) cannot be shared by two or more CPUs. When using such an output module or advanced module with a BASIC program, set the Used/Unused option to Unused in the Configuration menu of the unused sequence CPU module.

B7.4 Access to Contact I/O Modules

This section details how to gain contact-I/O access in a BASIC program. For details on access to analog I/O modules or special modules such as communication modules, refer to the instruction manual of the respective modules.

B7.4.1 Contact Input Modules

A contact input module is used to input the ON/OFF status of contacts. Use an ENTER statement to input the contact status. Input 1 for the ON status of an input terminal and 0 for the OFF status of an input terminal. The status of each input terminal can be input by 1) selecting terminal numbers on a terminal-by-terminal basis, 2) making a batch selection of terminal numbers, or 3) making a batch selection of terminals using a 16-bit integer or a 32-bit long integer, as described below.



FB070401.EPS

(a) Selection of Terminal Number

This method is suited for reading the status of a specified terminal. This method uses the following formats.

● Terminal-by-terminal Input

The following statement reads the status of the specified terminal only.

ENTER m, n;P

m: Slot number in numeric expression

n: Terminal number in numeric expression

P: Input variable (numeric)

Example: ENTER 305, 2;P

This statement inputs the status of terminal 2 of a module in slot 305 into the variable P.

● 16-terminal Collective Input

The following statement collectively inputs the statuses of 16 terminals, which are consecutive from the specified terminal, into the input variable.

ENTER m, n NOFORMAT;I

m: Slot number in numeric expression

n: Terminal number in numeric expression, where n = 1, 17, 33 or 49

I: Input variable (integer)

Example: ENTER 305, NOFORMAT;I

This statement inputs the statuses of terminals from 17 to 32 of a module in slot 305 into the numeric integer variable I.



CAUTION

Use an integer variable for I; do not use a long-integer variable. The functionality of an ENTER statement is not guaranteed if you use a long-integer variable.

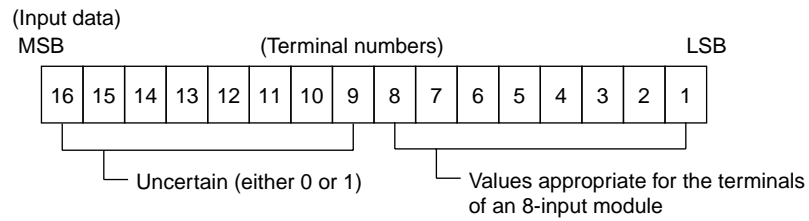
The relationship between the variable that stores input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of the variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.

	MSB	(Terminal numbers)																LSB
If n = 1:	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		
If n = 17:	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17		
If n = 33:	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33		
If n = 49:	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49		

FB070402.EPS

If you attempt to read the statuses of terminals from a contact input module having less than 16 terminals, the values of bits whose terminal numbers are higher than the module's maximum terminal number become uncertain (either 0 or 1).

Example: F3XA08-□N module



FB070403.EPS

● 32-terminal Collective Input

The following statement collectively inputs the statuses of 32 terminals, which are consecutive from the specified terminal, into the input variable.

ENTER m, n BFORMAT;L

m: Slot number in numeric expression

n: Terminal number in numeric expression, where n = 1 or 33

L: Input variable (long-integer)

Example: ENTER 305, 33 BFORMAT;L

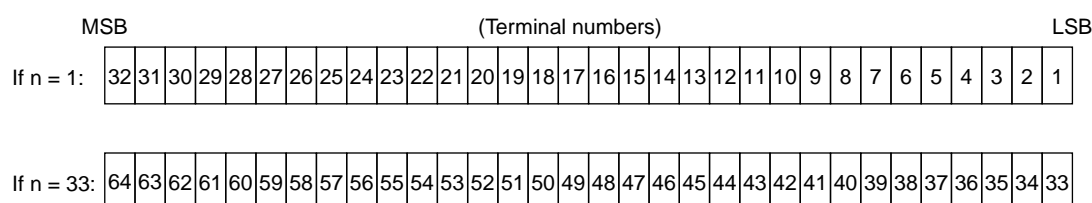
This statement inputs the statuses of terminals from 33 to 64 of a module in slot 305 into the long-integer variable L.



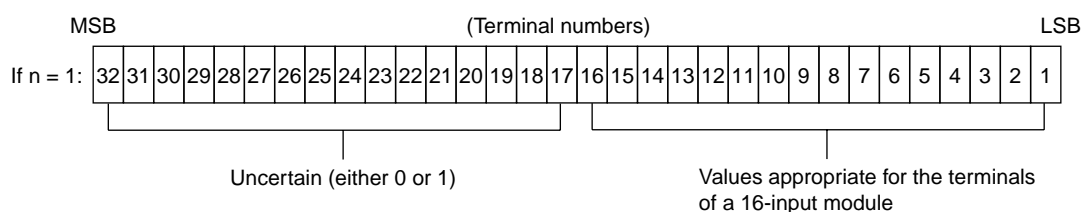
CAUTION

Use a long-integer variable for L; do not use an integer variable. The functionality of an ENTER statement is not guaranteed if you use an integer variable.

The relationship between the variable that stores input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of the variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.



If you select BFORMAT for the format field for a 16-input module, values are input as right-justified, beginning with the LSB.



(b) Batch Selection as an Array

This method is suited for reading the statuses of all input terminals. No terminal number should be specified. The method includes collective input on a terminal-by-terminal basis and collective input in 16-terminal increments using a 16-bit integer or in 32-terminal increments using a 32-bit integer. This method uses the following formats.

● Terminal-by-terminal Collective Input

The method inputs the statuses of all terminals as a value of either 0 or 1 and has the following format.

ENTER m;P(*)

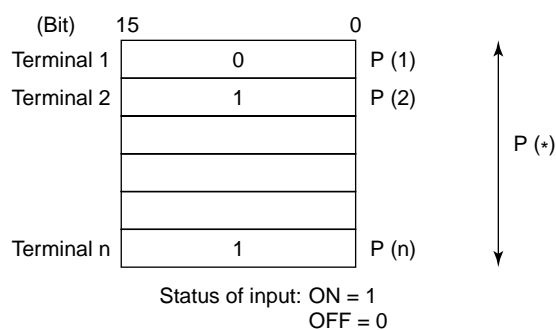
m: Slot number in numeric expression

P (*): Input variable list for collectively specifying numeric array variables

This list requires as many elements as the number of terminals whose statuses are collectively input. If the number of elements in the array is smaller than the number of terminals of a specified module, this statement inputs as many statuses as the number of elements, beginning with the lowest terminal number.

Example: ENTER 204;P(*)

This statement inputs the statuses of all input terminals of a module in slot 204 into the numeric array variable P (*) (for an OPTION BASE 1 statement).



FB070406.EPS

● Collective Input in 16-terminal Increments Using a 16-bit Integer

This method inputs the statuses of all input terminals in 16-terminal increments into the input variable. Use either of the following formats.

ENTER $_m$ $_NOFORMAT$;I1[, I2, I3, I4]

ENTER $_m$ $_NOFORMAT$;I(*)

m: Slot number in numeric expression

In: Input variable list for integer-type, simple variables

I (*): Input variable list for collectively specifying integer-type, array variables



CAUTION

Use integer-type simple variables and integer-type array variables for In and I (*), respectively; do not use long-integer type simple variables and long-integer type array variables. The functionality of an ENTER statement is not guaranteed if you use long-integer type simple variables or long-integer type array variables.

The relationship between the variables that store input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of each variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.

• Example of Integer-type Simple Variables

(Input data)	MSB	(Terminal numbers)																LSB
I ₁		16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	
I ₂		32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	
I ₃		48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	
I ₄		64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	

FB070407.EPS

I₂ is only valid for a 32- or 64-input module, while I₃ and I₄ are only valid for a 64-input module.

• Example of Integer-type Array Variables (for an OPTION BASE 1 Statement)

(Input data)

	MSB	(Terminal numbers)																LSB
I (1)	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		
I (2)	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17		
I (3)	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33		
I (4)	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49		

FB070408.EPS

I (2) is only valid for a 32- or 64-input module, while I (3) and I (4) are only valid for a 64-input module.

If you attempt to read the statuses of terminals from a contact input module having less than 16 terminals, the values of bits whose terminal numbers are higher than the module's maximum terminal number become uncertain (either 0 or 1).

Example: F3XA08-□N module

(Input data)

	MSB	(Terminal numbers)																LSB
	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		

Uncertain (either 0 or 1)

Values appropriate for the terminals of an 8-input module

FB070409.EPS

● Collective Input in 32-terminal Increments Using a 32-bit Long Integer

This method inputs the statuses of all input terminals in 32-terminal increments into the input variable. Use either of the following formats.

ENTER \underline{m} \underline{B} FORMAT;L1[, L2]

ENTER m BFORMAT;L(*)

m: Slot number in numeric expression

Ln: Input variable list for long-integer type, simple variables

L (*): Input variable list for collectively specifying long-integer type, array variables



CAUTION

Use long-integer type simple variables and long-integer type array variables for L_n and L (*), respectively; do not use integer-type simple variables and integer-type array variables. The functionality of an ENTER statement is not guaranteed if you use integer-type simple variables or integer-type array variables.

The relationship between the variables that store input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of each variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.

- Example of Long-integer Type Simple Variables

(Input data)

	(Terminal numbers)																																
MSB																													LSB				
L ₁	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	

L_2	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

L_2 is only valid for a 64-input module.

FB070410.EPS

- Example of Long-integer Type Array Variables (for an OPTION BASE 1 Statement)

(Input data)

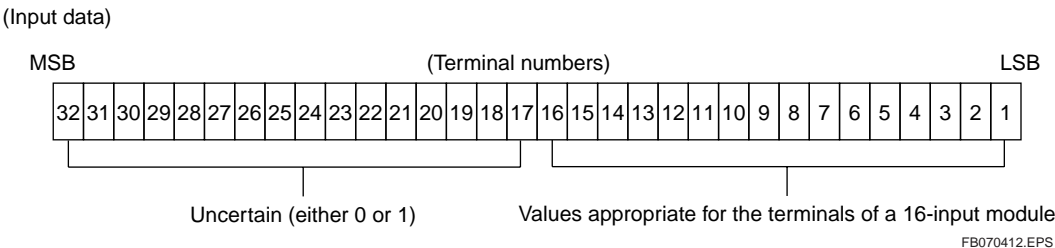
	MSB																(Terminal numbers)																LSB															
L(1)	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1																

L(2)	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

L (2) is only valid for a 64-input module.

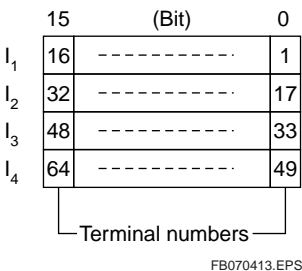
FB070411.EPS

If you select BFORMAT for the format field for a 16-input module, values are input as right-justified, beginning with the LSB. This is true for both long-integer simple variables and long-integer array variables.



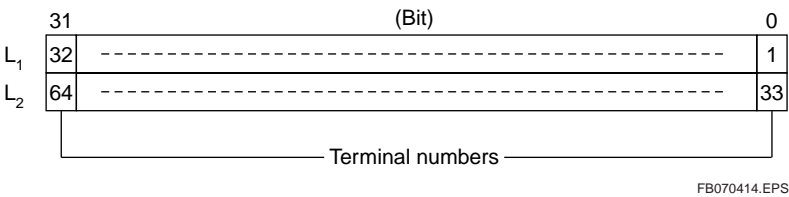
Example: ENTER 204 NOFORMAT;I₁, I₂, I₃, I₄

This statement inputs the statuses of all input terminals of a 64-input module in slot 204 into the integer-type numeric variables I₁, I₂, I₃ and I₄.



ENTER 205 BFORMAT;L₁, L₂

This statement inputs the statuses of all input terminals of a 64-input module in slot 205 into the long-integer type numeric variables L₁, and L₂.



B7.4.2 Interrupt from a Contact Input Module

For interrupts to a BASIC program, you can use any type of contact input module, except for a 64-point contact input module. An F3XH04-3N high-speed input module for the FA-M3 multi-controller can also be used to interrupt a BASIC program if the module is set to the Interrupt Function option. This subsection explains two cases of interrupt input, i.e., interrupt input from a standard contact input module and interrupt input from an F3XH04-3N high-speed input module.

B7.4.2.1 Interrupt from a Contact Input Module

You can cope with irregular events in a real-time manner by accepting interrupt input. However, a 64-point contact input module has no interrupt function. To be able to accept interrupts, you must set interrupt conditions using a CONTROL statement and declare acceptance of interrupts using an ON INT statement. Furthermore, use a STATUS statement if you want to read the status of interrupt or the data of the interrupt conditions.

(a) Setting Interrupt Conditions

Specify the edge type and define the sampling interval for interrupts using a CONTROL statement. Set these conditions for the terminals of each contact input module collectively in 8-terminal increments. You cannot set the conditions on a terminal-by-terminal basis. The ON/OFF state of each of these terminals is sampled at a fixed interval and retained in input sampling registers as a value of 1 for the ON state and 0 for the OFF state.

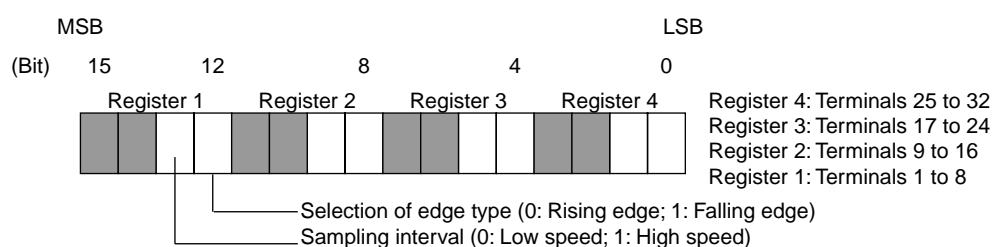
Specifying the edge type means you determine whether an interrupt should occur when the state of a terminal retained in a given input sampling register changes from 0 to 1 (rising edge) or from 1 to 0 (falling edge). The sampling interval can be selected from the low speed (16 ms) and high speed (1 ms). The CONTROL statement has the following format.

CONTROL m, 1;l

m: Slot number in a numeric expression

l: Data value set as an integer or integer-type variable

Define the data of control registers in 8-terminal increments, as shown below. Upon system startup, all of the control registers are set to 0.



FB070415.EPS

Note that the functionality of the CONTROL statement is not guaranteed if you set data such that the unused bits (2, 3, 6, 7, 10, 11, 14 and 15) of control registers are set to the ON state (1).

Example: CONTROL 1, 1;\$3333

This statement sets the interrupt conditions of all terminals to the High Speed option for the sampling interval and to the Falling Edge option for the edge type.

(b) Interrupt Declaration/Cancellation

● Interrupt Declaration

Using an ON INT statement, declare that a change (specified in a CONTROL statement) in the state of input to terminal n is accepted as an interrupt. In this statement, you can also specify the destination of a branch in your program so that the program flow is changed by the occurrence of an interrupt. The ON INT statement has the following format.

$$\text{ON } \underline{\hspace{1cm}} \text{ INT } \underline{\hspace{1cm}} \text{ m, n } \left\{ \begin{array}{l} \text{GOTO} \\ \text{GOSUB} \\ \text{CALL} \end{array} \right\} \left\{ \begin{array}{l} \text{Label} \\ \text{Line number} \\ \text{Subprogram name} \end{array} \right\}$$

FB070416.EPS

m: Slot number in a numeric expression

n: Terminal number in a numeric expression

Example: ON INT 205, 3 GOSUB 100

This statement causes a branch to a subroutine from line number 100 to handle an interrupt that is input to terminal 3 of a module in slot 205.

In the case of contact input modules, once a declaration has been made, interrupts are accepted until an interrupt declaration is cancelled.

● Interrupt Cancellation

Use an OFF INIT or RESET statement to cancel the interrupt declaration. The OFF INIT statement has the following format.

OFF INT m, n

m: Slot number in a numeric expression

n: Terminal number in a numeric expression

For details on the RESET statement, see item (d).

(c) Reading the Interrupt Status and Interrupt-conditions Data

Using a STATUS statement, you can read the interrupt status and interrupt-conditions data. The interrupt status is indicated by the value of an interrupt-handling auxiliary flag register. The auxiliary flag register changes from 0 to 1 as its corresponding input sampling register makes a specified state transition, either from 0 to 1 (rising edge) or from 1 to 0 (falling edge). This auxiliary flag register changes to 0 if read once and retains 0 until the value of the input sampling register changes the next time.

In contrast, the interrupt-conditions data is indicated by the value of a control register. The STATUS statement has the following format.

STATUS $\underline{\hspace{0.5cm}}$ m, n;In

m: Slot number in a numeric expression

n: Register number in a numeric expression

where

n = 1, if a control register is read; or

n = 101 for terminals 1 to 16

n = 102 for terminals 17 to 32

n = 103 for terminals 33 to 48, or

n = 104 for terminals 49 to 64, if the interrupt status is read.

In: Integer-type variable

The data of control registers (see item (a)) are input to the variable I_1 .

Example: STATUS 304, 1; I_1

This statement reads the interrupt-conditions data of all of the terminals of a module in slot 304 into the variable I_1 .

DP HEX\$ (I_1) to 2222

The interrupt status, on the other hand, is read by inputting the values of auxiliary flag registers as integers to the variables I_1 to I_4 , in 16-terminal increments. After having been read by a STATUS statement, the auxiliary flag registers are reset to 0. These behaviors can be represented by a flowchart, as shown in Figure B7.1.



CAUTION

Any terminal for which acceptance of interrupts has been declared by an ON INT statement causes its auxiliary flag register to reset to 0 immediately after the occurrence of an interrupt. The register therefore always reads 0 whenever it is read using a STATUS statement.

Example: STATUS 304, I01;I₁

This statement reads the statuses of terminals 1 to 16 of a module in slot 304 into the integer-type variable I₁.

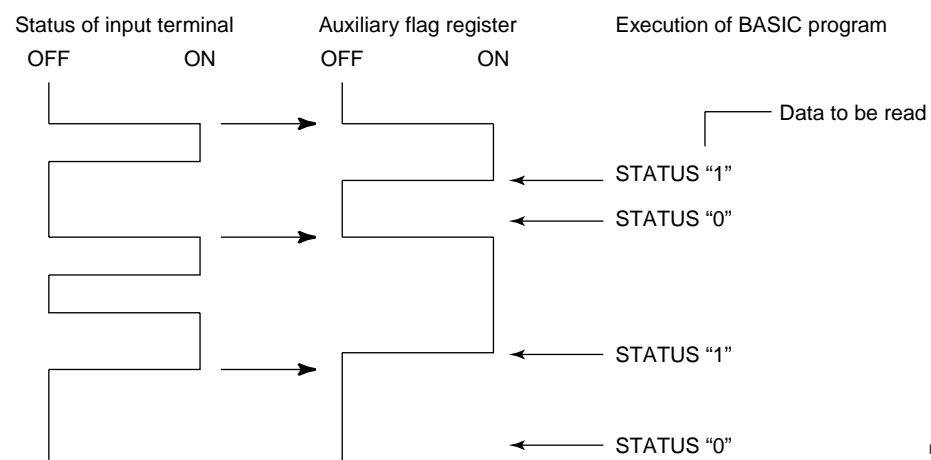
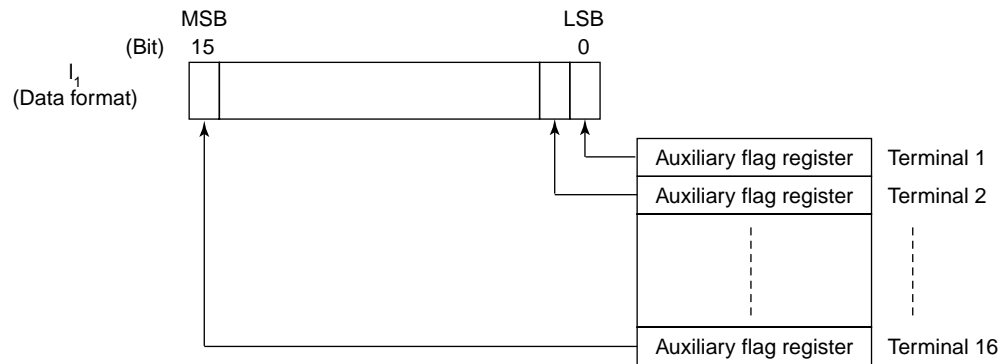


Figure B7.1 Flowchart Showing How Status Is Input to the Variable

FB070417.EPS

(d) Initializing the Interrupt-conditions Data and Cancelling the Interrupt Declaration

Use a RESET statement to initialize all interrupt-conditions data (into a value of 0) defined by a CONTROL statement. Furthermore, use an ON INT statement to cancel all interrupt declarations that have been made. The RESET statement has the following format.

RESET m

m: Slot number in a numeric expression

Example: RESET 407

This statement cancels the interrupt declaration made for the module in slot 407 and initializes all the interrupt-conditions data values to 0.

TIP

To cancel an interrupt declaration on a terminal-by-terminal basis, use an OFF INT statement. Note however, that only interrupt declarations can be cancelled by an OFF INT statement. The statement cannot initialize interrupt-condition data defined by a CONTROL statement.

B7.4.2.2 Interrupt Input from a High-speed Input Module

An F3XH04-3N high-speed input module (hereinafter simply referred to as a high-speed input module) for the FA-M3 multi-controller can also be used to interrupt a BASIC program if the module is set to the Interrupt Function option. For details on how to set the interrupt function, see the “FA-M3 Hardware Manual” (IM 34M6C11-01E).

Table B7.1 BASIC Statements Available for a High-speed Input Module

Function	Statement Format	Description
Declaration of use of modules	ASSIGN XH04=m m: Slot number	Defines the slot number where the module in question is installed.
Read-out of interrupt status of each terminal	STATUS m, I01:I m: Slot number I: Variable where the reading is stored	Reads the interrupt status of a high-speed input module installed in slot m and stores it in the variable I.
Interrupt declaration	ON INT m, n GOTO ON INT m, n GOSUB ON INT m, n CALL m: Slot number n: Input relay's number (1 to 4)	Declares that an OFF-to-ON transition in the input relays X□□□01 to X□□□04 of a high-speed input module installed in slot m is accepted as an interrupt.
Interrupt cancellation	OFF INT m, n m: Slot number n: Input relay's number (1 to 4)	Cancels interrupt declarations made by an ON INT statement on a terminal-by-terminal basis.
Module initialization	RESET m m: Slot number	Cancels interrupt declarations made by an ON INT statement on a module-by-module basis.

TB070401.EPS



CAUTION

- The functionality of any of the statements listed in Table B7.1 above is not guaranteed if you set the high-speed input module to the Pulse-capture Function option.
- The functionality of any statement other than those listed in Table B7.1 is not guaranteed if it is used with a high-speed input module.

(a) Declaration of Use of Modules

This statement declares the use of a high-speed input module. Use an ASSIGN statement for this purpose. Be sure to execute this statement before you use any other statements for the module. In this statement, define the number of the slot where the module is installed. The ASSIGN statement has the following format.

ASSIGN XH04=m

m: Slot number in a numeric expression

(b) Interrupt Declaration/Cancellation

● Interrupt Declaration

Using an ON INT statement, declare that a change in the state of input to terminal n is accepted as an interrupt. In this statement, you can also specify the destination of a branch in your program so that the program flow is changed by the occurrence of an interrupt. The ON INT statement has the following format.

$$\text{ON } \underline{\text{INT}} \underline{m}, \underline{n} \begin{cases} \text{GOTO} \\ \text{GOSUB} \\ \text{CALL} \end{cases} \begin{cases} \text{Label} \\ \text{Line number} \\ \text{Subprogram name} \end{cases}$$

FB070418.EPS

m: Slot number in a numeric expression

n: Terminal number in a numeric expression

Example: ON INT 205, 3 GOSUB 100

This statement causes a branch to a subroutine from line number 100 to handle an interrupt that is input to terminal 3 of a module in slot 205.

In the case of contact input modules, once a declaration has been made, interrupts are accepted until an interrupt declaration is cancelled.

● Interrupt Cancellation

Use an OFF INIT or RESET statement to cancel the interrupt declaration. An OFF INT statement cancels interrupt declarations on a terminal-by-terminal basis, while a RESET statement cancels them on a module-by-module basis.

The OFF INIT statement has the following format.

OFF $\underline{\text{INT}}$ $\underline{m}, \underline{n}$

m: Slot number in a numeric expression

n: Terminal number in a numeric expression

Example: OFF INT 102, 2

This statement cancels the interrupt declaration made for terminal 2 of a high-speed input module in slot 102.

The RESET statement has the following format.

RESET \underline{m}

m: Numeric expression

Example: RESET 4

This statement cancels all of the interrupt declarations made for the terminals of a high-speed input module in slot 4.

(c) Reading the Interrupt Status

You can read the interrupt status with a STATUS statement. The interrupt status is indicated by the value of an interrupt-handling auxiliary flag register. The auxiliary flag register changes from 0 to 1 at a rising edge of input (OFF-to-ON transition). This auxiliary flag register changes to 0 if read once by a STATUS statement and retains 0 until the next rising edge of input occurs. Note however, that any terminal for which acceptance of interrupts has been declared by an ON INT statement causes its auxiliary flag register to reset to 0 immediately after the occurrence of an interrupt. The register therefore always reads 0 whenever you read it using a STATUS statement.

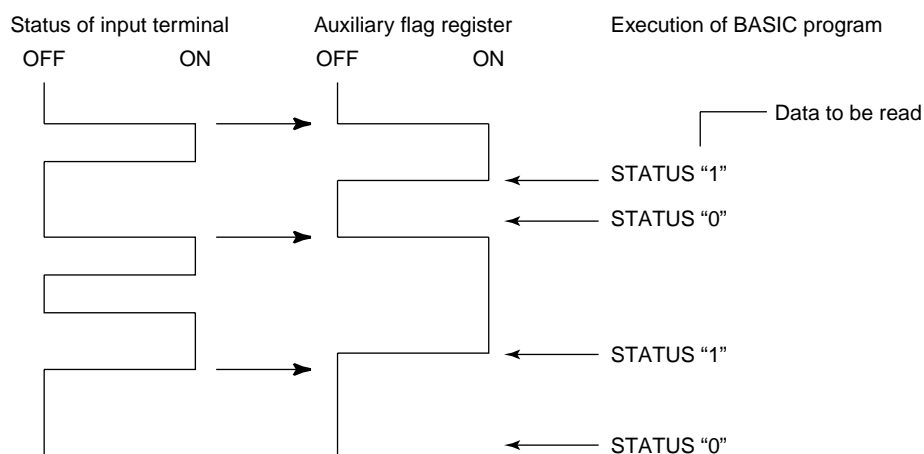
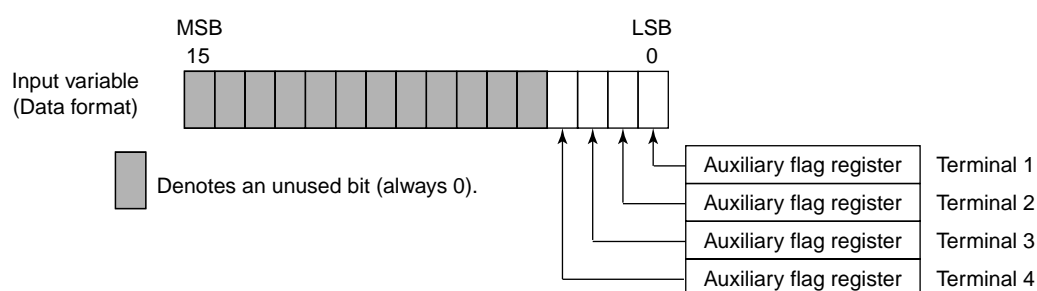
The STATUS statement has the following format.

STATUS m, 101;i

m: Slot number in a numeric expression

101: Register number in a numeric expression, which is always 101

i: Integer-type input variable



FB070419.EPS

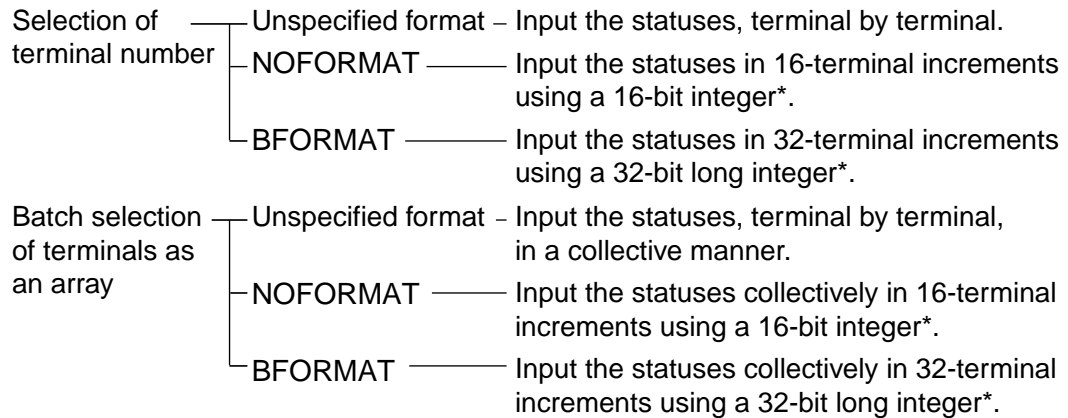
Figure B7.2 Flowchart Showing How Status Is Input to the Variable

Example: STATUS 304, 101;*i*

This statement reads the interrupt status of a module in slot 304 into the integer-type variable *i*.

B7.4.3 Contact Output Modules

A contact output module is used to output the ON/OFF status of contacts. Use an OUTPUT statement to output the contact status. Input 1 for the ON status of an output terminal and 0 for the OFF status of an output terminal. The status of each output terminal can be output by 1) selecting terminal numbers on a terminal-by-terminal basis, 2) making a batch selection of terminal numbers, or 3) making a batch selection of terminals using a 16-bit integer or a 32-bit long integer, as described below.



FB070420.EPS

* Applies only to those terminals that have been given permission by mask data for status output.

(a) Selection of Terminal Number

This method is only suitable for outputting data to a specified terminal. This method uses the following formats.

● Terminal-by-terminal Output

The following statement outputs data to the specified terminal only.

OUTPUT m, n;P

m: Slot number in a numeric expression

n: Terminal number in a numeric expression

P: Output data in a numeric expression

The variable P takes a value of either 1 or 0 to output either the ON or OFF state to output terminal n.

Example: OUTPUT 308, 1;1

This statement sets terminal 1 of a module in slot 308 to the ON state.

● 16-terminal Collective Input

The following statement applies a masking process to the data of 16 terminals, which are consecutive from the specified terminal, and outputs the data. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals.

OUTPUT m, n NOFORMAT;I["%", M]

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = 1, 17, 33 or 49

I: Output data (integer-type or numeric variable)

M: Mask data (integer-type or numeric variable)



CAUTION

Use an integer variable for I; do not use a long-integer variable. The functionality of an OUTPUT statement is not guaranteed if you use a long-integer simple variable.

The relationship between the variable that stores output data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of the variable. The status of each terminal is represented as either "bit-ON" (1) for the ON state or "bit-OFF" (0) for the OFF state.

	MSB (Terminal numbers) LSB															
If n = 1:	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
If n = 17:	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17
If n = 33:	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
If n = 49:	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49

FB070421.EPS

Example: OUTPUT 308, 17 NOFORMAT;\$A4C1, "%", \$94A1

This statement outputs data, as shown below, to terminals 17, 22, 24, 25, 26, 29 and 32 of a module in slot 308.

Output data	1	0	1	0	0	1	0	0	1	1	0	0	0	0	0	1
Mask data	1	0	0	1	0	0	1	1	1	0	1	0	0	0	0	1
	O			O			O	O	O		O					O
	N			F			F	F	N		F					N
Terminal numbers	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17

FB070422.EPS

● 32-terminal Collective Input

The following statement applies a masking process to the data of 32 terminals, which are consecutive from the specified terminal, and outputs data. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals.

OUTPUT \underline{m}, n \underline{B} FORMAT;L[, "%",LM]

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = 1 or 33

L: Output data (long-integer type or numeric variable)

LM: Mask data (long-integer type or numeric variable)



CAUTION

Use a long-integer variable for L and LM; do not use an integer variable. The functionality of an OUTPUT statement is not guaranteed if you use an integer variable.

The relationship between the variable that stores input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of the variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.

MSB
(Terminal numbers)
LSB

If n = 1:

32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	---	---	---	---	---	---	---	---	---

If n = 33:

64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

FB070423.EPS

(b) Batch Selection as an Array

This method is suited for outputting data to all terminals collectively or outputting the data to terminals excluding (by masking) a specific terminal or terminals. No terminal numbers should be specified. This method uses the following formats.

● All-terminal Collective Output

This method outputs the values of data (1 or 0) to all of the terminals and uses the following format.

OUTPUT m;P(*)

m: Slot number in a numeric expression

P (*): Output data for collectively specifying numeric array variables

Example: OPTION BASE 1
 DIM P(6) Terminals for collective output
 FOR I=1 TO 8
 P(I*2-1)=1: P(I*2)=0
 NEXT I
 OUTPUT 305;P(*)

	Status of terminal	Output data
Terminal 1	ON	1
Terminal 2	OFF	0
Terminal 3	ON	1
⋮	⋮	⋮
Terminal 16	OFF	0

FB070424.EPS

● Collective Output in 16-terminal Increments Using a 16-bit Integer

This method applies a masking process to the output data of all output terminals, 16 bits of the data at a time, in 16-terminal increments. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals. Use either of the following formats.

OUTPUT \underline{m} $\underline{\text{NOFORMAT}}$; I_1, I_2, I_3, I_4 [, "%", M_1, M_2, M_3, M_4]

OUTPUT \underline{m} $\underline{\text{NOFORMAT}}$; $I(*)$ [, "%", $M(*)$]

m : Slot number in a numeric expression

I_n : Output variable list for integer-type simple variables or numeric values

M_n : Mask data list for integer-type simple variables or numeric values

$I(*)$: Output variable list for collectively specifying integer-type, array variables

$M(*)$: Mask data for collectively specifying integer-type, array variables



CAUTION

Use integer-type simple variables and integer-type array variables for I_n , M_n , $I(*)$ and $M(*)$; do not use long-integer type simple variables and long-integer type array variables. The functionality of an OUTPUT statement is not guaranteed if you use long-integer type simple variables or long-integer type array variables.

The relationship between the variables that store input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of each variable. The status of each terminal is represented as either "bit-ON" (1) for the ON state or "bit-OFF" (0) for the OFF state.

• Example of Integer-type Simple Variables

(Output data)													(Mask data)				
MSB													LSB				
	(Terminal numbers)																
I_1	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	M_1
I_2	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	M_2
I_3	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	M_3
I_4	64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	M_4

FB070425.EPS

I_2 and M_2 are only valid for a 32- or 64-input module, while I_3 , I_4 , M_3 and M_4 are only valid for a 64-input module.

• Example of Integer-type Array Variables (for an OPTION BASE 1 Statement)

(Output data)	MSB	(Terminal numbers)																LSB	(Mask data)
I (1)		16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1		M (1)
I (2)		32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17		M (2)
I (3)		48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33		M (3)
I (4)		64	63	62	61	60	59	58	57	56	55	54	53	52	51	50	49		M (4)

FB070426.EPS

I (2) and M (2) are only valid for a 32- or 64-input module, while I (3), I (4), M (3) and M (4) are only valid for a 64-input module.

● Collective Output in 32-terminal Increments Using a 32-bit Long Integer

The following statements apply a masking process to the output data of all output terminals, 32 bits of the data at a time, in 32-terminal increments. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals. Use either of the following formats.

OUTPUT \underline{m} $\underline{\text{BFORMAT}}$; L_1 [, L_2] [, “%”, LM_1 , LM_2]

OUTPUT \underline{m} $\underline{\text{BFORMAT}}$; $L(*)$ [, “%”, $LM(*)$]

m : Slot number in a numeric expression

Ln : Output data list for long-integer type simple variables or numeric values

LMn : Mask data for long-integer type simple variables or numeric values

$L(*)$: Output data list for collectively specifying long-integer type array variables

$LM(*)$: Mask data for collectively specifying long-integer type array variables

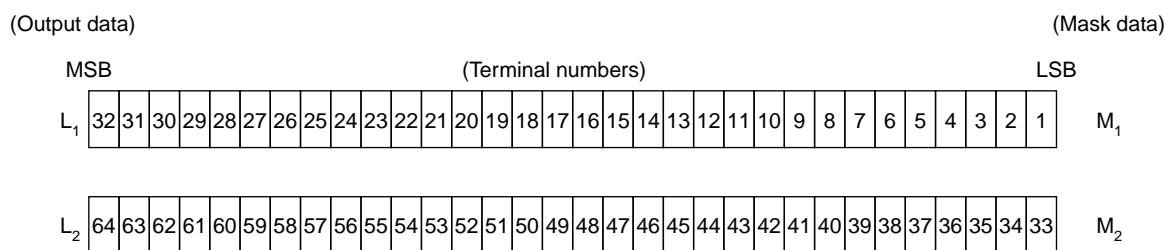


CAUTION

Use long-integer type simple variables and long-integer type array variables for Ln , LMn , and $L(*)$ and $LM(*)$; do not use integer-type simple variables and integer-type array variables. The functionality of an OUTPUT statement is not guaranteed if you use integer-type simple variables or integer-type array variables.

The relationship between the variables that store input data and the terminals is as follows. Lower terminal numbers correspond to the lower-order bits of each variable. The status of each terminal is represented as either “bit-ON” (1) for the ON state or “bit-OFF” (0) for the OFF state.

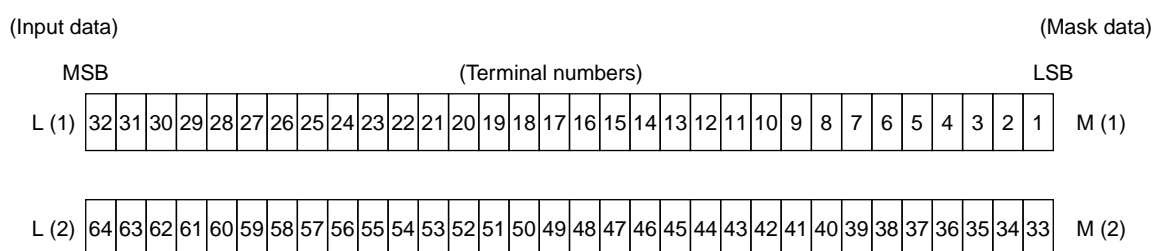
• Example of Long-integer Type Simple Variables



FB070427.EPS

L2 and M2 are only valid for a 64-input module.

• Example of Long-integer Type Array Variables (for an OPTION BASE 1 Statement)

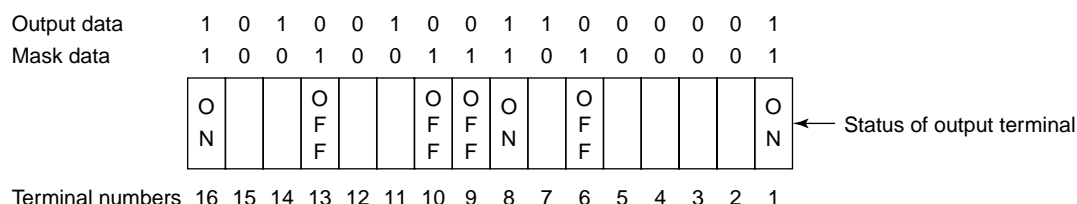


FB070428.EPS

L (2) and M (2) are only valid for a 64-input module.

Example: OUTPUT 207 NOFORMAT;\$A4C1, “%”, \$93A1

This statement outputs data, as shown below, to terminals 1, 6, 8, 9, 10, 13 and 16 of a module in slot 207.



FB070429.EPS

B7.4.4 Defining the Operating Mode of a Contact Output Module

For contact output modules except for 64-point models, it is possible to define whether the modules should retain their current states of outputs or reset their outputs in case of a failure in the master CPU module. A failure in the master CPU module here refers to a serious failure level (CPU failure, memory failure, power failure, etc.) in which the RDY green LED indicator lamp turns off.

(a) Setting an Operating Mode against Master Module Failure

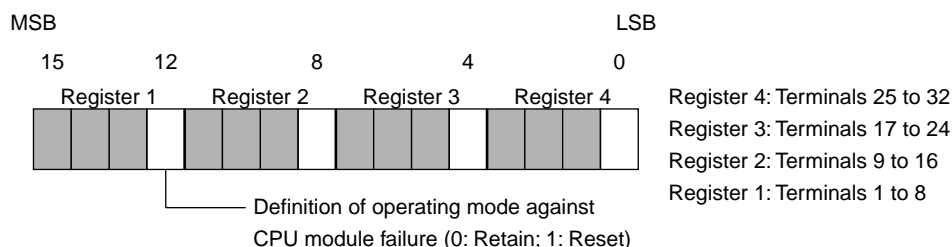
Using a CONTROL statement, define whether the contact output modules should retain their current states of outputs or reset their outputs in case of a failure in the master CPU module. Set the definition in the control registers of each contact output module in 8-terminal increments. Terminal-by-terminal setting is not possible. The CONTROL statement has the following format.

CONTROL $\underline{\text{m}}$, 1;I

m: Slot number in a numeric expression

I: Data (integer or integer-type variable)

The control registers are configured as shown below. Upon system startup, all of the control registers are set to 0.



Denotes an unused bit (always 0).

FB070430.EPS

Example: CONTROL 206, 1;\$0010

This statement causes the contact output module in slot 206 to reset its terminals 9 to 16 and retain all the remaining terminals at their current states in case of a failure in the master CPU module.

(b) Reference to Parameters

Using a STATUS statement, you can refer to parameters configured by a CONTROL statement. The STATUS statement has the following format.

STATUS m, 1;I

m: Slot number in a numeric expression

I: Input variable (integer-type)

Example: STATUS 204,1;I

This statement reads the data of control registers in a module in slot 204 into the variable I.

(c) Initializing the Parameters

Use a RESET statement to initialize (to a value of 0) all data items set with a CONTROL statement. The RESET statement has the following format.

RESET m

m: Slot number in a numeric expression

Example: RESET 308

This statement initializes all the values of control registers in the module in slot 308 to 0.

B7.4.5 Contact I/O Modules

Access to an F3WD64-□N contact I/O module that has both inputs and outputs is basically the same as access to contact input modules in terms of input operation and access to contact output modules in terms of output operation. In the case of contact I/O modules however, it is not possible to use the interrupt function. Nor is it possible to define whether the contact I/O modules should retain their current states of outputs or reset their outputs in case of a failure in the master CPU module.

(a) Access to Input Block

Using an ENTER statement, input the states of contacts. Access to the input block is basically the same as access to contact input modules. However, terminal numbers to be specified differ from those of a contact input module. For details on the batch selection as an array, among the input methods described below, refer to the explanation of 32-point contact input modules.

● Terminal-by-terminal Input

This input method inputs the state of a specified terminal only.

ENTER m, n;P

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = a value from 1 to 32

P: Input variable (numeric)

● 16-terminal Collective Input

This input method inputs the states of 16 terminals collectively into the variable, beginning with the specified terminal.

ENTER m, n NOFORMAT;I

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = 1 or 17

I: Input variable (integer-type)

● 32-terminal Collective Input

This input method inputs the states of 32 terminals collectively into the variable, beginning with the specified terminal.

ENTER m, n BFORMAT;L

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n is always 1

L: Input variable (long-integer type)

(b) Access to Output Block

Using an OUTPUT statement, output the states of contacts. Output 1 for the ON status of a contact output and 0 for the OFF status of a contact output.

Access to the output block is basically the same as access to contact output modules.

Terminal numbers to be specified differ from those of a contact output module, however.

For details on the batch selection as an array, among the input methods described below, refer to the explanation of 32-point contact output modules.

● Terminal-by-terminal Input

This input method outputs the status to a specified terminal only.

OUTPUT m, n;P

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = a value from 33 to 64

P: Output data in a numeric expression

The variable P takes a value of either 1 or 0 to output either the ON or OFF state to output terminal n.

● 16-terminal Collective Input

The following statement applies a masking process to the data of 16 terminals, which are consecutive from the specified terminal, and then outputs the data. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals.

OUTPUT m, n NOFORMAT;I[, "%", M]

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n = 33 or 49

I: Output data (integer-type variable or numeric value)

M: Mask data (integer-type variable or numeric value)

● 32-terminal Collective Input

The following statement applies a masking process to the data of 32 terminals, which are consecutive from the specified terminal, and outputs data. It outputs the data only to those terminals whose bits of mask data are 1. If you omit the mask data, the statement outputs data to all of the terminals.

OUTPUT m, n BFORMAT;L[, "%", M]

m: Slot number in a numeric expression

n: Terminal number in a numeric expression, where n is always 33

L: Output data (long-integer type variable or numeric value)

LM: Mask data (long-integer type variable or numeric value)

B7.5

82-xx error codes that may appear during access to a contact I/O module have the meanings summarized in the following table.

Error Code	Detailed Error Code (Hexadecimal)	Meaning	Possible Cause
082	\$82	Wrong selection of function	<ul style="list-style-type: none"> The module has executed an unsupported statement (e.g., an OUTPUT statement made to an input module). The slot number specified is one allocated to a different contact I/O module.
	\$91	Parameter error	<ul style="list-style-type: none"> The device number is out of the specified range.
	\$92	Setpoint data error	<ul style="list-style-type: none"> There is an error in the setpoint data (e.g., data type error). A value other than 0 and 1 has been specified during output by specifying a terminal number.
	\$94	Wrong selection of module	<ul style="list-style-type: none"> The module name included in the ASSIGN statement is wrong. No ASSIGN statement has been executed yet.
	\$D1	Device error	<ul style="list-style-type: none"> The module is faulty.
	\$E1	Device not ready	<ul style="list-style-type: none"> The module is not installed yet. The slot number is wrong. The module is faulty.

TB070501.EPS

B7.6 Contact Input/Contact Output Modules- Programming Exercise

B7.6.1 Contact Input Modules

```

10  ! Contact input modules-Programming exercise
10  DEFINT A-Z
10  OPTION BASE 1
10  DIM P(32)
10  SG4
10  ! Declare use of modules
10  ASSIGN XD32=SC
10  ! Initialize parameters
10  RESET SC
10  ! Set interrupt conditions
20  CONTROL SC,1;$1111
20  ! Read interrupt conditions
20  STATUS SC,1;STUS
20  PRINT HEX$(STUS)
20  ! Declare acceptance of interrupts
20  ON INT SC,8 GOSUB MSG@
20  ! Input data by specifying a terminal number
20  ENTER SC,8;P(8)
20  PU "8A,DZ";"P(NO.8)=",P(8)
20  ! Input data by collectively specifying terminal numbers
30  ENTER SC;P(*)
30  FOR I=1 TO 32
30  PU "5A,DZ,2A,DZ";"P(NO.",I,")=",P(I)
30  NEXT I
30  ! Input data by collectively specifying terminal numbers in 16-terminal increments
30  ENTER SC NOFORMAT;A
30  FOR I=1 TO 16
30  PU "5A,DZ,2A,DZ";"P(NO.",I,")=",BIT(A,I-1)
30  NEXT I
30  ! Read the interrupt status (No. 16)
40  STATUS SC,101;ST
40  PRINT BIT (ST,15)
40  STOP
40  MSG@
40  PRINT "Interrupt from terminal 8"
40  RETURN
40  END

```

B7.6.2 Contact Output Modules

```

10  ! Contact output modules-Programming exercise
10  DEFINT A-Z
10  OPTION BASE 1
10  DIM PON(32),POF(32),PIN(2),MASK(2)
10  SC=3
10  ! Declare use of modules
10  ASSIGN YD32=SC
10  ! Initialize parameters
10  RESET SC
10  ! Define operating mode against master CPU failure
20  CONTROL SC,1;$1111
20  ! Read control registers
20  STATUS SC,1;STUS
20  PRINT HEX$(STUS)
20  ! Output data by specifying a terminal number
20  FOR I=1 TO 32
20      OUTPUT SC,I;1
20      PON(I)=1 :POF(I)=0
20      WAIT 100
20  NEXT I
30  ! Output data by collectively specifying terminal numbers
30  OUTPUT SC;POF(*)
30  ! Output data by collectively specifying terminal numbers in 16-terminal increments
30  FOR I=1 TO 2
30      MASK(I)=$FFFF
30      FOR J=1 TO 10
30          PIN(I)=RND($8000)
30          OUTPUT SC NOFORMAT;PIN(1),PIN(2),"%",MASK(1),MASK(2)
30          WAIT 500
30      NEXT J
40      PIN(1)=0
40  NEXT I
40  ! Output data by collectively specifying terminal numbers
40  OUTPUT SC;PON(*)
40  WAIT 1000
40  OUTPUT SC;POF(*)
40  END

```


B8. Libraries

This chapter provides general information on the libraries.

B8.1 What Is a Library?

A library is a package of software based on machine language supplied by Yokogawa Electric Corporation. It is used to call subroutines (see Section B8.3, "Program Flow," later in this part, "Description of YM-BASIC/FA," for more details). Libraries are supplied either as a standard accessory for the FA-M3 multi-controller or as an optional software package.

B8.2 Incorporating Libraries into a User Program

■ Libraries Provided as an Optional Software Package

These libraries are supplied in the form of files (AS type). To be able to use them in a user program, you must go through the given procedures, such as incorporating them into the user program.

A library is built into (linked with) a user program when the program is being created. To link the library with the program, use a LINKLIB command. Each library shares the same name with its file and is supplied as one file. A library is executed in a user program area. Save a program containing libraries on disk in an intermediate-language format. If you attempt to save the program in a source-code format, part of the program consisting of the libraries will be excluded from the saving.

■ Libraries That Come Standard with the FA-M3 Multi-controller

The FA-M3 multi-controller comes standard with the following libraries.

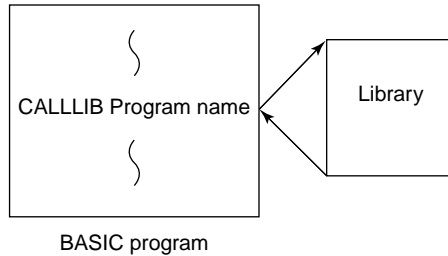
- INICOMM3
- IFPCNV

Both of these libraries come already built in the YM-BASIC/FA programming language. You therefore do not have to incorporate them into your user program using a LINKLIB command. For more details on these libraries, see Section C3 in Part C, "Syntax of YM-BASIC/FA," later in this manual.

B8.3 Program Flow

■ Calling Libraries

Use a CALLLIB statement to call libraries from a BASIC program. Pass parameters (arguments) as necessary.



FB080301.EPS

Since the character string "CALLLIB" can be omitted, you can write the statement just like a BASIC statement or an arithmetic function.

■ Interrupts

Any interrupt branch that occurs during branching from a BASIC program to a library is placed in a wait state until the CPU returns to the BASIC program.

C1. Syntax Usage

C1.1 Positioning the Part “Syntax”

This part is configured as shown below.

Chapter 1. PART II Syntax Usage

Positioning of the part “Syntax” and terms commonly used throughout this part are described.

Chapter 2. YM-BASIC/FA Function Lists

Lists and functions of the following items that can be used as standard in YM-BASIC/FA are briefly described :

- Command and subcommand

- Statement

- Function

- Library

Chapter 3. Syntax

The syntax of the following items that can be used as standard in YM-BASIC/FA are described in detail.

- Command and subcommand

- Statement

- Function

- Library

The order of description of the syntax for these items in this manual is in alphabetic order regardless of statements and functions.

For syntax of commands and subcommands, see manuals shown in “2. YM-BASIC/FA Function Lists”.

Chapter 4. Error Codes


This chapter describes error codes and detailed error codes that can occur during execution of YM-BASIC/FA, and their meaning.

C1.2 Terms Used in This Part

■ Commands and Statements

BASIC instructions are divided into several types.

- **Commands and Subcommands**

These are instructions which are executed by direct specification to YM-BASIC/FA from the keyboard and are entered without line number. At the end of a command or a subcommand, enter the return key .

Commands can be used in a command entry panel (with prompt of “:”) and subcommands, in editor startup (with prompt of “>”).

- **Statements**

Instructions which are executed by describing them in programs are called “statements”. Most statements can be used as commands but if a variable is used, an undefined variable error occurs.

- **Functions**

These are similar to statements but those that have each value (answer) are called functions.

- **Libraries**

A library is a program described by machine language provided by YOKOGAWA. Libraries realize the contents which cannot be processed by YM-BASIC/FA or that require longer processing times.

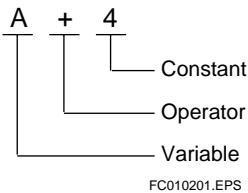


CAUTION

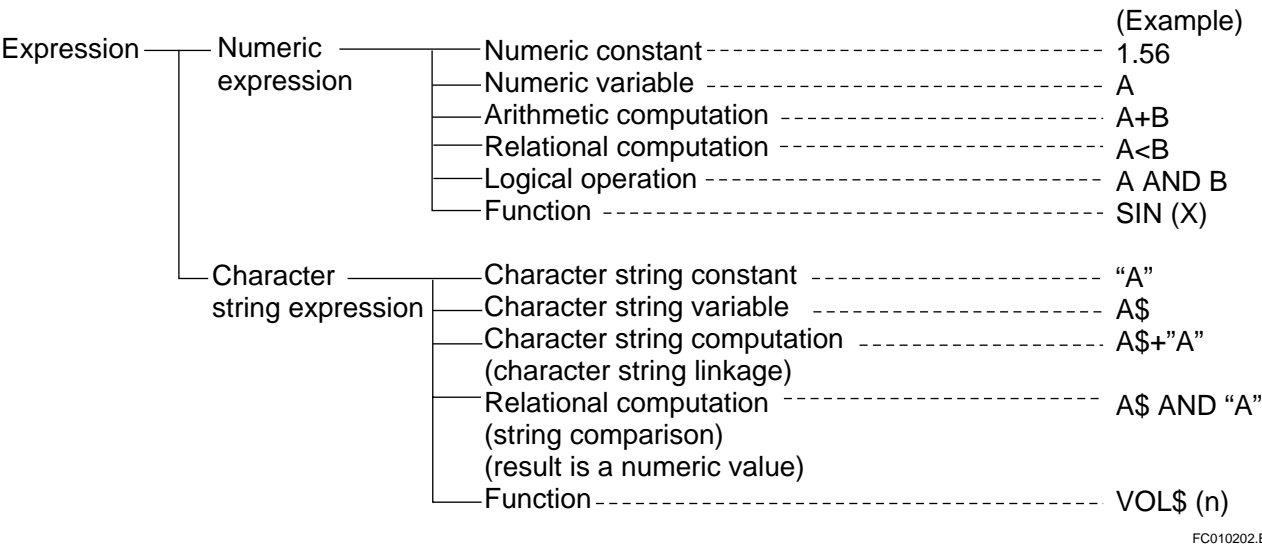
In BASIC Programming Tool M3 for Windows, the commands, subcommands, and key operations [ESC], [CTRL]+[S], [CTRL]+[P], [CTRL]+[C], and the like, are operated from the menu bar, toolbar or edit window.

■ Numeric Expressions and Character String Expressions

An expression is a representation in which constants and variables are connected with operators.



As an expression result is a numeric value or a character string, a simple character or numeric value or a variable only is also considered as an expression. Numeric expressions and character string expressions include the following:



Use parentheses for changing the operation execution order.
Computation in parentheses is first executed. Multiple parentheses can be nested.
For variables, constants, and computation, see PART I "REFERENCE".

C2. List of YM-BASIC/FA Functions

C2.1 Commands and Subcommands

Commands and subcommands that YM-BASIC/FA supports are listed below. Use commands or subcommands when creating (entering) YM-BASIC/FA programs. YM-BASIC/FA programs can be created (entered) in a personal computer or a YEWMAC line computer.

Since commands and subcommands usable in YM-BASIC/FA slightly differ in a personal computer and a YEWMAC line computer, lists for both types of computer are given separately.



CAUTION

In BASIC Programming Tool M3 for Windows, the commands, subcommands, and key operations [ESC], [CTRL]+[S], [CTRL]+[P], [CTRL]+[C], and the like, are operated from the menu bar, toolbar or edit window.

■ Commands and Subcommands for Personal Computer

The stand-alone personal computer can use commands and subcommands other than those marked △

● Commands and Subcommands

Command	Function	Manual to be referred to
General commands		
FREE	Display of free area size in the program area (user area except for common area) (for both FA500 and personal computer)	C3-30
RENUM	Line number replacement	C3-68
DEL	Deletion of subprogram or library	C3-19
SAVE	Saving a program in the user area in an auxiliary memory	C3-72
LOAD	Loading a program stored in an auxiliary memory to the user area	C3-50
MERGE	Merging a program stored in an auxiliary memory into a program in the user area	C3-52
APPEND	Adding a program stored in an auxiliary memory to the current program block in the user area	C3-4
LINKLIB	Loading a library to the user area and incorporating the library in a program in the user area	C3-48
LIST	Outputting a program in the user area to a specified output device (CRT or printer)	C3-49
ERLIST	Outputting a program list (to CRT or printer) in a line where an error is occurring	C3-27
BYE	Terminating the debug mode	C3-9
△ CONT	Restarting a program whose execution has once been suspended	C3-14
△ RUN	Executing a program in the user area	C3-71
NEW	Initializing (erasing) a program in the user area	C3-53
PROG	Specifying a program block which becomes an object of command processing	C3-66
AUTO	Automatically generating the line number	C3-6
STEP	Executing a line of program	C3-80
EDIT	Starting the line editor	C3-22
LCOPY	Copying on a line basis	C3-46
△ SETMD RUN	Designation/release of BYE & RUN mode	C3-77
△ SETMD RES	ON/OFF of designating residence in a program	C3-76
△ TRACE	Tracing branch-generated locations	C3-84
△ SCRATCH	Resetting trace of branch-generated locations	C3-74
△ TRACEP	Pause during execution	C3-85
△ SCRATCHP	Resetting pause during execution	C3-74
△ TRACEV	Tracing variabl	C3-86
△ SCRATCHV	Resetting trace of variables	C3-74

TC020101.EPS

△ : If the FA500 is not connected, this command cannot be executed.

Command	Function	Manual to be referred to
Subcommands (used in EDIT)		
QUIT (Q)	Termination of the editor	C3-66
FIND (F)	Searching a character string	C3-28
CHG (C)	Replacing character strings	C3-11
APPEND (A)	Linking a specified line	C3-4
DEL	Deleting a specified line	C3-19
RENUM	Renumbering a line	C3-68
LCOPY	Copying on a line basis	C3-46
LIST (L)	Displaying a program list	C3-49
EDIT	Displaying a specified line	C3-22
ERLIST	Displaying an error statement and error details	C3-27

TC020102.EPS

● MS-DOS Commands

Command	Function	Manual to be referred to
MS-DOS* Commands		
! CHDIR	Changing directory and displaying the current directory	Manual attached to a personal computer
! CLS	Clearing the display panel	
! COPY	Copying a specified file	
! DATE	Date display and setting (clock in a personal computer)	
! DEL	Deleting a specified file	
! DIR	Displaying directory information	
! MKDIR	Creating a directory	
! PATH	Displaying a path for searching the command	
! REN	Changing a file name	
! RMDIR	Deleting a directory	
! TIME	Date displaying and setting (clock in a personal computer)	
! TYPE	Displaying the contents of a specified file	
! VOL	Displaying a volume label	

TC020103.EPS

For commands with !, MS-DOS* command is executed as part of these commands.

* : MS-DOS is a trademark of Microsoft Corp.



CAUTION

MS-DOS commands other than those listed above cannot be used in YM-BASIC/FA. Furthermore, while some commands have abbreviations in MS-DOS (e.g., CHDIR → CD), YM-BASIC/FA does not support them.



CAUTION

In BASIC Programming Tool M3 for Windows, the commands, subcommands, and key operations [ESC], [CTRL]+[S], [CTRL]+[P], [CTRL]+[C], and the like, are operated from the menu bar, toolbar, or edit window.

C2.2 Statements

A list of statements supported by YM-BASIC/FA is shown below.

Statement	Function	Page on which the statement is explained
General Statements		
REM	Annotation of program	C3-67
DEFINT	Integer type declaration	C3-18
DEFLNG	Long integer type declaration	C3-18
DEFSNG	Single-precision type declaration	C3-18
DEFDBL	Double-precision type declaration	C3-18
OPTION BASE	Specifying array subscript starting number (lower limit)	C3-61
DIM	Specifying an array	C3-20
ALLOCATE	Specifying an array whose size is variable	C3-3
DEF FN	Specifying user functions	C3-17
LET	Assigning to a variable	C3-47
MOVE	Moving the array data	C3-53
SWAP	Data exchange	C3-82
READ	Data reading	C3-67
DATA	Data setting	C3-16
RESTORE	Setting a pointer for reading data	C3-69
GOTO	Unconditional branching	C3-31
GOSUB	Branch to a subroutine	C3-31
RETURN	Return from a subroutine	C3-69
ON GOTO	Branching based on a calculated result	C3-61
ON GOSUB	Branching to a subroutine based on a calculated result	C3-61
IF ... THEN ... ELSE ... ENDIF	Execution of statement by the conditions	C3-34
WHILE ... END WHILE	Iterative execution during the time when conditions are met	C3-90
FOR ... NEXT	Iterative execution	C3-29
STOP	Program stopping	C3-80
PAUSE	Temporary stopping of a program	C3-63
END	Program termination	C3-24

TC020201.EPS

Statement	Function	Page on which the statement is explained
General Statements		
RANDOMIZE	Random number generation	C3-66
CALLLIB	Branching to a library	C3-11
CALL	Branching to a subprogram	C3-10
COM	Common variable declaration	C3-13
SUBCOM	Specifying the starting position of the common area	C3-81
RECOM	Specifying a common variable position	C3-67
INIT COM	Initializing the common area	C3-42
Subprogram Statements		
SUB	Declaring the beginning of a subprogram	C3-81
SUBEXIT	Return from a subprogram	C3-82
SUBEXIT RETRY	Return from a subprogram	C3-82
SUBEND	Declaring subprogram termination	C3-82
Fundamental I/O Statements		
<input type="checkbox"/> PRINT	Print output	C3-64
<input type="checkbox"/> PRINT USING	Print output with format	C3-65
IMAGE	Specifying a format	C3-37
<input type="checkbox"/> DISP	Output to CRT	C3-21
<input type="checkbox"/> DISP USING	Output to CRT with a format	C3-21
<input type="checkbox"/>	Real-time Statements	
WAIT	Waiting for program execution	C3-89
ON TIME	Branching at a timing	C3-58
OFF TIME	Canceling branching at a timing	C3-58
ON TIMER	Starting a timer	C3-60
OFF TIMER	Stopping a timer	C3-60
DISABLE	Disabling interrupt	C3-21
ENABLE	Enables interrupt from the disabling state	C3-24

TC020202.EPS

☐ : Can be used in debugging.

Statement	Function	Page on which the statement is explained
I/O Module Support Statements		
ASSIGN	I/O module configuration definition	C3-5
ENTER	Entry from I/O instruments	C3-26
OUTPUT	Output to I/O instruments	C3-62
TRANSFER	Starting I/O using an I/O buffer	C3-87
ON EOT	Branching at the end of transfer	C3-54
OFF EOT	Resetting branching at the end of transfer	C3-54
ENABLE INTR	Masking the cause of I/O interrupt	C3-24
ON INT	Branching by I/O interrupt	C3-56
OFF INT	Resetting branching by I/O interrupt	C3-56
SET TIMEOUT	Branching at the time-out for I/O operation	C3-76
ON TIMEOUT	Branching at the I/O time-out	C3-59
OFF TIMEOUT	Resetting branching at the I/O time-out	C3-59
RESET	Resetting I/O module	C3-68
STATUS	Reading a status register	C3-79
CONTROL	Writing to a control register	C3-15
HALT	Canceling the transfer action	C3-31
SET STATUS	Specifying an I/O status information variable	C3-75
RESET STATUS	Resetting an I/O status information variable	C3-69
Statements Exclusively Used for Sequences		
COM #S ~	Declaration of sequence element (common register) common variable	C3-13
ON SEQEV	Declaration accepting interrupt from a sequence	C3-57
OFF SEQEV	Resetting declaration accepting interrupt from a sequence	C3-57
SEQACTV	Start/stop of a sequence program	C3-75
Exception Processing and Debugging Statements		
DEFAULT ON	Declaration of implicit processing when the computed result is abnormal	C3-18
DEFAULT OFF	Resetting of implicit processing when the computed result is abnormal	C3-18
ON ERROR	Branching when BASIC error occurs	C3-55
OFF ERROR	Canceling branching when BASIC error occurs	C3-55
RETURN RETRY	Return from a subroutine and re-execution	C3-69
<input type="checkbox"/> TRACE	Tracing the branch-generated location	C3-84
<input type="checkbox"/> SCRATCH	Resetting tracing of the branch-generated location	C3-74
<input type="checkbox"/> TRACEP	Temporary stop during execution	C3-85
<input type="checkbox"/> SCRATCHP	Resetting temporary stop during execution	C3-74
<input type="checkbox"/> TRACEV	Variable tracing	C3-86
<input type="checkbox"/> SCRATCHV	Resetting variable tracing	C3-74

TC020203.EPS

☐ : Can be used in debugging.

C2.3 Functions

A list of functions supported by YM-BASIC/FA is indicated.

Function	Function	Page on which the statement is explained
Arithmetic Functions		
SIN (x)	Returns the sine of x.	C3-78
COS (x)	Returns the cosine of x.	C3-15
TAN (x)	Returns the tangent of x.	C3-83
ATN (x)	Returns the arc tangent of x.	C3-6
EXP (x)	Returns the value of the exponential function for natural number x.	C3-83
LOG (x)	Returns the natural logarithm of x.	C3-50
SQR (x)	Returns the square root of x.	C3-78
ABS (x)	Returns the absolute value of x.	C3-3
SGN (x)	Returns the sign of x.	C3-77
INT (x)	Returns the maximum integer not exceeding x.	C3-42
RND (x)	Returns random numbers in the range of $0 < \text{RND}(x) < x$.	C3-70
DIV (A, B)	Determines the quotient of A divided by B.	C3-22
MOD (A, B)	Determines the remainder of A divided by B.	C3-52
Bit Handling Functions		
BINAND (m, n)	Returns the AND operation of m and n bit by bit.	C3-7
LBINAND (m, n)	Returns the AND operation of long integers m and n bit by bit.	C3-44
BINOR (m, n)	Returns the OR operation of m and n bit by bit.	C3-7
LBINOR (m, n)	Returns the OR operation of long integers m and n bit by bit.	C3-44
BINXOR (m, n)	Returns the exclusive OR operation of m and n bit by bit.	C3-8
LBINXOR (m, n)	Returns the exclusive OR operation of long integers m and n bit by bit.	C3-45
BINNOT (m)	Returns the one's complement of m.	C3-7
LBINNOT (m)	Returns the one's complement of long integer m.	C3-44
SHIFT (m, n)	Shifts m by n bits.	C3-78
LSHIFT (m, n)	Shifts a long integer m by n bits.	C3-51
ROTATE (m, n)	Rotates (shifts) m by n bits.	C3-71
LROTATE (m, n)	Rotates (shifts) a long integer m by n bits.	C3-51
LASTBIT	Returns the LASTBIT value.	C3-43
BIT (m, n)	Returns the bit value of m at the location designated by n.	C3-8
LBIT (m, n)	Returns the bit value of a long integer m at the location designated by n.	C3-45

TC020301.EPS

Function	Function	Page on which the statement is explained
Character Handling Functions		
MID \$ (c, m)	Returns a character string from the m-th character from the left end of a character string c.	C3-52
MID \$ (c, m, n)	Returns a character string composed of n characters extracted from the m-th character from the left end of a character string c.	C3-52
LEFT \$ (c, m)	Returns a character string composed of m characters extracted from the left end of a character string c.	C3-46
RIGHT \$ (c, m)	Returns a character string composed of m characters extracted from the right end of a character string c.	C3-70
CHR \$ (n)	Creates a character corresponding to a code value n.	C3-12
STR \$ (x)	Converts a numeric value x to a character string.	C3-81
VAL (c)	Converts a character string c to a numeric value.	C3-88
ASC (c)	Returns the code value of the first character of a character string c.	C3-5
LEN (c)	Returns the number of characters in a character string c.	C3-46
HLEN (c)	Returns the number of standard characters in a character string c.	C3-33
BLEN (c)	Returns the number of characters in bytes in a character string c.	C3-8
INSTR (c, m)	Searches for the character string represented by m in a character string c and determines its starting position.	C3-42
HMID \$ (c, m, n)	MID\$ function on the standard character basis.	C3-33
HLEFT \$ (c, n)	LEFT\$ function on the standard character basis	C3-32
HRIGHT \$ (c, m)	RIGHT\$ function on the standard character basis	C3-33
HINSTR (c, m)	INSTR function on the standard character basis	C3-32
Other Functions		
PI	Returns the circular constant.	C3-63
ERRL	Returns the latest error line number.	C3-27
ERRC	Returns the error code of the latest error.	C3-27
ERRCE	Returns a detail error code in I/O access error.	C3-27
TIMEMS	Returns an elapsed time starting at 00:00 in ms.	C3-83
RNPAR	Returns startup parameters when a program is activated.	C3-70
SPC (x)	Returns x spaces.	C3-78
IOSIZE	Returns the number of transferred bytes in the latest I/O operation.	C3-43
HEX \$ (x)	Returns the hexadecimal notation of x.	C3-32
LHEX\$(x)	Returns the hexadecimal notation of x in the range of long integer.	C3-47
BCD (x)	Returns the binary coded decimal notation of x.	C3-6
LBCD(x)	Returns the binary coded decimal notation of x in the range of long integer.	C3-43
TIME \$	Obtains the time of the day.	C3-83
DATE \$	Obtains the date.	C3-16
NAM (c)	Obtains the value of a simple variable represented by c.	C3-53
ARNAM (c, m)	Obtains the value of m elements of a array represented by c.	C3-5
FREE	Returns the size of free program area.	C3-30

TC020302.EPS

C2.4 Libraries

A list of libraries supported by YM-BASIC/FA is as follows:

● Standard libraries for FA-M3

Library	Function	Reference page in this text
INCOMM3	Initializes (0 clear) the shared register area in the CPU.	C3-42
IFPCNV	Converts IEEE floating-point format and YM-BASIC/FA internal representation.	C3-35

TC020401.EPS

C3. Syntax

This chapter describes the details of syntax for statements and functions supported by YM-BASIC/FA.

■ Commands and subcommands

These are instructions executed by direct specification to YM-BASIC/FA from the keyboard and are entered without line number. At the end of a command or a subcommand, press the Enter/Return key. Commands can be used in a command entry panel (with prompt of "BSC:" or "bsc") in debug mode and subcommands in editor startup (with prompt ">").

■ Statements

Statements and functions are normally used by being described in programs.

Most statements can also be used as commands as immediately executable statements.

However, statements processed using more than one line such as FOR-NEXT, WHILE-END WHILE, or GOSUB-RETURN, and the branching statements such as ON-cannot be used as commands. In addition, using a variable in a command causes an error regarding it as an undefined variable.

Most statements and all functions can be executed either in real mode or in debug mode of execution mode. Some statements have meanings only in debug mode. Such statements have no meaning but cause no error even if they are executed in real mode.

■ Functions

Functions are always used to on the right side of the assignment expression to perform calculations on given numeric values and return the result of the calculation. However, functions — ARNAM and NAM — can be used on the left side. Numeric expressions and character string expressions can be used as function arguments. YM-BASIC/FA built-in functions and user-defined functions are used. For user-defined functions, consult the DEF FN statement.

■ Libraries

Libraries are groups of machine-language subprograms offered by YOKOGAWA. Libraries are used to perform operations which in YM-BASIC/FA would require excessive processing time, or not be possible at all.

For the use of libraries, refer to item C8 of YM-BASIC/FA "REFERENCE."



CAUTION

In BASIC Programming Tool M3 for Windows, the commands, subcommands, and key operations [ESC], [CTRL]+[S], [CTRL]+[P], [CTRL]+[C], and the like, are operated from the menu bar, toolbar or edit window.

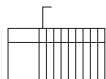
Symbols used in this manual:

Instructions given herein show commands, subcommands, statements, functions, and libraries.



: Shows that instructions can be used in F3BP20, F3BP30.

Instructions can be used when either the controller is connected to or disconnected from a personal computer.



: Shows that instructions can be used in either F3BP20.

Instructions can be used only when the controller is connected to a personal computer.

- Function:** The functions of commands, subcommands, statements, functions, or libraries are briefly described.
- Format:** How to enter commands, subcommands, statements, functions, or libraries are described.
1. Enter items represented by upper-case alphabetical letters, #, \$, and parentheses () as they are. Lower-case alphabetical letters cannot be used.
 2. For `___`, enter a space. This must not be omitted.
 3. (For Functions)

Enter a numeric expression or a character string expression for lower-case alphabetical letters. Numeric expressions include numeric constants and numeric variables. Character string expressions include character string constants and character string variables. For details of numeric and character string expressions, see PART I REFERENCE "2.8 Expression and Computation".

If a character string is to be entered, the character string to be entered must be enclosed with quotation marks (" "). The range and contents that can be described differ depending on the function.

(For commands, subcommands, statements, or libraries)

For items indicated with lower-case alphabetical letters or `Sc`, enter numeric constants or numeric variables or character string constants or character string variables.

If a character string is to be entered, the character string to be entered must be enclosed with quotation marks (" "). The range and contents that can be described differ depending on the function.
 4. Items enclosed with brackets [], since they are the optional items, can be omitted.

Brackets "[", "]" are not necessary for actual entry.

Example : `AUTO ___ [starting line number [, increment]]`

[1] The starting line number and the increment can be omitted at the same time.

[2] Even if the starting line number is specified, the increment can be omitted.

[3] If the starting line number is omitted, the increment cannot be specified.
 5. For items enclosed with braces { }, select any one of the items.
- Explanation:** Detailed instruction functions and precautions for commands, subcommands, statements, functions and libraries are described.

ABS

● Function



Function: Returns the absolute value of x.

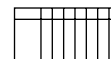
Format: ABS(x)

x : Numeric expression.

Explanation: Returns the absolute value of x.

ALLOCATE

● Statement



Function: Declares an array of variable size and allocates an array area in memory by determining the size and elements of the array.

Format:

- (1) ALLOCATE variable-name
- (2) ALLOCATE variable-name (size)
- (3) ALLOCATE variable-name \$ number-of-characters
- (4) ALLOCATE variable-name \$ number-of-characters (size)

Size: Specify the upper bound to subscripts as a numeric value or variable. Positive integer.

An array may be one or two dimensional.

Number-of-characters :

Up to 512 ASCII characters. The default value is 18. Positive integer.

Explanation: Formats (1) through (4) above are declaration statements for numeric variables, array numeric variables, character-string variables, and array element string variables respectively.

The ALLOCATE statement functions in the same way as the DIM statement, except that :
(See DIM statement)

- A variable can be specified as the array size. Hence, the array size can be adjusted to the conditions of program execution by varying this variable value.
- In main program, the ALLOCATE statement is the same as the DIM statement except that a variable can be specified as the array size.

APPEND (A)

- Command
- Subcommand



Function: In debug mode, this command appends a program stored in the auxiliary memory to the current program block in the user area. In subcommand mode, a program with two or more lines is converted to one line and is added to the specified line. At that time, ":" that means multiple statement is not appended.

If the one line has 253 bytes or more, an error occurs.

Format: (1) When APPEND (A) is used as a command:
 APPEND program-name [, start-line-number]
 Program-name (character string): [device-specification:] [path \] file-name [.extension]
 When [device-specification] is omitted, the current device is specified.
 Extension cannot be omitted as far as it is used.

(2) When APPEND (A) is used as a subcommand:
 APPEND destination-line-number, source-start-line-number [- source-end-line-number]

Explanation: When APPEND (A) is used as a command, it operates differently from when it is used as a subcommand.

(1) When APPEND (A) is used as a subcommand:
 The designated program is appended to the current program block in the user area, with its lines renumbered beginning with start-line-number. In a program, even if start-line-number is designated, line number is unaffected (no error occurs).
 When start-line-number is omitted, the lines in the appended program are renumbered, beginning with the maximum line number in the program block + 10.
 The program block in the user area to which a new program is to be appended must have been designated as the current program block by a PROG command.
 When the designated program is a subprogram, the subprogram is inserted between the current program block and the next program block.
 The program to be appended must be in source format or subprogram. (Programs in intermediate language cannot be appended.)



CAUTION

When specifying the start-line-number, it and each line number after appending specified program should not coincide with the line numbers in the user program block, or lines in the appended program may overlay existing lines in the user program block.

(2) When the APPEND (A) is used as a subcommand:
 Lines starting from the source start-line-number to source-end-line-number are edited in a line and added to the destination line number.
 At this time, a colon (:) is not inserted.

ARNAM

● Function



Function: Returns the value of an array variable element.

Format: ARNAM(c , m [, n])

c: Array variable name (character string expression).

m , n: Specify the position of an array (numeric expression).

Explanation: This function returns the value of an array variable element represented by character string c. Array elements can be designated indirectly.

For a one-dimensional array, specify only m (omit n). For a two-dimensional array, specify both m and n.

When a simple variable is indirectly designated, use NAM function.

ASC

● Function



Function: Returns a character code value.

Format: ASC(c)

c: Character string expression.

Explanation: The code value of the first character of a character string or character string variable c is provided. For correspondence of characters and code values, see "Appendix 1 LOCAL CODE LIST".

ASSIGN

● Statement



Function: Declares the use of modules.

Format: ASSIGN — { Module-ID
Sequence-ID } = Slot-number

FC030101.EPS

Module-ID: Four characters (character string) from the beginning of a module model

Sequence-ID: Four characters (character string) from the beginning of a module model

Slot-number: Numeric expression

Explanation: This statement declares the use of I/O module or CPU module of the specified slot number. This statement must be executed before accessing the module.

Declaration for the sequence CPU module is required only when a program is to be started or read status for a sequence CPU module using CONTROL statement, SEQACTV statement or STATUS statement.

Use this statement within the main program. If it is used in a subprogram, an error of "ASSIGN statement not executed" may occur in I/O access.

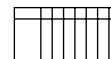
(Example)

Module-ID=YD32, XD32, YC16, RS22, ... etc.

Sequence-ID=MP30, MP20, AP10

ATN

● Function



Function: Returns the arc tangent of an argument.

Format: ATN(x)

x: Numeric expression.

Explanation: This function returns the arc tangent of x in radians. The range of the result is $-\pi/2$ thru $+\pi/2$ radians.

AUTO

● Command



Function Assigns line numbers automatically.

Format AUTO [_start-line-number [, increment]]

Increment: Positive integer. The default value is 10.

Explanation Entry of an AUTO command displays a line number, which is incremented automatically each time a statement is subsequently entered. Pressing only the ENTER key without entering a statement terminates the command and displays the prompt (BSC: or bsc:).

Format	Description
AUTO	Displays line numbers from the maximum line number plus 10 with the increment of 10. In the initial status, the line number starts at 10.
AUTO _start-line-number	Displays line numbers with the increment of 10, beginning with the start-line-number.
AUTO _start-line-number, increment	Displays line numbers with given increment, beginning with the start-line-number.

TC030101.EPS

When a statement with the current line number has already been entered, the line number is displayed with "*" in the first column.

Main programs and subcommands can be entered with the AUTO command.

The maximum line number is 65535. If a line number exceeding the maximum is entered, an error occurs.

BCD

● Function



Function: Returns a BCD integer converting decimal number.

Format: BCD(x)

x: Numeric expression.

Explanation: This function converts a numeric value or variable represented by "x" into a BCD integer and returns it. "x" must be an integer with a maximum of four digits. When a numeric expression contains digits after the decimal point, the numeric value — with any digits after the decimal point rounded off — is converted into a BCD number.

BINAND

● Function



Function: Returns the result of ANDing in an n bit by bit.

Format: BINAND(m , n)

m , n: Numeric expression.

Explanation: This function deals with bits (numeric values or variables represented by m and n are ANDed bit by bit).

Example : A=BINAND(\$FF00 , \$FFFF)

The value of A is \$FF00.

Only 16-bit integers are returned by this function.

BINNOT

● Function



Function: Returns one's complement of m.

Format: BINNOT(m)

m: Numeric expression.

Explanation: This function deals with bits (one's complement of the numeric value or variable represented by m).

Example : A=BINNOT(\$00FF)

The value of A is \$FF00.

Only 16-bit integers are returned by this function.

BINOR

● Function



Function: Returns m and n ORed bit by bit.

Format: BINOR(m , n)

m , n: Numeric expression.

Explanation: This function deals with bits (numeric values or variables represented by m and n are ORed bit by bit — see example below).

Example : A=BINOR(\$FF00 , \$0000)

The value of A is \$FF00.

Only 16-bit integers are returned by this function.

BINXOR

● Function



Function: Returns m and n exclusively ORed bit by bit.

Format: BINXOR(m , n)

m , n : Numeric expression.

Explanation: This function deals with bits (numeric values or variables represented by m and n are exclusively ORed).

Example : A=BINXOR(\$FF00 , \$FFFF)

The value of A is \$00FF.

Only 16-bit integers are returned by this function.

BIT

● Function



Function: Returns the bit of the specified bit position.

Format: BIT(m , n)

m: Numeric expression. m is provided only for 16-bit integer.

n: (1) Numeric expression.

(2) Character string or character string variable representing binary pattern.

Explanation: This function deals with bits.

When n=integer type expression (1) above.

Returns a bit value of the n-th digit (starting with the least significant digit 0) for binary expressions for m (numeric value or variable).

Example : A=BIT(\$0010 , 4)

Bit value for A is 1 in this example.

When n=character string or character string variable representing binary pattern (2) above.

When a pattern of binary expression for m (numeric value or variable) coincides with n (character string expression) representing binary pattern, 1 is returned; otherwise, 0 is returned.

For a bit which is not to be compared, set a capital X in the bit position corresponding to the character string (or character string variable) — see example below.

Example : A=BIT(\$0010 , "0XXXX00000010000")

In this example, the value of A is 1.

BLEN

● Function



Function: Returns the number of bytes in a character string.

Format: BLEN(c)

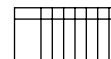
c: Character string expression.

Explanation: BLEN is used to provide the number of bytes for c (character string or character string variable).

If the number of characters is required on the standard character basis, use HLEN function and if the above is required by counting a large character as one character, use LEN function.

BYE

● Command



Function: Exits BASIC debug mode.

Format: BYE

Explanation: Ends the BASIC processing and exits the debug mode.

When there is a specified resident program (see SETMD RES command) and the BYE&RUN mode (see SETMD RUN command) is already specified, the resident program can be executed in REAL mode.



CAUTION

In non-resident mode, the program in the current user program area on display is erased. Before executing the BYE command, if the current version of the program has not been saved, save it using the SAVE command (refer to SAVE command).

Even though a BYE command is executed, the common area cannot be initialized (for details, refer to COM and INIT COM statements).

CALL

● Statement



Function: This statement is used to branch to a subprogram.

Format:

- (1) CALL $\underline{\hspace{1cm}}$ subprogram-name
- (2) CALL $\underline{\hspace{1cm}}$ subprogram-name(actual-argument , actual-argument ,)

Explanation: Format (1) above is used to branch to a subprogram requiring no actual arguments.

Format (2) above is used to branch to a subprogram requiring actual arguments.

However, when the CALL statement is used in the ON statement, arguments cannot be described.

The number and data types of actual arguments specified in format (2) must be matched with those of formal arguments defined in the SUB statement.

For actual arguments, the following variables can be used:

- (a) Numeric variables and character string variables.

Example :

```
CALL SI(A , B$ , C(*) , D$(*))
```

:

```
SUB SI(V , W$ , X(*) , Y$(*))
```

- (b) Numeric variables and character string variables (reference only).

Example :

```
CALL SA([ A ] , [ B$(*) ])
```

:

```
SUB SA(C , D$(*))
```

If such variables used as arguments in a subprogram are changed in the subprogram, an error may occur.

- (c) Numeric and character string constants.

Example :

```
CALL TS(3 , "ABC")
```

:

```
SUB TS(A , C$)
```

- (d) Numeric expressions and character string expressions (reference only).

Example :

```
CALL MA(10*2+3 , A$+"EF")
```

:

```
SUB MA(I , J$)
```

- (e) Common variable

Common variables are declared with a COM statement. Common variables can be used in the same format as in (a) or (b).

CALLLIB

● Statement



Function: Branches to a library program.

Format: [CALLLIB _] program-name [(parameter-1 [, parameter-2 ...])]

Program-name: Library name.

Explanation: This statement is used to branch to a library program in the same user area. Variables, numeric expressions, and array variables can be used as parameters. For more detailed instructions, see the Instruction Manual for Library Programs. The CALLLIB statement cannot be used to designate the destination to which a program is branched by ON... statement.

CALLLIB can be omitted.

CHG(C)

● Subcommand



Function: Converts the character pattern (character string).

Format: C [HG] _delm character-pattern-1 delm character-pattern-2 delm

C [HG] _delm character-pattern-1 delm character-pattern-2 delm [_starting-line-number – ending-line-number]

delm (delimiter) : This is a one-character symbol inserted to show the character pattern boundary. It must be a symbol that is not used in the character pattern.

Character-pattern 1: Pattern to be changed (up to 24 bytes).

Character-pattern 2: New pattern (up to 24 bytes).

The length of character-pattern-1 need not be the same as that of character-pattern-2.

Starting-line number: Line number to start conversion

Ending-line number: Line number to end conversion

Explanation: This subcommand is used only when activating the editor (refer to EDIT command).

The CHG (C) converts a character pattern (character string) including labels, variables, expressions, and statements to another character pattern.

Converted character patterns are displayed on the editor screen.

Any of the following characters and symbols can be used as the delimiter (delm), but they cannot be used in the character patterns (see below).

Characters and symbols used as delimiters for CHG (C) command
Lower-case alphabetical characters (a to z)
Symbols (! " # \$ % & ' () - ^ _ ` { @ ` [] { } + ; * , < > . / ? _)

TC030102.EPS

All character patterns (character-pattern-1) in the specified line number are converted. When the line number is not specified, the character pattern in the line where the cursor was positioned before executing the CHG (C) command, is converted to character-pattern-2. If the character pattern (character-pattern-1) does not exist in the specified range of the line numbers, an error will occur.

**CAUTION**

If character-pattern-1 or character-pattern-2 includes "=", the abbreviation (C) cannot be used for CHG.

Lower-case alphabetical characters are handled as follows:

- (1) when specifying a symbol other than ! as delimiter

Upper-case alphabetical characters are not discriminated from lower-case alphabetical characters. All are handled as upper-case alphabetical characters.

Example 1 10PRINT "ABC"

```
>C /BC/de/
```

```
10PRINT "ADE"
```

Example 2 10PRINT "abc"

```
>C /bc/de/
```

If Example 1 above is executed, an error "T1-E 121 specified character-string not found" will occur. This is because "bc" specified as character-pattern 1 was internally handled as "BC".

For Example 2 above, this conversion will be executed by changing delimiter to "!" (refer to Example 5 below).

If a character-string starting with "!" is specified as a character pattern, upper-case alphabetical characters are discriminated from lower-case alphabetical characters.

Example 3 10DEFINT A-Z

```
>C /def/!def/
```

```
10!defINT A-Z
```

- (2) When specifying symbol ! as delimiter

In the case of specifying statements, labels, or variable names in lower-case alphabetical characters, all are handled as upper-case alphabetical characters.

Example 4 10Z=X+Y

```
>C !Z!sum!
```

```
10SUM=X+Y
```

In other cases, upper- and lower case alphabetical characters are discriminated.

Example 5 10PRINT "Abc"

```
>C !bcd!e!
```

```
10PRINT "Ade"
```

CHR\$

● Function



Function: Returns a character corresponding to the code value specified.

Format: CHR\$(n)

n: Numeric expression.

Explanation: This function returns a pattern or character corresponding to code value n (numeric value or variable).

COM

● Statement



Function: Declares data transfers between multiple programs (including subprograms).

Format:

- (1) COM common-variable-name [, common-variable-name , ...]
- (2) COM #Sn common-variable-name [, common-variable-name , ...]

n (slot-number): Numeric value. Number of the slot in which the CPU card is mounted (01 to 04)

Explanation: Both programs for sending and receiving data require COM statements. Two formats for COM statement are provided according to applications.

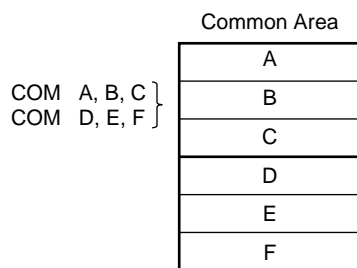
Format (1) reserves the data area referred to by other programs in its own user area.

Using format (2) data reserved by the programs in other areas is referenced.

Format	Purpose	Description	Applicable area
(1)	reservation	<ul style="list-style-type: none"> Declare the data area referred to by other programs in its own user area. Specify common data for use by subprograms in the same user area. 	F3BP20 F3BP30
(2)	reference/ reservation	<ul style="list-style-type: none"> Permit access to COM data stored in another FA500 (BASIC) or YEWMAC line controller program. However, data reserved by a YEWMAC line computer cannot be accessed by a FA500 (BASIC). Common variables are assigned to common registers of the specified sequence CPU in the order described in the program. 	F3BP20 F3BP30

TC030103.EPS

As many common variables as will fit in a line can be specified in a COM statement by delimiting them with commas “,”. Any given common variable name cannot appear more than once in COM statement(s) in a program. Care is required not to assign local variables (variables to be accessible in one program block only) the same names as common variables. If COM statement is described by dividing it into multiple pages, the area subsequent to areas defined for the previous COM statement, is used.



FC030102.EPS

All references to a given set of COM data must declare the correct number of data, and variables with the same data types must be in the same positions in the list. The variable names used to reference the data may be the same or different in different programs. Variables declared in a COM statement do not need to be declared by DIM or ALLOCATE statements. Variables used as internal parameters in such specific statements as FOR–NEXT cannot be used in COM statements.

Notice for programming

- For character string variables, use an even number of bytes for the size of a common variable. If an odd number of bytes is used, an error may occur in a library, or transfer rate may be reduced.
- The common variable size is set by high limit of the subscript. However, a positive integer must be used for this set value.
- If common variables specified with COM #Sn are used in a program, data is exchanged via the ML-BUS as frequently as the common variables are used.

Thus, it is recommended that programs be written so as to reduce references to common variables by transferring common variables into local variables (variables used in a program or subprogram only) in order to increase the speed of program execution.

- To transfer an common array variable to a local variable, it is convenient to use a MOVE statement.

CONT

● Command



Function: Restarts a program suspended by a PAUSE statement or pressing [ESC] key.

Format: CONT

Explanation: This command is used in debug mode. The FA-M3 requires to be connected with a personal computer to execute this command.

When the execution of a program has been suspended by entering a PAUSE statement or pressing [ESC] key, after checking the variable values by executing an immediately executable statement, the program can be restarted by the CONT command.

An error message is displayed if the CONT command is entered when program execution is not suspended.

Additionally, the CONT command is invalid and cannot restart the program when:

- a program source is modified (including EDIT),
- a program block is changed (using PROG command).

When the execution of a program is suspended by a STOP or END statement, it cannot be restarted by the CONT command.

CONTROL

● Statement



Function: Defines module specifications by setting parameters in modules.

Also sets special operations specific to a module.

In addition, performs a ladder sequence program start/stop.

Format: (1) CONTROL $\underline{\hspace{1cm}}$ slot-number [[, instrument-number [, port-number]], function-number ...] ; data

(2) CONTROL $\underline{\hspace{1cm}}$ slot-number , 1 ; start-stop-parameter

Slot-numbe: Numeric expression (integer type).

Instrument-number: Numeric expression. (Specifying possibility varies with the I/O card.)

Port-number: Numeric expression. Can be omitted for modules having the port number.

Function-number: Numeric expression. Register No. is 1 to 99. Numbers other than registers are 101 to 164.

Data: Numeric expression (integer type). Multiple data can be specified. Collective specification of array possible (However, not possible if a register is omitted.)

Start-stop-parameter: Numeric value.

1: Stop

2: Start (reset and start)

3: Start (holding and start)

Explanation: In formats (1), set the operation parameters to the module control registers.

Format (2) is used when the FA500 BASIC program starts/stops the FA500 ladder sequence programs.

For details of format (1), see instruction manuals for modules.

COS

● Function



Function: Returns the cosine of x.

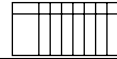
Format: COS(x)

x: Numeric expression.

Explanation: This function returns the cosine of the numeric value or variable represented by x (in radians).

DATA

● Statement



Function: Sets data to be assigned to variables specified in a READ statement.

Format: DATA $\underline{\hspace{1cm}}$ constant , constant ,

Constant: Numeric or character string constant.

Explanation: DATA statements may appear anywhere within a program, and several DATA statements can be used within a single program. If a quotation mark (") is to be used in a character string, a pair of quotation marks (" ") must be entered for it to be treated as a character string constant. It is optional for character string data to be enclosed in quotation marks when it does not include commas (,) or spaces. A DATA statement cannot be used in a multiple statement line; that is, a DATA statement cannot be entered after a colon (:). A space can be specified as a constant when it is to be assigned to a character string variable specified in a READ statement. But a space, if specified for a numeric variable, is ignored. A DATA statement to be read by a READ statement can be specified line by line by a RE-STORE statement. Multiple statements cannot be used after a DATA statement (including comments with REM statement).

DATE\$

● Function



Function: Returns the date.

Format: DATE\$

Explanation: The current date is always entered into DATE\$ and can be determined as a character string of a definite format. DATE\$ allows the date to be returned in the character string yy/mm/dd.

The "year" is expressed in the lower two digits of Christian era.

DEF FN

● Statement



Function: Defines a user function (arbitrary expression).

Format: DEF function-name(formal-argument-list)=expression

Function-name: A string of eight or less characters (alphanumeric) headed by FN.

Formal-argument-list: A list of formal arguments used in the expression. May be omitted.
(character string)

Expression: Numeric value or variable, or character string or character string variable.

Explanation: A function, once defined in the DEF statement, can be referenced and the result of the function can be obtained during a program. When a function-name has been specified as a simple numeric variable name, the function is defined as a numeric-type user-defined function, for which the right side expression must be a numeric expression. Likewise, when a function-name has been defined as the name of a simple character variable, the function is defined as a character-type user-defined function, and the right side expression must be a character string expression.

Note: The following instructions should be observed when defining a user function:

- A user function must be defined in a program before it is referenced in the same program.
- A function cannot be defined in two or more places in a program.
- If formal arguments have been specified in the definition of a user function, the user function must always be referenced with actual arguments when it is to be used within a program. If the formal argument is of numeric type, a numeric expression must be used as an actual argument; if the formal argument is of character string type, a character string expression must be used as an actual argument.
- The formal arguments used in defining a function have an effect only on the right side expression but have no bearing on any simple variable having the same name in another statement.
- Multiple formal arguments can be used by separating them with a comma (,). See the example below.

Example : DEF FNKANSU(X , Y)=100+10*X+Y

- For the variable which is not formal argument in the right side expression, the value is assigned when the function is referenced. This variable must have been defined before the function is referenced.
- The defining expression must not contain the function to be defined. (No recursive expressions)
- The character string expression defined by a character string type function can link a character string (up to 512 characters long).
- The function defined by a DEF FN statement is valid in the program block defined.
- In the format above, "function-name (formal-argument-list) = expression" cannot be changed to a character string expression.

DEFAULT

● Statement



Function: Declares implicit processing on computer errors.

Format: (1) DEFAULT _ ON
(2) DEFAULT _ OFF

Explanation: In DEFAULT ON status, when overflow occurs as a result of computation, the computation is completed by using a default value for out-of-range result and then error "Computational overflow" occurs. In DEFAULT OFF status, the computation is aborted immediately when overflow occurs as a result of computation and an error occurs. As data is not processed to store it, the storing variable (left side) holds the previous value. The initial status is DEFAULT ON status.

In DEFAULT ON status, if computation results in overflow, the default values are as follows :

	Overflow of positive number	Overflow of negative number
Integer	32767	-32768
Long integer	2147483647	-2147483648
Single-precision number	9.223372×10^{18}	-9.223372×10^{18}
Double-precision number	$9.223372036854776 \times 10^{18}$	$-9.22337203685576 \times 10^{18}$

TC030104.EPS

DEFINT/DEFLNG/DEFSNG/DEFDBL

● Statement



Function: Declares variable types.

Format: (1) DEFINT _ Initial-of-variable-name [, initial-of-variable-name]
(2) DEFLNG _ Initial-of-variable-name [, initial-of-variable-name]
(3) DEFSNG _ Initial-of-variable-name [, initial-of-variable-name]
(4) DEFDBL _ Initial-of-variable-name [, initial-of-variable-name]

Explanation: The variable types must be declared. All variables initiated from a character specified by "initial-of-variable-name" are the specified types.

In DEFINT, an integer type variable is declared.

In DEFLNG, a long integer type variable is declared.

In DEFSNG, a single-precision real type variable is declared.

In DEFDBL, a double-precision real type variable is declared.

A character to be specified by the initial of a variable is one upper-case alphabetical character.

When several initials are to be specified in alphabetical order, use a hyphen to join alphabetical characters.

In any of four formats (1) to (4), declaration must be made before the description of variable names in the program block. Description of multiple statements is not permitted. If the variable type declaration using these statements is omitted, a single-precision type is used.

Even when a variable - passed in a subprogram as an argument - is declared as some variable type in the subprogram, this declaration is ignored and the variable type on the calling side is followed.

DEL

- Command
- Subcommand



Function: Deletes a specified lines or the whole of a specified subprogram or specified library program in user program area.

Format: DEL $\left\{ \begin{array}{l} \text{start-line-number [, end-line-number] (can be used as an EDIT subcommand)} \\ \text{subprogram-name (cannot be used as an EDIT subcommand)} \\ \text{library-name (cannot be used as an EDIT subcommand)} \end{array} \right.$

FC030103.EPS

Explanation: The current program block (refer to PROG command) will be activated when line numbers are specified.

Format	Description
DEL $\underline{\hspace{1cm}}$ start-line-number	Deletes only the line with start-line-number.
DEL $\underline{\hspace{1cm}}$ start-line-number, end-line-number	Deletes all lines from start-line-number to end-line-number.

TC030105.EPS

If a specified line number does not exist, an error will occur.

When a subprogram-name is specified, the named subprogram is deleted as a whole.

When a library-name is specified, the whole of the named library program in the user program area is deleted.

SUB statements cannot be deleted. (An error message is displayed if deletion of a line containing a SUB statement is attempted.)

DIM

● Statement



Function: Declares an array and allocates an array area in memory corresponding to the declared size (number of elements) in the array.

Format:

- (1) DIM variable-name
- (2) DIM variable-name (size)
- (3) DIM variable-name \$ number-of-characters
- (4) DIM variable-name \$ number-of-characters (size)

Size: Specifies the upper bound to subscripts as a numeric constant (positive integer). An array may be one or two dimensional.

Number-of-characters: Up to 512 characters in ASCII code. The default value is 18. Positive integer.

Explanation: Formats (1) through (4) are used for the declarations of a numeric variable, an array numeric variable, a character string variable, and an array character string variable respectively. The maximum number of subscripts is 32,767 in one or two dimensional array. A variable name already used as a simple variable cannot be declared as an array name in the DIM statement. An array name, once declared, cannot be declared in another DIM statement.

When an array is declared, all elements of the array are set to 0 if the array is of numeric type, or to null character strings if the array is of character string type.

The lower bound to array subscripts in each dimension is specified in an OPTION BASE statement (see OPTION BASE). If a value of 513 or more is specified as number-of-characters, an error occurs.

The array must always be declared in a DIM, ALLOCATE, or COM statement (implicit declaration of arrays is not possible.)

However, specification for whole array passed by the subprogram argument need not be declared in the subprogram.

When several variable names are declared in a DIM statement, separate them with commas “,”.



CAUTION

- The DIM statement cannot use variables to declare array size. Use an ALLOCATE statement when variables are to be used to declare the size of an array.
- The DIM statement cannot be used in multiple-statement lines.

DISABLE

● Statement



Function: Disables the branch declaration temporarily by ON statement.

Format: DISABLE [_ C]

Explanation: The DISABLE statement disables the interrupting branch declaration (ON) temporarily (except for ON ERROR statement).

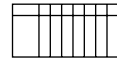
The DISABLE C statement disables the branch declaration (ON ...) which has been made in the same program block before execution except for ON ERROR statement. The branch declaration after DISABLE C statement execution or in the other program blocks is not disabled. This temporary inhibition of branch declaration set by DISABLE/DISABLE C statement is reset by ENABLE/ENABLE C statement.

The DISABLE/ENABLE statement is valid even in other program blocks.

The DISABLE C/ENABLE C statement is valid only in the same program block. However, the disabled state continues in other program blocks. Further, if the DISABLE statement is used in a program branched by an ON...GOSUB or ON...CALL statement, the status is valid when the level returns to low. Also, while branching is disabled by DISABLE statement, the status is automatically reset if WAIT statement is executed. However, the branching disabled by DISABLE C is not reset.

DISP (DP)

● Statement



Function: Outputs a display list to CRT screen in a personal computer or a YEWMAC line computer.

Format: DISP _ display-list

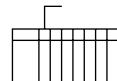
Display-list : Variable names, array variable names, numeric values, or character string expressions, separated by a comma (,) or semicolon (;).

DISP may be abbreviated as DP.

Explanation: Details, such as output items and delimiters, are the same as those for PRINT statement (see PRINT statement) except that this statement is for output to CRT screen only.

DISP USING (DU)

● Statement



Function: Outputs a formatted display list on the CRT screen.

Format: DISP _ USING _ { image-specification
label
line-number } ; display-list

FC030104.EPS

Image-specification: Format designation(see IMAGE statement provided later)

Line-number: IMAGE statement line number. Numeric value.

Label: IMAGE statement label.

Display-list: Variable names, array variables, numeric values, character string expressions, separated by a comma (,).

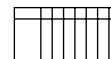
DISP _ USING can be abbreviated as DU.

Explanation: Details, such as output items and delimiters, are the same as those for PRINT USING statement. For more information, see PRINT USING and IMAGE statements.

This statement is valid only in debug mode and ignored in real mode.

DIV

● Function



Function: Returns a result of integer division.

Format: DIV(x , y)

x , y: Numeric expression.

Explanation: INT(x)/INT(y) is calculated and the digits after the decimal point are rounded off. Integers, long integers, single-precision real numbers or double-precision real numbers are used for x and y. For INT(x), see the INT function.

EDIT

● Command
● Subcommand

Function: Activates a line editor in debug mode.

Format:

$$\text{EDIT} \left[\begin{array}{l} \text{Subprogram-name} \\ \text{EDIT_start-line-number} \\ * \end{array} \right]$$

FC030105.EPS

Explanation: The EDIT command is usually used for editing or debugging the program.

This command can also be used in subcommand mode.

When the EDIT command is executed, one line of the program is displayed and the next line is started with the prompt ">", which is waiting for a subcommand to be entered.

Subcommands can be entered in this condition. Furthermore, this program line comes with a template, enabling you to edit using a special edit function of MS-DOS.

When the subprogram name is omitted, the program block last specified by the PROG command is accessed.

If the specified subprogram name does not exist, the main program is accessed.

When entering EDIT command, if an error "command error, command disabled state" occurs, specify the main program block by PROG command, and again enter the command.

Format	Description
EDIT	Displays the program from the beginning of the program block.
EDIT__start-line-number	Displays the program from the start-line-number onward.
EDIT__subprogram-name	Displays subprogram from the head of the specified program.
EDIT__*	<ul style="list-style-type: none"> When the EDIT command has already been used in the same program block: The display returns to the last EDIT display. In other cases, the program listing is displayed starting with the head of the program block.

TC030106.EPS

Editor

In Editor screen, it is possible to insert/delete lines line by line.

Enter QUIT or Q to end the editor and to return to command entry screen.

The commands that can be used to activate the editor are called subcommands.

Subcommands are as follows. For details, refer to the descriptions of the subcommands in this manual.

Subcommand	Subcommand (abbreviations)	Function summary
FIND	F	Searches for character strings.
QUIT	Q	Changes from editor screen to command entry screen.
CHG	C	Interchanges character strings.
LCOPY		Copies lines.
EDIT		Displays a specified program from its head or from the specified line number.
DEL		Deletes program lines.
RENUM		Resets program line numbers.
APPEND	A	Changes multiple program lines to a single program line.
LIST	L	Displays program listing.
ERLIST		Displays erroneous program lines as a program list.

TC030107.EPS

List of special edit functions

Key	Edit function	
[F1] [→]	COPY 1	Copies one character from template to command line.
[F2]	COPY UP	Copies characters before the specified character from template to command line.
[F3]	COPY ALL	Copies all characters remaining in template to command line.
[F4]	SKIP UP	Skips to characters just before the specified characters (do not copy)
[F5]	NEW LINE	Copies the contents of command line to template (create a new template without pressing Enter key).
[↓]	VOID	Deletes the current command entry and goes to a new line, though contents of template are not changed.
[Ins]	INSERT MODE	Will activate Insert Mode.
[Del]	SKIP 1	Skips one character in template (do not copy).

TC030108.EPS

ELSE

● Statement



Function: Indicates the start of ELSE block added to a structured IF THEN block.

Format: ELSE

Explanation: The ELSE statement is optionally added to a structured IF THEN block. When the ELSE statement is used, the ELSE statement is described separately in one line and conditional statements are described in the following line and thereafter. For further details, see IF statement.

ENABLE

● Statement



Function: Releases branch declarations disabled by the DISABLE statement.

Format: ENABLE [$\underline{\hspace{0.5cm}}$ C]

Explanation: All branch declarations which were disabled before the ENABLE statement was executed are released. Only the branch declaration disabled by the last DISABLE C statement is released by the ENABLE C statement.

The DISABLE statement can be released by any program block. However, the DISABLE C statement can be released, only by a statement in the same program block.

ENABLE INTR

● Statement



Function: Sets a factor of hardware interruption to an I/O module.

Format: ENABLE $\underline{\hspace{0.5cm}}$ INTR $\underline{\hspace{0.5cm}}$ slot-number [, instrument-number] ; mask

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression. The default possibility varies with I/O modules.

Mask: Character string or character string variable.

Explanation: This statement sets a factor of hardware interrupt to an I/O module to enable a program to be branched.

Since the format is set to an interrupt disabled status if interruption is once generated, it must be declared again in the interrupt subroutine. In this case, if the statement is used before the ENTER statement, the program branches twice causing an error. Always declare this statement after the ENTER statement. The mask pattern is a character string that indicates the following contents.

"X₁, X₂, X₃"

X₁: Receiving completion interrupt

X₂: Error interrupt

X₃: No-data interrupt

In any case, status 1 shows that the interrupt is possible. For more information, see instruction manuals for each I/O module.

END

● Statement



Function: Terminates a main program.

Format: END

Explanation: The END statement also designates the end of main program block. Thus the END statement must appear as the last program line, and cannot be omitted.

For a program with no STOP statement, this statement functions to terminate program execution.

END WHILE

● Statement



Function: Terminates a structured WHILE block.

Format: END ┘ WHILE

Explanation: See WHILE statement.

ENDIF

● Statement



Function: Terminates a structured IF THEN block

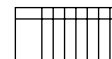
Format: ENDIF

Explanation: The structured IF THEN block is terminated with an ENDIF statement. Multiple statements on the same line separated by colons cannot be used (see IF statement).

Always use this statement latest to end the structured IF ... THEN statement.

ENTER

● Statement



Function: Enters the data from an I/O module, an I/O buffer or a sequence device.

Format: (1) ENTER $\underline{\hspace{1cm}}$ slot-number [, instrument-number]

$$\left[\begin{array}{l} \left\{ \begin{array}{l} \text{FUSING} \\ \text{USING} \end{array} \right\} \text{---} \left\{ \begin{array}{l} \text{image-specification} \\ \text{line-number} \end{array} \right\} \\ \left\{ \begin{array}{l} \text{FORMAT} \\ \text{NOFORMAT} \\ \text{BFORMAT} \end{array} \right\} \end{array} \right] ; \text{Input-variable}$$

FC030106.EPS

(2) ENTER $\underline{\hspace{1cm}}$ slot-number [, instrument-number [, terminal-number]]

(3) ENTER $\underline{\hspace{1cm}}$ input-buffer-name ; input-variable

(4) ENTER $\underline{\hspace{1cm}}$ slot-number , device-name string expression $\left[\text{---} \left\{ \begin{array}{l} \text{NOFORMAT} \\ \text{BFORMAT} \end{array} \right\} \right]$

FC030107.EPS

; input-variable

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression. Possibility for omission varies with the I/O modules.

Terminal-number: Numeric expression (integer type).

Input-buffer-name: Simple variable or array variable (numeric variable or character string variable), though common variables cannot be used.

Line-number: Line-number of IMAGE statement. Numeric value.

Image-specification: Character string (see IMAGE statement).

Input-variable: Simple variable or array variable (Numeric variable or character string variable).

Device-name character string expression :

Character string expression representing a sequence device name.

Explanation: Format (1) enters the data in complete-return type from I/O modules other than multiple transmission modules and format (2) enters the data in complete-return type from multiple transmission modules.

For asynchronous inputs from serial communication I/O modules, use ON INT statement for receiving asynchronous signals and enter them using ENTER statements. If ENTER is directly used to receive asynchronous signals without ON INT interruption, the user program being executed cannot proceed to the next step until the current input processing is completed.

Format (3) enters the data of I/O buffers entered with TRANSFER statement. Necessity of format specification such as NOFORMAT depends on I/O cards. For details, see PART I of this manual, "7.4 Access To Contact I/O Module" and instruction manuals for each I/O module.

Format (4) is used for entering the sequence device data.

If an error occurs in ENTER statement when the status information variable is specified in SET STATUS statement, store the error code to the status information variable. In this case, no system error occurs.

ERLIST

● Command



Function : Outputs a list of only program lines containing errors.

Format: ERLIST

Explanation: For the current program block (see PROG command), the ERLIST command outputs a list of only program lines in which the syntax check functions have found errors.

The output destination is a personal computer display. When this command is executed after pressing [CTRL] key and [P] key together, the output will be printed out to a printer connected to the personal computer.



CAUTION

In YM-BASIC/FA, to allow a CALLLIB statement to be omitted, even if an incorrect word (for example, PRIN for PRINT) other than reserved words are entered, it appears to be a library name and the ERLIST does not detect any error. When the program is executed, an error occurs.

ERRC

● Function



Function: Returns an error code.

Format: ERRC

Explanation: Returns the error code (numeric value 1 to 255) for the latest error (see "4. ERROR CODE LIST"). A detail error code for I/O access error, etc. is provided by the ERRCE function. The line number in which the error occurs is provided by the ERRL function.

ERRCE

● Function



Function: Returns a detail error code.

Format: ERRCE

Explanation: Returns a detail error code (numeric value) for the most recent error occurred when I/O is accessed. An error code (returned by the ERRCE function) is used together with an error code (returned by the ERRC function). (See "4. ERROR CODE LIST"). The error code returned by the ERRCE function is displayed in hexadecimal format. For ERRCE display, correspondence with error code is easier in hexadecimal display converted by the HEX\$ function. The ERRL function returns a line number in which the error occurs. ERRCE is 16 bit integer value.

ERRL

● Function



Function: Returns the line number in which the most recent error occurred.

Format: ERRL

Explanation: Returns the line number (numeric value) in which the most recent error occurred.

The numeric value given by the ERRL function is a single-precision real number.

The error code is given by the ERRC function.

EXP

● Function



Function: Returns the value of the x-th power of e, the base of natural logarithm.

Format: EXP(x)

x: Numeric expression.

Explanation: This function returns the value of x-th power of e, the base of natural logarithm (transcendental number) (x is numeric value or variable). If the value of x is greater than 4366827237527656, an error occurs as computational overflow (x is double precision real number).

FIND (F)

● Subcommand



Function: Searches for a specified character string.

Format: FIND [delm] character-string [delm] or F [delm] character-string [delm]

Delm (delimiter): This is a one-character symbol inserted to show the character pattern boundary. The delimiter symbol must be a symbol that is not used in the character pattern.

Character-string: Arbitrary character string (up to 24 bytes) to be searched for. A space is also regarded as one character.

Explanation: FIND(F) is used only as a subcommand.

Searches for a specified character string backward from the line next to that of the current cursor position in the display screen (in ascending order of line numbers) and positions the cursor at the first character of the character string found.

Searching forward cannot be executed. So move the cursor to the head of the character string before executing another search (the cursor moves to the head by entering EDIT then pressing Enter key).

Characters and symbols, which are not being used in a character pattern, can be used for delimiters (delm) (see below).

Characters and symbols used as delimiters for FIND (F) command
Symbols (! " \$ % & ' () - ^ _ ` @ [] { } + ; * , < > . / ? _)

TC030109.EPS

The delimiter may be omitted; however, In this case the only types of character string that can be specified are a statement name, variable name, label, or branched line number of a program. Such words must be written completely (not abbreviated). For example, for the variable name BASIC, the character string BAS is not permitted (however, when delimiters are used, BAS can be specified).

After the FIND command is executed, the status is subcommand-wait status.



CAUTION

If the character string contains "=", or "=" is used as the delimiter, the abbreviation (F) cannot be used.

FOR-NEXT

● Statement



Function: Executes a specified sequence of statements a specified number of times in a FOR-NEXT loop.

Format: FOR variable = initial-value TO final-value [STEP increment]
NEXT variable

Variable: Simple numeric variable (a common variable and argument in a subprogram are not allowed); used for counting. The same variable must be specified for FOR and NEXT.

Initial-value, final-value, increment: Numeric value or variable.

When STEP is omitted, the increment is assumed to be one (+1) by default.

Explanation: The FOR NEXT statements are always used in a pair and are executed in the following way.

- (1) The initial- and final-values and the increment are computed.
- (2) The initial-value is assigned to the variable. A common variable cannot be used.
- (3) The statements between FOR and the corresponding NEXT statement are executed. To exit from the FOR loop (part of the program enclosed between the FOR and NEXT statements), execute a GOTO or GOSUB statement during the loop period.
- (4) When control reaches the NEXT statement corresponding to FOR, the variable is incremented by the step value when an increment is positive or decremented when the increment is negative.
- (5) The resultant value of the variable is compared with the final-value. The program is executed depending on the sign of the increment as shown below.

When the increment is positive:

Control returns to (3) if variable \leq final-value. The statement next to the NEXT statement is executed if variable $>$ final-value.

When the step is negative:

Control returns to (3) if variable \geq final-value. The statement next to the NEXT statement is executed if variable $<$ final-value.

Since the computation of the final-value and the increment is completed when control encounters a FOR statement, the final-value and the increment are unaffected if the values of the variables used in these numeric expressions are varied during execution of the FOR-NEXT loop.

The variable values are not changed when control exits from the FOR-NEXT loop on execution of a GOTO statement, etc.

The variable values are also not changed when control exits from the FOR-NEXT loop in operation (5). Note that, when the value of the variable specified in the FOR statement is altered before execution of the corresponding NEXT statement or when the increment is set to 0 (which results in an infinite loop), operations (2) and (3) are executed based on these values.

If the increment is non-interger decimal step value, the final-value may not be executed owing to the error of real operation.

Notice for programming

- When a pair of FOR and NEXT statements is nested in another pair of FOR and NEXT statements, the inner pair of FOR and NEXT statements must be completely inside the outer pair of FOR and NEXT statements. Also, both the order of line numbers and the order of execution must be properly arranged.
- A FOR statement having the same variable name cannot be written within the range up to NEXT.
- A FOR statement having the same variable name cannot be written in a subroutine called up within the range up to NEXT.
- During execution of a FOR statement, another FOR statement having the same variable name cannot be written in a subroutine branched by an interrupt caused by an ON GOSUB statement.

FREE

● Command



Function: Displays the free program area in bytes.

Format: FREE

Explanation: This command displays the free user program area in bytes (hexadecimal).

Common areas are not included. Using DISP or PRINT with the FREE function displays free program area as a decimal expression.

Free PC area = personal computer free user program area

Free FA area = FA-M3 free user program area

For the FA-M3 free user program area, part of operation area will be reserved when executing a program, so the free area is less than just after the program is loaded.

FREE

● Function



Function: Returns free (user area) program area size in the user area in bytes.

Format: FREE

Explanation: When the numeric value obtained by the FREE function is assigned to a variable, if the numeric value is an integer, overflow may occur. Use a single-precision real number for the variable.

GOSUB–RETURN

● Statement



Function: A program branches to a subroutine or returns from the subroutine.

Format: GOSUB — { label
line-number }
RETURN [—RETRY]

FC030108.EPS

Explanation: This statement passes control to the subroutine that begins at a specified line number; the RETURN statement terminates the subroutine execution and passes control to the statement following the GOSUB statement. A label can be used as a line number. The GOSUB statement enables the same procedure (subroutine) to be used repeatedly within a program.

One subroutine can contain another subroutine. The nesting level depends on the available working area.

An error occurs if the specified line number does not exist in the program or if a RETURN statement is executed without previously executing a corresponding GOSUB statement.

For the RETURN RETRY, see RETURN RETRY statement.

GOTO

● Statement



Function: Branches to a specified line unconditionally.

Format: GOTO — { label
destination-line-number }

FC030109.EPS

Explanation: If the specified destination-line-number or label is not used in the program, an error occurs.

Notice for programming

An infinite loop using the GOTO statement to test if an event has occurred, such as “100 GOTO 100”, should be avoided (it simply wastes CPU time). Use a WAIT statement with an ON (interrupt) request instead — to put the program to sleep until the event (e.g., key entry) occurs.

HALT

● Statement



Function: Halts I/O operations of I/O module.

Format: HALT — slot-number [, instrument-number]

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression (integer type).

Explanation: This statement unconditionally halts (cancels) the specified I/O operations of the I/O whose I/O has been started by TRANSFER statement.

For details, see instruction manuals for each I/O module.

If an error occurs in the HALT statement when the status information variable is specified in SET STATUS statement, store the error code to the status information variable. In this case, no system error occurs.

HEX\$

● Function



Function: Converts a decimal number to a hexadecimal number and returns its character string.

Format: HEX\$(x)

x: Numeric expression (–32768 to 32767).

Explanation: Returns a character string consisting of a hexadecimal number (converted from a numeric value or variable represented by x). Digits after the decimal point included in the value of x are rounded off. If the hexadecimal number is over four digits (value of x is outside the range –32768 to 32767), an error occurs.

HINSTR

● Function



Function: Searches for an arbitrary character string in a character string, and returns the position at which the character string begins.

Format: HINSTR(c1 , c2 [, n])

c1: Character string expression.

c2: Character string expression.

n: Numeric expression.

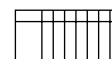
Explanation: This function searches for the character string represented by c2, starting from the n-th position from the head of the character string represented by c1 and returns a numeric value corresponding to the position of the beginning of the character string c2. If n is omitted, the character string c2 is searched for from the beginning of the character string c1.

As the position, a numeric value counted from the head of c1 which is assumed as 1 is returned. When the character string c2 is not found, the function returns the numeric value 0.

If a large character is used as one character, use the INSTR function.

HLEFT\$

● Function



Function: Takes out a character string of arbitrary length starting from the leftmost position of another character string and returns it.

Format: HLEFT\$(c , m)

c: Character string expression.

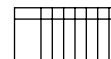
m: Numeric expression.

Explanation: This function takes out a character string of m characters starting from the leftmost position of a character string represented by c and returns the character string of m characters.

If m exceeds the total number of characters in character string c, the whole character string c is returned. If a large character is used as one character, use the LEFT\$ function.

HLEN

● Function



Function: Returns the total number of characters of a character string.

Format: HLEN(c)

c: Character string expression.

Explanation: This function returns the total number of characters of a character string represented by c. It also counts spaces. This allows the length of the character string in printing or display to be determined.

When a large character is used as one character, use the LEN function.

HMID\$

● Function



Function: Returns a character string of specified length from another character string.

Format: HMID\$(c , m [, n])

c: Character string expression.

m , n: Numeric expressions.

Explanation: This function returns a character string of n characters, starting from the m-th character from the leftmost character of a character string represented by c.

If n is omitted, a character string of all the characters to the right beginning with the m-th character is returned.

When a large character is used as one character, use the MID\$ function.

HRIGHT\$

● Function



Function: Returns a character string of specified length to the left from the rightmost character of another character string.

Format: HRIGHT\$(c , m)

c: Character string expression.

m: Numeric expression.

Explanation: This function returns a character string of m characters counted to the left from the rightmost character of another character string represented by c.

If m is equal to or larger than the total number of characters of the character string represented by c, a character string equal to the character string represented by c is returned.

When a large character is used as one character, use the RIGHT\$ function.

IF ... THEN

● Statement

Function: Branches depending on the result of a relational expression or a specified condition.

Format:

- (1) IF __ expression __ THEN __ { label
line-number } __ [ENDIF]
- (2) IF __ expression __ THEN __ statement [__ELSE__statement]__[ENDIF]
└─> Line-number and label are also allowed.
- (3) IF __ expression __ THEN

}
[ELSE]
}
ENDIF

FC030110.EPS

FC030110.EPS

Statement: Multiple statements can be used, but any statement that is not allowed in a multiple statement structure is excluded. Nesting of IF statements is allowed.

Explanation: A statement next to THEN is executed when the condition is true (other than 0); a statement next to ELSE is executed when the condition is false (0). Relational expressions such as (A>0) or the like are used. Other expressions may be used if the result of an expression is numeric.

Format (1) above.

When the condition is true ($\neq 0$), a program branches to the specified line number or label.

Format (2) above.

When the condition is true ($\neq 0$), the statement next to THEN is executed; when the condition is false ($=0$), the statement next to ELSE is executed. This format must be described in one line. GOTO and GOSUB statements can also be described. However, for these statements, using Format (3) above allows easier programs to be generated. In Format (2), statements following ELSE may be omitted.

Format (3) above.

When the condition is true ($\neq 0$), multiple statements from THEN to ELSE or ENDIF are executed. When the condition is false ($=0$), multiple statements (over multiple lines) from ELSE to ENDIF are executed. Statements following ELSE may be omitted. Execution of statements is ended by ENDIF. This format allows a program to be branched regardless of line numbers (labels). This format can also be used for multiple IF statements.

IFPCNV

● Library



Function: Converts numeric data from IEEE floating-point format to YM-BASIC/FA internal representation, and vice versa.

Format: [CALLIB _] IFPCNV (icmnd, input, form\$, output, ierr)

Explanation: This library is used to convert numeric data from IEEE floating-point format to YM-BASIC/FA, and vice versa.

Each parameter of this library is described as follows:

icmnd: Specifies the conversion method using integer-type, simple numeric variables.

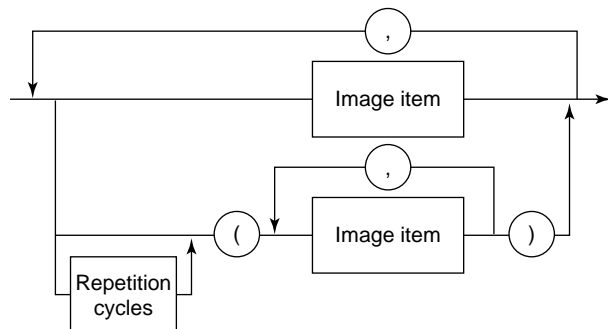
0: Conversion from YM-BASIC/FA internal representation to IEEE floating-point format

1: Conversion from IEEE floating-point format to YM-BASIC/FA internal representation

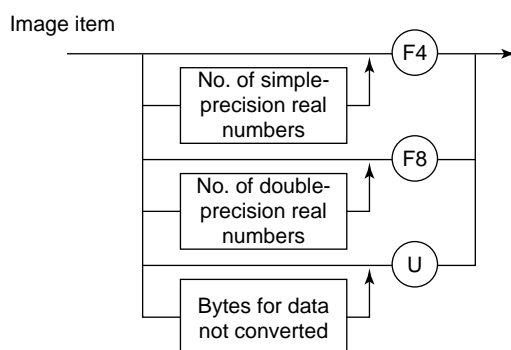
input: Specifies input variable data (numeric variable or character-string variable).

Either of simple variable or array variable can be used. For numeric variables, use either single-precision real number or double-precision real type. Conversion will be executed from the beginning of numeric variables or character-string variables given to this parameter.

form\$: Specifies image specification for conversion. Simple character-string variables are used. If a meaningless space is included, an error will occur.



Positive integer values
(up to 32767)



Example: FORM\$="4F4,8,(2F8,6U)"
FORM\$="3(10U,F8,F4)"

FC030111.EPS

output: Output-data-assigned variable (numeric variable or character-string variable). Either simple variables or array variables can be used. When numeric variable is used, either single-precision real number or double-precision real type can be used.

Output parameters must have the same variable size as that given to input parameters. Converted data will be assigned at the beginning of this parameter.

ierr: Integer type, simple-numeric variable. If an error occurs during execution of this library, the error codes below will be assigned. When there is no error, 0 (zero) will be assigned.

If a parameter error occurs, a BASIC error (88-□□) will be given.

At this time, parameter values are not guaranteed.

Error codes

\$81 form\$ image specification error

\$82 Insufficient input data

\$83 Insufficient output data areas

\$84 Detected data out of range

● BASIC error codes

A list of BASIC error codes for the library IFPCNV is shown below.

Error code		Explanation
Error code	Detailed error code	
88	\$71	icmnd parameter error
	\$72	Input parameter error Or parameter following input is not described (insufficient number of parameters).
	\$73	form\$ parameter error Or parameter following form \$ is not described (insufficient number of parameters).
	\$74	Output parameter error Or parameter following output is not described (insufficient number of parameters).
	\$75	ierr parameter error Or parameter following ierr is not described (insufficient number of parameters).

TC030110.EPS

Notice for programming

- If error code 88 (numeric value assigned to the ERRC function) in library IFPCNV occurs, all error messages are related to the library. If an error code 88 occurs, judge error details by checking detailed error codes (numeric values assigned to ERRC function, expressed in hexadecimal) and the table above.
- In library IFPCNV, though no BASIC errors occur, other errors may have occurred. In this case, error codes are returned to ierr parameter of this library. Therefore, it is necessary to program so that the value of parameter ierr is referred to just after this library is executed.

IMAGE

● Statement



Function: Defines formats for representing I/O.

Format: IMAGE $\underline{\hspace{1cm}}$ image-specification

Image-specification: Format character string; need not be enclosed in quotation marks.

Explanation: I/O statements listed below can define I/O formats.

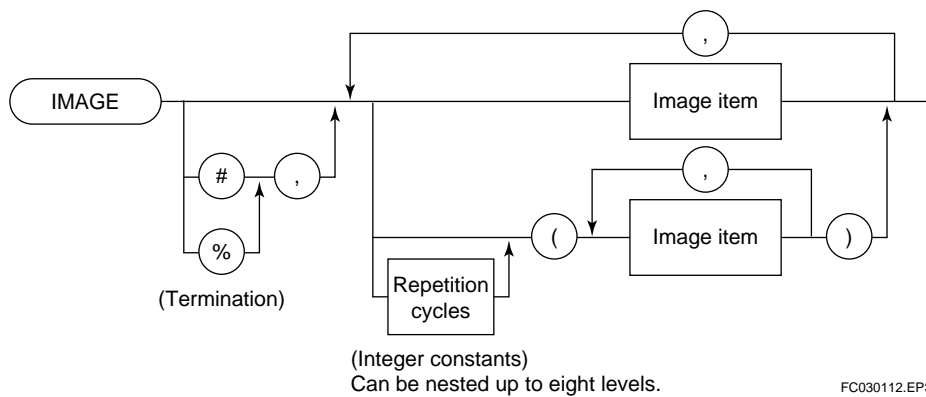
PRINT USING, DISP USING, OUTPUT...USING, ENTER...USING

For these I/O statements, format may be specified in each statement.

When I/O format specifications are complex, or when the same format is used repeatedly, the IMAGE statement allows you to define I/O formats. Format specification image parameters – output and input image specifications – are explained below. (In addition, not so far as any remark is described, when image specifications are defined in each I/O statement, image parameters are used in the same way for each output or input image specification. However, in this ease, quotation marks (") are necessary before and after image specification.) Multiple statements on the same line separated by colons cannot be used (comments using an REM statement cannot be described with an IMAGE statement either.)

In addition, negative numbers are not permitted for repeated designation of [n]. Also, 0's (zero) are all regarded as 1.

(1) Output image specification



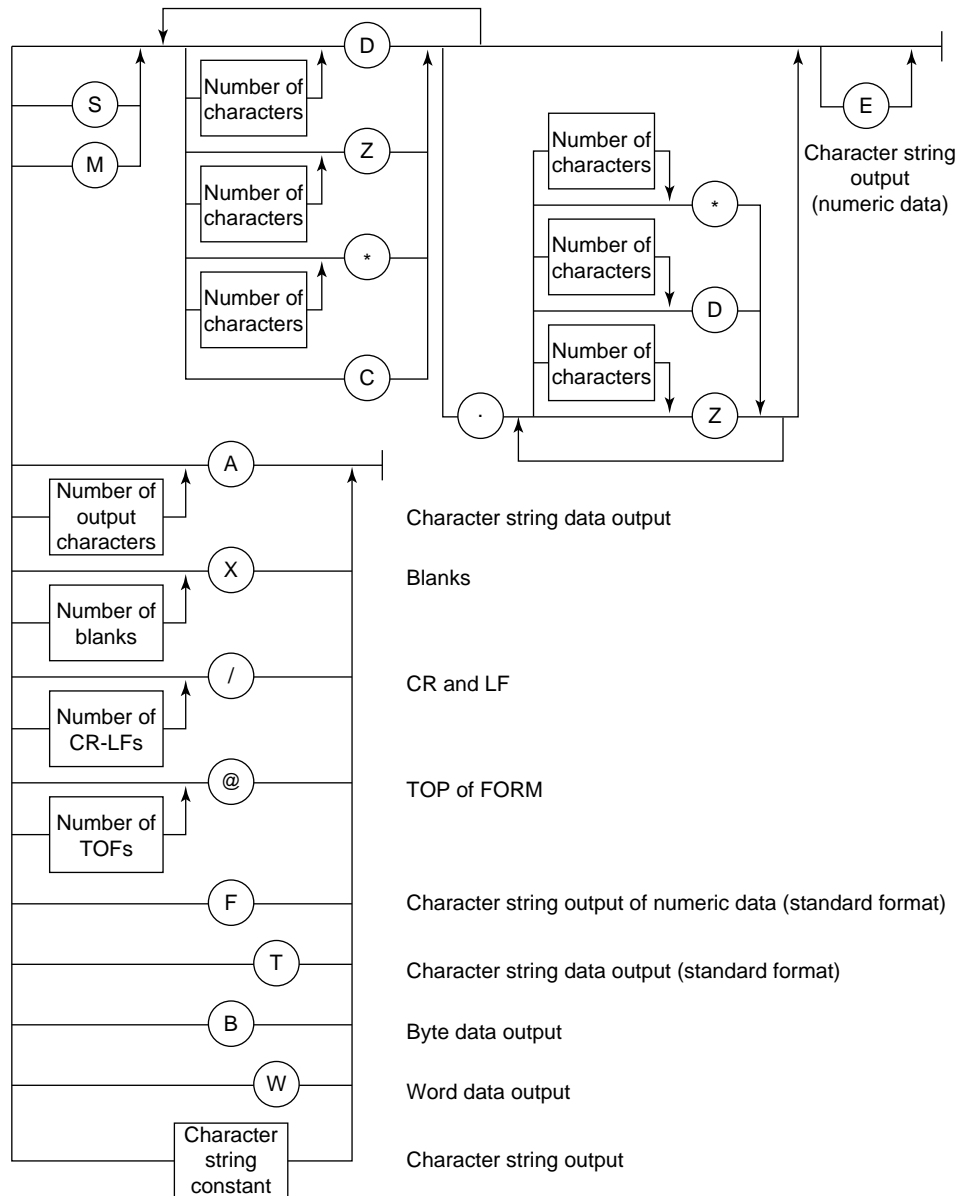
- **Termination**

Specify terminator presence/absence if an output is given to the YEWMAC line computer character panel or the printer connected to the YEWMAC line computer.

If an output is given to a personal computer, CR and LF are always added. Specifying # or % is ignored.

A terminator for output from a serial communication module is set using CONTROL statement. For details, see instruction manuals for each module.

- **Output statement image items**



Note: Numeric value representation is allowed up to 64 digits.

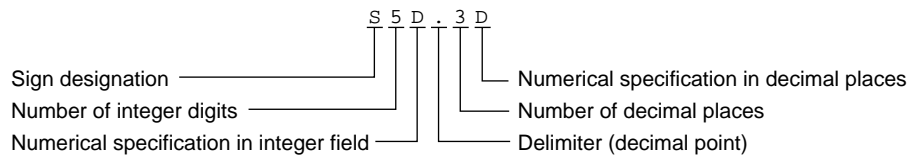
Example:

	Normal	Error
nD	64D	65D
nD.D	62D.D	63D.D
D.nD	D.62D	D.63D
nD.nD	30D.33D	31D.33D

FC030113.EPS

- **Character string output (numeric data)**

(Example)



FC030114.EPS

Sign designation

- S: Sign output designation in upper-case alphabetic character; “+” for positive values, “-” for negative values
- M: Sign output designation in upper-case alphabetic character; blank for positive values, “-” for negative values

Numeric specification

- [n] D: Sets the position of n digits; when the number of digits is less than the number of specified digits, the output is right-justified with zero suppression and leading zeros replaced with blanks in the integer part. Use the upper-case alphabetic character for D.
- Therefore, nothing is displayed for 0. Equals to Z specification in the decimal part.
- [n] Z: Sets the position of n digits; When the number of digits is less than the number of specified digits, the output is right-justified in the integer part and leading zeros are output as they are. Use the upper-case alphabetic character for Z.
- [n] *: Sets the position of n digits; When the number of digits is less than the number of specified digits, the output is right-justified in the integer part and leading zeros are replaced by *.
- E: The numeric field containing this symbol is output in a floating point format. An exponent sign (+ or -) and a three-digit exponent are output for the E specification. (The E symbol must be preceded by at least one symbol (D, Z or *) representing a numeric value.) Use the upper-case alphabetic character for E.

Delimiter

- .: A period (.) is output at this position as a delimiter (decimal point).
- C: A comma (,) is output at this position as a delimiter of integer part. Use the upper-case alphabetic character for C.
- However, if the numeric value is less than the number of integer part digits, the comma is omitted.

(Example)

Numeric data	Image specification	Output format
123.45	S5D	┐┐ +123
123.45	M5D	┐┐┐ 123
123.45	S5Z	+00123
123.45	S5*	***123
123.45	S5DE	+12345E-002
123.45	S5D.3D	┐┐ +123.450
123.45	S5Z.3Z	+00123.450
123.45	S3DC2D.3D	┐┐ +1,23.450
123.45	S2DC3D.3D	┐┐┐ +123.450
123.45	S3DC2D.3DE	+123,45.000E-002

TC030111.EPS

- **Character string data, character strings, blanks are output.**

[n] A: n character are output from the corresponding output character string (a large character corresponds to 2A. Thus, if the n-th character is a large character, that character is not output but a space is output).

If the corresponding output character is shorter than the specified one, spaces are added. Use the upper-case alphabetic character for A.

[n] X: n blanks are output.

Character string constant

Can be inserted enclosing another field specification with a double quotation mark (") to output a character string unaltered. (A character string constant can be used only within an IMAGE statement.)

- **CR and LF, Top of Form**

[n]/: n CR-LFs are output.

[n] @: n Top of Forms are output.

- **Numeric data and character string data are output in a standard format**

F (numeric data) and T (character string data):

Corresponding data items are output in a free-field format, in which numeric items are output in a standard format, leading and trailing zeros are suppressed, and the entire character string is output without blanks before and after character string items.

- **Byte data and word data output**

B: One-byte binary data output, 0 to 255 (any other value becomes 256 MOD); can be used only for numeric constants and numeric variables.

W: Two-byte binary data output, -32,768 to 32,767 (-32,768 or 32,767 is output for any other value); can be used only for numeric constants and numeric variables.

[Remarks]

(i) Repeated symbols (n in the above explanation).

For the described above, integers 1 to 127 can be set.

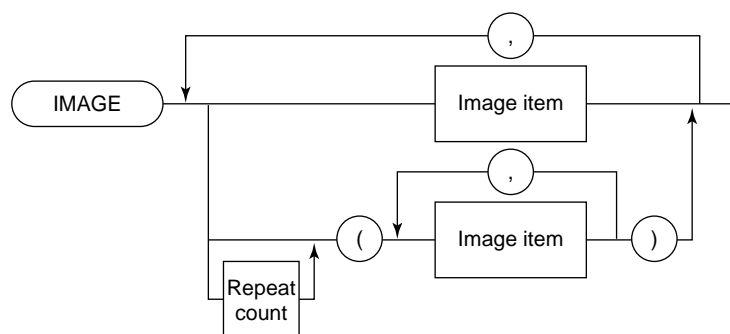
(ii) Reusing formatted character strings.

If a formatted character string ends before the print list is completed, the formatted character string is reused from the beginning.

(iii) Field overflow

When a numeric item overflows the range of a specifier, ?????? is output at the position corresponding to the data.

(2) Input image specification

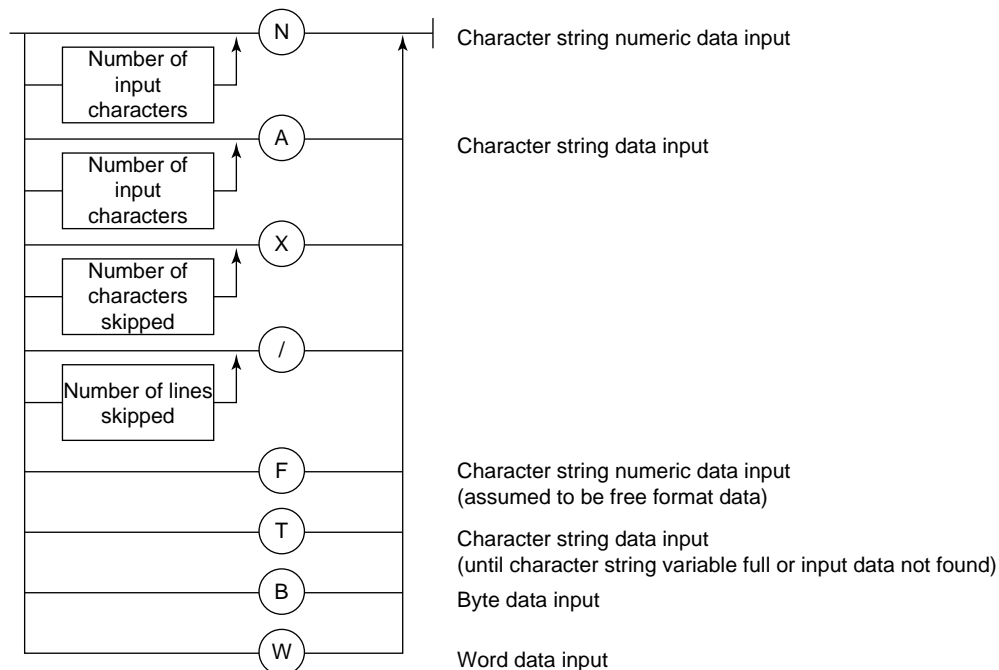


FC030115.EPS

- **Termination**

Entry of CR and LF terminates reading (default). The termination designation can be changed with an EOL (end of line) statement.

- **Input statement image items**



FC030116.EPS

- [n] N: All the digits including sign and delimiter in one numeric data item is set to [n]. A period is assumed to be the delimiter between the integer and the fraction parts.
- [n] A: n characters are input to a character string variable.
- [n] X: n characters are skipped.
- [n] /: All characters are skipped until the n-th CR-LF is encountered.
- F: Numeric data is input in a free field format using a decimal point as a delimiter between the integer and the fraction part. Leading spaces are ignored and non-numeric characters after numeric data items are handled as delimiters. Numeric data characters include +, -, E, decimal point, and digits 0 to 9.
- T: Character string data are input in a free-field format. Input to a character string variable is completed when a number of characters equivalent to the defined length of the variable is input (this limit does not apply when the input data is finished).
- B: 1-byte binary input. n [B] is not possible. Can be used only for numeric variables.
- W: 1-word binary input. n [W] is not possible. Can be used only for numeric variables.

Use each upper-case alphabetic character for N, A, X, F, T, B, and W.

INICOMM3

● Library



Function: Initializes (0-clear) the shared register area in its own CPU

Format: [CALLLIB _] INICOMM3

Explanation: Initializes the shared register area in its own CPU.

This library initializes (0-clear) the shared register area in its own CPU defined by the configuration. When the shared register area is not configured, the initialization is not performed.

INIT COM

● Statement



Function: The INIT COM statement initializes the common variable area in the relevant program area.

Format: INIT _ COM

Explanation: Numeric variables are initialized to 0 and character variables are initialized to null.

Common areas cannot be initialized with command NEW used for program area initialization.

INSTR

● Function



Function: Searches for an arbitrary character string in a character string, and returns the position at which the character string begins.

Format: INSTR(c1 , c2 [, n])

c1: Character string expression.

c2: Character string expression.

n: Numeric expression.

Explanation: This function searches for the character string represented by c2, starting from the n-th position from the head of the character string represented by c1 and returns a numeric value corresponding to the position of the beginning of the character string c2. If n is omitted, the character string c2 is searched for from the beginning of the character string c1.

As the position, a numeric value counted from the head of c1 which is assumed as 1 is returned. When the character string c2 is not found, the function returns the numeric value 0.

Both a standard and a large character are handled as one character. If a character string is to be handled on the standard character basis, use HINSTR function.

INT

● Function



Function: Returns an integer part of x.

Format: INT(x)

x: Numeric expression.

Explanation: This function returns the largest integer less than or equal to x (i.e., by rounding off decimal places).

IOSIZE

● Function



Function: Returns the number of bytes transferred by executing the latest I/O statement.

Format: IOSIZE

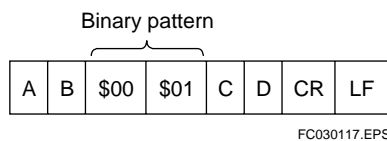
Explanation: Statements set are ENTER, OUTPUT.

ENTER and OUTPUT statements used with TRANSFER statement cannot be included.

Differences between IOSIZE and BLEN statements

A BLEN handles data in bytes and ends counting of data when the null code is found. So when a binary I/O pattern matches the null code, real number of bytes may not be returned. The IOSIZE can return real number of bytes even if the binary I/O pattern matches the null code (see below).

Example: When the following data is transferred:



BLEN returns two bytes.

IOSIZE returns eight bytes.

LASTBIT

● Function



Function: Returns the last bit removed from a word as a result of a shift or rotation.

Format: LASTBIT

Explanation: This function deals with bits. This function returns the bit value of 0 or 1, the last bit removed from a word by the SHIFT, LSHIFT or ROTATE, LROTATE functions.

LB CD

● Function



Function: Returns a BCD double-length integer.

Format: LB CD(x)

x: Numeric expression.

Explanation: This function converts a numeric value or variable (decimal number) represented by “x” in the long integer type into a BCD integer and returns it.

“x” must be an integer with a maximum of eight digits. When a numeric value or variable contains digits after the decimal point, the numeric value – with any digits after the decimal point rounded off – is converted into a BCD number.

LBINAND

● Function



Function: Returns the result of ANDing bit by bit.

Format: LBINAND(m , n)

m , n: Numeric expression.

Explanation: This function deals with bits in the long integer type.

Numeric values and variables represented by m and n are ANDed bit by bit.

Example : A=LBINAND(\$FF000000 , \$FFFFFFFF)

The value of A is FF000000.

The object is long integers only.

LBINNOT

● Function



Function: Returns one's complement of m.

Format: LBINNOT(m)

m: Numeric expression.

Explanation: This function deals with bits in a long integer type

One's complement of the numeric value or variable represented by m.

Example : A=LBINNOT(\$0F0000FF)

The value of A is \$F0FFFF00.

The object is long integers only.

LBINOR

● Function



Function: Returns m and n ORed bit by bit.

Format: LBINOR(m , n)

m , n: Numeric expression.

Explanation: This function deals with bits in a long integer type.

Numeric values or variables represented by m and n are ORed bit by bit – see example below.

Example : A=LBINOR(\$FF0000F0 , \$00000000)

The value of A is \$FF0000F0.

The object is long integers only.

LBINXOR

● Function



Function: Returns m and n exclusively ORed bit by bit.

Format: LBINXOR(m , n)

m , n: Numeric expression.

Explanation: This function deals with bits in a long integer type.

Numeric values or variables represented by m and n are exclusively ORed.

Example : A=LBINXOR(\$FF0000F0 , \$FFFFFFFF)

The value of A is \$00FFFF0F.

The object is long integers only.

LBIT

● Function



Function: Returns the bit of a specified bit position.

Format: LBIT(m , n)

m: Numeric expression.

n: (1) Integer type expression.

(2) Character string or character string variable representing binary pattern.

Explanation: This function deals with bits in the long integer type and the long integers only.

When n=integer type expression (1) above :

Returns a bit value of the n-th digit (starting with the least significant digit 0) for binary expressions for m (numeric value or variable).

Example : A=LBIT(\$01000010,4)

Bit value for A is 1 in this example.

When n=chracter string or character string variable representing binary pattern (2) above:

When a pattern of binary expression for m (numeric expression) coincides with n (character string expression) representing binary pattern, 1 is returned; otherwise, 0 is returned.

For a bit which is not to be compared, set a capital X in the bit position corresponding to the character string (or character string variable) – see example below.

Example : A=LBIT(\$01000010 , "00000001000000000XXXX00000010000")

In this example, the value of A is 1.

LCOPY

● Command
● Subcommand



Function: Copies a statement in a user area line-by-line.

Format: (1) LCOPY source-line-number, destination-line-number
(2) LCOPY source-line-number-1 – source-line-number-2, destination-line-number

Explanation: The LCOPY command is used as a command (in command entry mode) or as a subcommand (used in an editor).

(1) When this command is used in format (1) above:

A source line specified by a source-line-number is copied to a destination-line-number. The statement with the source-line-number remains unchanged. If the destination-line-number already exists, the original statement is erased and only the copied statement remains.

(2) When LCOPY is used in format (2) above:

Line spaces remain in each line of the source line numbers to create a copy of each line of the source-line-number-1 to source-line-number-2 after the destination line numbers. If the source-line-number-1 is not equal or smaller than the source-line-number-2, an error occurs.

If the destination line numbers of some or all copied lines coincide with the line numbers of existing lines, these copied lines will replace the existing lines.

LEFT\$

● Function



Function: Returns a substring of specified length, starting from the leftmost character.

Format: LEFT\$(c , m)

c: Character string expression.

m: Numeric expression.

Explanation: This function is used to isolate a specific number (m) of string characters starting from the leftmost character in the string.

Both standard and large characters are handled as each one character.

When “m” is greater than the number of all characters in the character string, the entire character string (c) is returned.

If character strings are handled on the standard character basis, use HLEFT\$ function.

LEN

● Function



Function: Returns the number of characters in a character string c.

Format: LEN(c)

c: Character string expression.

Explanation: A space is counted as one character. When characters are counted in byte units, use the BLEN function. When characters are counted on the standard character basis, use HLEN function.

Both standard and large characters are handled as each one character.

LET

● Statement



Function: Assigns the result of computation of the expression on the right side to the variable on the left side.

Format: (1) LET variable name=expression
(2) Variable name=expression

Variable: Name of a numeric variable or character string variable.

Expression: Numeric expression or character string expression.

LET can be omitted.

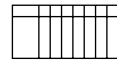
Explanation: When the right side contains a numeric expression, the left side must be a numeric variable; when the right side contains a character string expression, the left side must be a character string variable.

When the size of a character string variable is defined by a DIM statement, the number of characters is limited only by the number of characters defined by this DIM statement. Only the plus sign (+) can be used in character string expression. It denotes a concatenation of character strings.

When an integer-type numeric variable is assigned, the result of computation of an expression on the right side is rounded off to an integer value for assignment.

LHEX\$

● Function



Function: Converts a decimal number to a hexadecimal number and returns the character string.

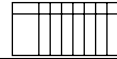
Format: LHEX\$(x)

x: Numeric expression (−2147483648 to 2147483647)

Explanation: Returns a character string consisting of a hexadecimal number (converted from a numeric expression represented by x in a long integer type). Digits after the decimal point included in the value of x are rounded off. If the hexadecimal number is over eight digits (the value of x is outside the range −2147483648 to 2147483647), an error occurs.

LINKLIB

● Command



Function: Loads a library file into user area.

Format: LINKLIB program name

Program-name (character string): [device-specification:] [path \] file-name [.extension]

When [device-specification] is omitted, the current volume is specified.

Extension cannot be omitted as far as it is used.

Explanation: Loads a library file into the user area. Programs in the user area are not affected by the LINKLIB command.

If the identical library name or subprogram already exists in the user area, an error occurs. When a library is replaced with another library having the same name, delete the existing library with a DEL command and load the new library file into the user area with LINKLIB command.

User intermediate language format to save user programs together with libraries. If user programs are saved in source form, they are saved without libraries. For intermediate format, refer to the SAVE command.

When a library is in the user area, this is shown by the LIST PROG command (see the LIST command). Some libraries do not require the LINKLIB command to load them. For such libraries, if the LINKLIB command is executed, an error occurs. For more information, see YM-BASIC/FA REFERENCE.

LIST (L)

● Command



Function: Outputs a specified range of program lines in user area to the display of a personal computer or to a printer. Press and hold [CTRL] key and then press [P] key. This action will output to both the display of a personal computer and to a printer.

An error message is also generated and output if any syntax errors are detected in the program.

Format:

LIST $\left[\begin{array}{l} \text{Start-line-number [, end-line-number]} \\ \text{All} \\ \text{PROG} \end{array} \right]$

FC030118.EPS

LIST may be abbreviated as L.

Explanation: The ranges of program listing associated with the line number specifications are as follows:

Format	Description
LIST	Outputs all lines of the program block specified in the PROG Command.
LIST_ start-line-number	Outputs only the line with the start-line-number of the program block specified in the PROG command.
LIST_ start-line-number, End-line-number	Outputs the lines from start-line-number to the end-line-number of the program block that is specified in the PROG command.
LIST_ ALL	Outputs all program lines in program blocks in the user area.
LIST_ PROG	Outputs program names, subprogram names and library names in the use area.

TC030112.EPS

To pause or abort listing:

- Press [CTRL] key and [S] character key together

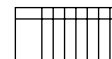
Suspends program listing. Pressing any key e.g., space key restarts the output of listing.

- [ESC] key

Aborts program listing.

LOAD

● Command



Function: Loads a program stored in auxiliary memory into the current user area.

Format: LOAD _program-name

Program-name (character string): [device-specification:] [path \] file-name [.extension]

When [device-specification] is omitted, the current drive is specified. Extension cannot be omitted as far as it is used.

Explanation: In non-resident mode, LOAD command deletes all programs in the user area that is now open, then loads the program specified by the program name stored in auxiliary memory to the user area.

When there are any resident programs, an error will occur. To delete the resident program then load a new program, change the resident program to non-resident (refer to SETMDRES command and NEW command.).

For the program name, refer to the SAVE statement.



CAUTION

Program LOAD time is depended on the program size.

When the user program size is 120K bytes, it takes more than three minutes in 9600bps.

During the execution, "Downloading now" message appears.

LOG

● Function



Function: Returns the natural logarithm.

Format: LOG(x)

x: Numeric expression (x is greater than 0).

Explanation: This function returns the natural logarithm, whose base is natural number e, represented by x (numeric value or variable).

x must be a positive number.

LROTATE

● Function



Function: Rotates (shifts) bits.

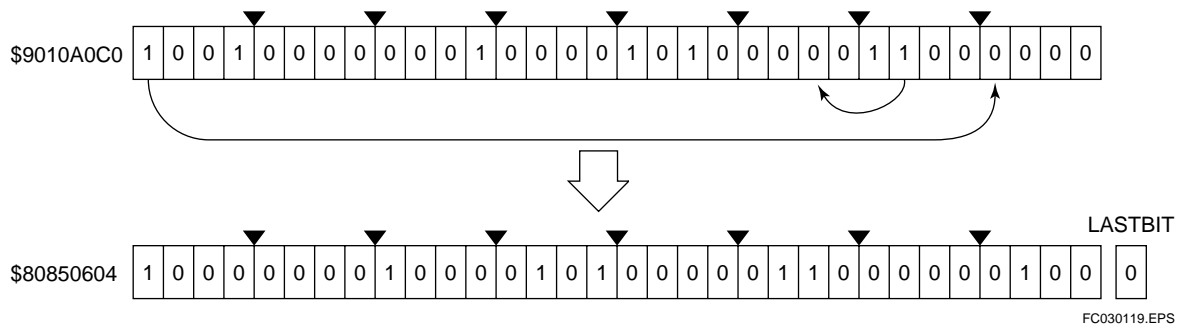
Format: LROTATE(m , n)

m , n: Numeric expression (integer).

Explanation: This function deals with bits in long integer type. This function is used to rotate (shift) bits represented by m (numeric value or variable) by n bits. When n is positive, the bits are rotated right. When n is negative, the bits are rotated left.

Last bit removed from a bit string is assigned to the LASTBIT function. When n is 0, rotation is not performed and the value of the LASTBIT function is 0.

Example: For LROTATE(\$9010A0C0 , -3)



LSHIFT

● Function



Function: Shifts bits.

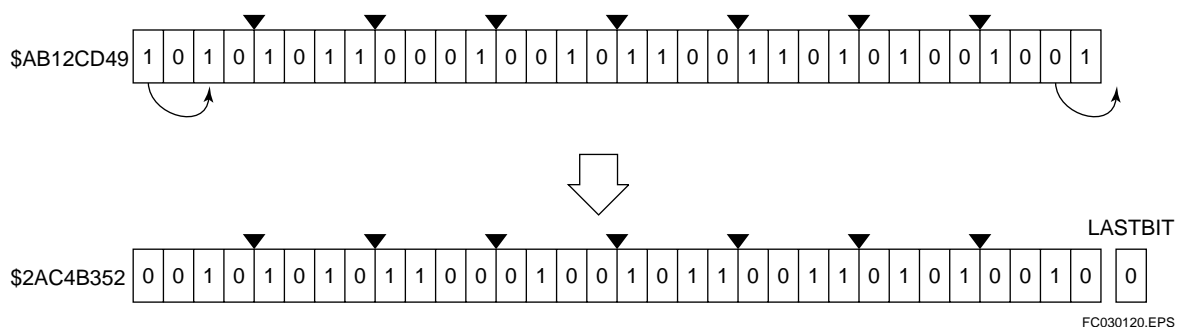
Format: LSHIFT(m , n)

m , n: Numeric expression (integer type).

Explanation: This function deals with bits in long integer type. This function is used to shift bits represented by m (numeric value or variable) by n bits. When n is positive, the bits are shifted right. When n is negative, the bits are shifted left. Last bit removed from a bit string is assigned to the LASTBIT function.

When n is 0, the shift is not performed and the value of the LASTBIT function is 0.

Example: For LSHIFT(\$AB12CD49 , 2)



MERGE

● Command



Function: Merges a program stored in auxiliary storage with the user program area.

Format: MERGE progra-name

Program-name (character string): [device-specification:] [path \] file-name [.extension]

When [device-specification] is omitted, the current drive is specified. Extension cannot be omitted as far as it is used.

Explanation: Merges a program in auxiliary storage with the current program block (refer to the PROG command).

The MERGE command performs substitution or insertion of line numbers by checking them against the line numbers of the program in main memory.

Accordingly, if a statement with the same line number as that of an existing statement in the program area is entered, the statement in the program area is replaced with the new statement. To preserve the program in the program area intact, therefore, it is necessary to eliminate duplicated line numbers by using a RENUM command, or the like. The program to be merged must be in source file format. Programs in intermediate language cannot be merged.

For the source file format and intermediate language file format, see the SAVE command.



CAUTION

If programs in the auxiliary storage contain a SUB statement, MERGE may not work correctly. Use an APPEND command in this case.

MID\$

● Function



Function: Returns a character string extracted from character string c.

Format: MID\$(c , m [, n])

c: Character string expression.

m , n: Numeric expression.

Explanation: This function returns a character string composed of n characters extracted from the m-th character at the left side of character string c. When n is omitted, the character string from the m-th character to the end of character string c is extracted.

MOD

● Function



Function: Returns the remainder after integer division.

Format: MOD(x , y)

x , y: Numeric expression.

Explanation: Calculates INT(x)/INT(y) and returns a remainder.

Integers, long integers, single-precision real numbers, and double-precision real numbers can be used. For INT(x), see INT function.

MOVE

● Statement



Function: Moves all the data in an array to another array.

Format: MOVE $\underline{\hspace{1cm}}$ array-name-1(*) , array-name-2(*)

Explanation: All the data in the array identified by array-name-1 is moved to the array identified by array-name-2. The two arrays must have the same variable type (numeric or character string).

The arrays may not necessarily have the same array size. If the arrays do not have the same array size, move data on the basis of the array having the smaller array size.

The data in the array identified by array-name-1(*) is unaffected.

The array data size which MOVE statement can copy is 64KB or less.



CAUTION

This statement cannot be executed if a different array name is declared in the same common area by RECOM statement.

NAM

● Function



Function: Returns the value of the variable name in character string c.

Format: NAM(c)

c: Character string expression.

Explanation: A quotation mark (") is not necessary. Numeric variable name and character string variable name can be assigned to character string c.

This function enables a simple variable to be designated indirectly. In the case of array variable, use the ARNAM function.

Either numeric variables or character string variables can be used as variables.

NEW

● Command



Function: Initializes a user program area.

Format: NEW

Explanation: Initializes user program area and prepares it to store new programs. Resident programs are also initialized. However, common area is not initialized (see the INIT COM statement).

NEXT

● Statement



Function: Defines the end of a program range with FOR statement.

Format: NEXT $\underline{\hspace{1cm}}$ simple-numeric-variable

Explanation: See FOR – NEXT statement.

ON EOT/OFF EOT

● Statement



Function: The ON EOT statement declares branching to the specified processing when data transmission by the TRANSFER statement is completed. The OFF EOT statement resets the declaration of the ON EOT statement.

Format:

(1) ON — EOT — slot-number [, port-number] — $\left\{ \begin{array}{l} \text{GOSUB — } \left\{ \begin{array}{l} \text{label} \\ \text{line-number} \end{array} \right\} \\ \text{CALL — subprogram-name} \\ \text{GOTO — } \left\{ \begin{array}{l} \text{label} \\ \text{line-number} \end{array} \right\} \end{array} \right\}$

(2) OFF — EOT — slot-number [, port-number]

FC030121.EPS

Slot-number: Numeric expression (integer type).

Port-number: Numeric expression (integer type).

Explanation: The ON EOT statement declares branching to the specified processing when the data transfer to I/O modules (specifically to communication card) by TRANSFER statement is completed.

ON EOT ... CALL ... can be used to branch to a subprogram but no argument can be passed to the subprogram.

The OFF EOT statement cancels the declaration of the ON EOT statement.

For more information, see instruction manuals for each I/O module.



CAUTION

ON EOT...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON ERROR/OFF ERROR

● Statement



Function: The ON ERROR statement is used to prevent some recoverable program execution errors from halting execution by causing branching when an error occurs and suppressing the normal error process. The OFF ERROR statement resets the declaration of the ON ERROR statement.

Format:

(1) ON ERROR { GOSUB {label
line-number}
CALL subprogram-name
GOTO {label
line-number} }

(2) OFF ERROR

FC030122.EPS

Explanation: When an error occurs and the ON ERROR condition has been established, execution is transferred to the specified line. Then the built-in functions ERRL, ERRC and ERRCE can be tested, and error recovery procedures can be executed. These built-in functions return information related to the last error trapped with ON ERROR. ERRL returns the line-number in which the most recent program execution error occurred. ERRC returns the error code of the most recent program execution error.

ERRCE returns the detailed error information for a hardware error.

When a control transfer has been caused by GOSUB or CALL, processing resumes on the line following the line in which the error occurred upon execution of a RETURN or SUBEXIT (or SUBEND) statement after error processing.

RETURN RETRY or SUBEXIT RETRY can be used to return from error processing to retry executing the line affected by an error (hardware error). To transfer control with a GOSUB or CALL, the program execution level is raised to an interrupt level, regardless of the previous execution level and restored on execution of RETURN or SUBEXIT (or SUBEND). For a control transfer with GOTO, the program execution level remains unchanged regardless of the previous execution level. An interrupt is generated immediately when an error occurs in a multiple statement line, and all the remaining statements on the line are ignored.

Without an ON ERROR declaration, program execution is terminated when an error occurs. An error message is output together with the program execution in the debug mode.

An ON ERROR statement declared in a subprogram is cancelled when a SUBEXIT (or SUBEND) statement is executed. The ON ERROR statement is valid only for one program block. When program execution is transferred to other program blocks, the ON ERROR statement should be declared again. If the error-recovery routine branched with ON ERROR GUSUB (or CALL) itself contains an error, program execution is terminated with an error message. When the status information variable has been set by a SET STATUS statement, control transfer does not occur with an ON ERROR statement even if the execution of a “file access statement” or an input/output statement (see SET STATUS statement) causes an error. An ON ERROR CALL... can be used to branch to a subprogram, but no arguments can be passed to the subprogram. An OFF ERROR statement cancels the declaration of the ON ERROR statement.

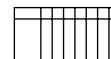


CAUTION

- The ON ERROR statement is valid only in one program block. When control is moved to the other block, the ON ERROR statement must be declared again.
- ON ERROR CALL ... can branch to a subprogram, but no argument can be passed to the subprogram.

ON INT/OFF INT

● Statement



Function: Declares/cancels interruptions by keying in character string.
When interruptions are declared, specify the branch destination for the interrupt.

Format:

(1) ON _ INT _ slot-number [, instrument-number] _ { GOSUB _ { label
line-number }
CALL _ subprogram-name
GOTO _ { label
line-number } }

(2) OFF _ INT _ slot-number [, instrument-number]

FC030123.EPS

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression (integer type).

Explanation: ON INT statement branches the program control to the specified line number or subprogram when an interrupt request from an I/O module is received.

ON INT statement is valid until an OFF INT statement is executed. ON INT ... CALL ... can be used to branch to a subprogram, but no argument can be passed to the program.

An OFF INT statement is used to cancel the declaration of ON INT statement.

For more information, see instruction manual for each I/O module.



CAUTION

ON INT...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON SEQEVTOFF SEQEVTO

● Statement



Function: These statements declare/cancel interruption acceptance owing to event generation in a ladder program.

When interruptions are declared, specify the branch destination for the interrupt.

Format:

(1) ON _ SEQEVTO _ signal-name [, variable-name] _ { GOSUB _ {label
line-number }
CALL _ subprogram-name
GOTO _ {label
line-number } }

(2) OFF _ SEQEVTO _ signal-name

FC030124.EPS

Signal-name: Character string or character string variable of up to 8 bytes representing an event.

Variable name: Variable (integer type).

Explanation: The ON SEQEVTO statement declares branching to the specified BASIC program processing corresponding to a signal name when the signal is received from the ladder sequence program (generated by interrupt only application command to BASIC).

Interrupt to BASIC program from the ladder sequence program is informed after execution of ladder sequence program command.

Specify a variable name to receive data from the ladder program.

Only integer type values are valid.

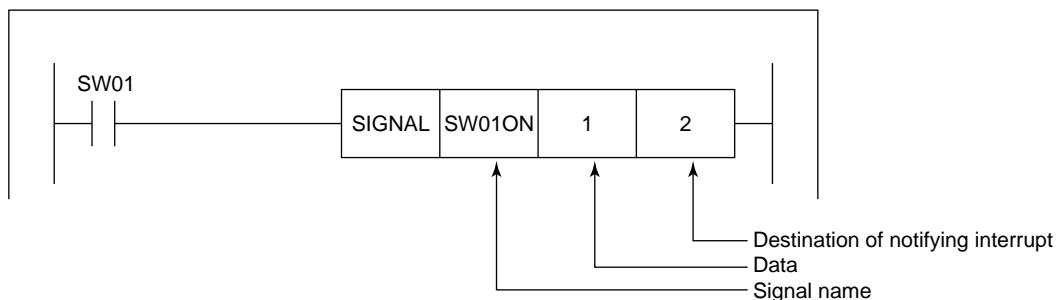
The OFF SEQEVTO statement cancels the ON SEQEVTO declaration.

BASIC program

```
10 DEFINT D
100 ON SEQEVTO "SW01ON",DATA GOSUB A@
```

Signal transmission

Ladder program



FC030125.EPS

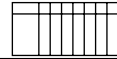


CAUTION

ON SEQEVTO...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON TIME/OFF TIME

● Statement



Function: The ON TIME statement declares interrupts at each interval at the designated period at or following the specified time and designates the branching destination.

The OFF TIME statement resets the branch at a specified time.

Format:

(1) ON TIME #time-number , time [, time-interval] { GOSUB {label
line-number}
CALL subprogram-name
GOTO {label
line-number} }

(2) OFF TIME #time-number

FC030126.EPS

Timer-number: Specify any of 1 to 8. A numeric value or variable (integer) is set.

Time: Specifies time as a character string expression in the format "hh:mm:ss" (24-hour system).

Time-interval: Specifies the interval (in seconds) when a control transfer occurs after the time specified as above. Numeric value or variable. (omissible)

Explanation: When the specified time is reached, control passes to a specified line number, to the subroutine starting at a specified line number, or to the specified subprogram if the control transfer is caused by GOTO, GOSUB, or CALL respectively.

After a control transfer caused by GOSUB or CALL, control returns to the original execution flow on completion of RETURN or SUBEXIT respectively.

If a control transfer is caused by GOSUB or CALL, the program execution level is raised to an interrupt level; if the control transfer is caused by GOTO, the program execution level is unaffected and only the execution flow is altered.

A branch caused by the ON TIME statement is valid only once. For example, if a branch is required every day at a specific time, first branch and then execute an ON TIME statement or set the time-interval to 86400 seconds (24 hours). The timer numbers can be used independent of those in ON TIMER statement. Therefore, it is not necessary to consider duplication of timer numbers in ON TIME and ON TIMER statements. ON TIME ... CALL ... can be used to branch to a subprogram, but no arguments can be passed to the subprogram.



CAUTION

ON TIME...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON TIMEOUT/OFF TIMEOUT

● Statement



Function: Declares and releases interrupt when an I/O operation is not terminated within the specified time.

Specifies the branch destination when an interrupt occurs.

Format:

(1) ON _ TIMEOUT _ slot-number [, instrument-number] _ { GOSUB _ { label
line-number }
CALL _ subprogram-name
GOTO _ { label
line-number } }

(2) OFF _ TIMEOUT _ slot-number [, instrument-number]

FC030127.EPS

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression (integer type).

Explanation: The ON TIMEOUT statement declares interrupt when an I/O operation specified by SET TIMEOUT statement is not terminated within the specified time. An ON TIMEOUT statement is valid until OFF TIMEOUT statement is executed. ON TIMEOUT ... CALL ... can be used to branch to a subprogram, but no arguments can be passed to the subprogram. An OFF TIMEOUT statement is used to cancel the declaration of the ON TIMEOUT statement.

For details, see instruction manuals for each I/O module.



CAUTION

ON TIMEOUT...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON TIMER/OFF TIMER

- Statement



Function: Declares or releases interrupt after a specified time interval elapses.

When interrupts are declared, specify the branch destination for the interrupt.

Format:

(1) ON _TIMER _ #timer-number, time-interval _ {
GOSUB _ {label
line-number}
CALL _ subprogram-name
GOTO _ {label
line-number}}

(2) OFF _TIMER _#timer-number

Timer-number: Specify any of 1 to 8. A numeric expression.

Time-interval: The time interval (in ms) to the time when control transfers to the specified line. 1 to 268435455. A numeric expression.

FC030128.EPS

Explanation: The ON TIMER statement declares interrupt and branch destination after a specified time interval elapses.

Once a timer is activated by an ON TIMER statement, a control transfer occurs at the specified time interval until the timer is deactivated by an OFF TIMER statement. The time interval can be set in ms (1/1000 sec). Its range is 1 to 604800000 (7 days).

If a branch is caused by GOSUB or CALL statement, control is returned to the original program execution flow by RETURN statement or SUBEXIT (or SUBEND) statement respectively. If a branch is generated by GOSUB or CALL statement, the program execution level is raised to an interrupt level; but for a branch by GOTO statement, the program execution level is unaffected and only the execution flow is altered.

The timer numbers can be used independent of those in ON TIME statement. Therefore, it is not necessary to consider duplication of timer numbers in ON TIME and ON TIMER statements.

ON TIMER ... CALL ... can be used to branch to a subprogram, but no argument can be passed to the subprogram. An OFF TIMER statement is used to reset the ON TIMER statement declaration.



CAUTION

ON TIMER...CALL... can be used to branch to a subprogram, but no argument can be passed to the subprogram.

ON ... GOSUB

● Statement



Function: Passes control to one of many subroutines based on the results of the computation of a numeric expression.

Format: ON numeric-variable GOSUB {label
line-number-1}, {label
line-number-m}...

FC030129.EPS

Explanation: The value of a numeric-variable is rounded off to an integer following JIS standard. Control passes to the subroutine starting at line-number-1 if the integer is 1, or to the subroutine with line-number-m if the integer is m.

After RETURN statement is executed, the statement next to ON GOSUB statement is executed.

The numeric variable rounded to an integer is less than 1 or greater than the last line number, the statement next to ON GOSUB statement is also executed.

ON ... GOTO

● Statement



Function: The ON GOTO statement passes control to a line number or label based on the results of the computation of a numeric expression.

Format: ON numeric-variable GOTO {label
line-number-1}, {label
line-number-m}...

FC030130.EPS

Explanation: The value of a numeric variable is rounded off to an integer following JIS standard. Control passes to the line with line-number-1 if the integer is 1, or to the line with line-number-m if the integer is m.

The statement next to the ON GOTO statement is executed when the value of the numeric variable rounded off to an integer is less than 1 or greater than m which corresponds to the last specified line number.

OPTION BASE

● Statement



Function: Specifies the starting number (lower bound) of array subscripts.

Format: OPTION BASE subscript-start-number

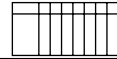
Subscript-start-number: Specify either 0 or 1.

Explanation: Subscripts start with 0 unless otherwise specified by an OPTION BASE statement. The OPTION BASE statement must be specified at only one position in the program before any array variables are used.

After the execution of the OPTION BASE 1 statement, if 0 is used for the element of array variable, an error occurs.

OUTPUT

● Statement



Function: Outputs data to an I/O module, an I/O buffer or a sequence device.

Format:

- (1) OUTPUT slot-number [, instrument-number] $\left[\begin{array}{l} \text{USING } \left\{ \begin{array}{l} \text{image-specification} \\ \text{line-number} \end{array} \right\} \\ \text{BFORMAT} \\ \text{NOFORMAT} \end{array} \right]$
; output-variable
- (2) OUTPUT output-buffer-name; output-variable
- (3) OUTPUT slot-number, device-name-character-string-expression [NOFORMAT]
; output-variable
- (4) OUTPUT slot-number, terminal-number $\left[\begin{array}{l} \text{NOFORMAT} \\ \text{BFORMAT} \end{array} \right]$; output-variable

FC030131.EPS

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression. Omitting possibility depends on the I/O modules.

Terminal-number: Numeric expression (integer type)

Line-number: IMAGE statement line-number. Numeric value.

Image-specification: Character string (see IMAGE statement).

Output-variable: Variable-name [{ ; } variable-name ...] (numeric value or variable or character string or character string variable)

Areas in which the numeric values or character strings are to be displayed are predetermined so that each line is divided into 8 characters and the next value or character string is output, from the beginning of the next area when a comma is used as a delimiter, and subsequent to the preceding output when a semicolon is used as a delimiter.

Output-buffer-name: Simple variable of array variable (numeric variable or character string variable). However, common variables cannot be used.

Device-name-character-string-expression : Character string expression representing a sequence device name.

Explanation: Format (1) or (2) outputs data in the complete-return type to an I/O module other than the multiple transmission module or the multiple transmission module respectively.

Format (3) is used in pair with TRANSFER statement.

If the communication rate (baud rate) is low mainly in serial communication channels (RS-232-C, etc.), the time to OUTPUT execution completion may be long. In such a case, use format (3) associated with TRANSFER statement in a pair.

If an output is to be given to a communication channel I/O module using a character string expression, that string, to immediately before the null code, is identified as an effective string. The null code cannot be output.

Presence of format specification such as NOFORMAT and operation in that condition vary depending on the I/O module. For details, see instruction manual for each I/O module.

Format (4) is used for outputting data to the sequence device.

If a status information variable is specified by a SET STATUS statement, and the execution of the OUTPUT statement causes an error, the corresponding error code is stored in the status information variable. In this case, no system error occurs.

PAUSE

● Statement



Function: Suspends the execution of a program temporarily.

Format: PAUSE

Explanation: The PAUSE statement is used anywhere in a program to suspend program execution temporarily. However, this statement is valid only in debug mode. PAUSE statement is invalid in real mode. Execution of a PAUSE statement in the debug mode suspends program execution temporarily, displays the following message:

PAUSE XXXXXXXX Line = XXXXX
 └──────────┘ └──────────┘
 Line number of the line next to the PAUSE statement
 Program block name

FC030132.EPS

Program block names that are displayed are as follows:

	For Programs Input from Keyboard	For Programs Input from Auxiliary Memory
Main program	*****	Program Name
Subprogram	Subprogram Name	Subprogram Name

TC030113.EPS

When the program execution is suspended, a variable value can be output, the suspended program execution can be resumed with a CONT command after resetting variable values, or the next one line can be executed with a STEP command. However, program execution suspended by the PAUSE statement cannot be resumed with the CONT command, and execution of the next line with a STEP command is disabled, when:

- a program list is changed (including EDIT),
- a program block is changed (using PROG command), or
- program execution is stopped with a STOP statement.

Program execution can also be suspended by the following key operation or TRACEP statement (however, key operation cannot suspend a program at its specified line).

For personal computer: Press [ESC] key.

PI

● Function



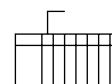
Function: Returns the circular constant pi (π).

Format: PI

Explanation: 3.141592653589793 is returned.

PRINT (PR)

● Statement



Function: Outputs a print list to a personal computer display.

Format: PRINT  print-list

Print-list: Variable names, array variable names, numeric values, or character string, can be specified. Each item should be delimited with comma (,) or semicolon (;). Each item length should be up to 1024 bytes.

PRINT may be abbreviated as PR.

Explanation: PRINT statement is valid in the following cases.

	In debug mode	In real mode
Personal computer	Valid	Invalid (PRINT statement is ignored.)

TC030114.EPS

(1) Output items

(a) Numeric expression

The value of a numeric expression is output in decimal. The numeric value range and representation is given below. When the output data is positive, a blank is indicated instead of a plus "+" sign.

Representation	Example	Range
Indication without a decimal point	100 -5798	Up to 16 characters, including sign
Decimal point indication	1.2345 -1234.567	Up to 16 characters, including sign and decimal point
When none of the above representation methods apply (exponential indication)	-1.23E-010	Up to 16 characters, including a sign, a decimal point, and an exponent part (5 characters)

TC030115.EPS

(b) Character string expression

The character string generated by a character string expression is output.

(c) Limitation

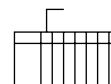
Variables up to 232 bytes can be displayed. For variables of more than 232 bytes, an error occurs.

(2) Delimiter

Carriage return and line feed occurs when the PRINT statement is not terminated by a delimiter.

PRINT USING (PU)

● Statement



Function: Outputs a print list in a specified format.

Format: PRINT USING { image-specification
label
line-number } ; print-list

FC030133.EPS

Image-specification: Format character string constant or character string variable. Put double-quotation marks (") before and after an image-specification.

Label: Label for IMAGE statement.

Line-number: Line number of the IMAGE statement

Print-list: Variable names, arrays, variable names, numeric expressions, or character string expressions, can be specified separated by a comma (,) or a semicolon (;).

PRINT USING can be abbreviated to PU.

Explanation: If an FA-M3 is connected to a personal computer, this statement outputs the list to the personal computer display.

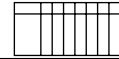
After PRINT USING, specify image-specification or line number (or label) of the IMAGE statement defining image-specification. For designation of image-specification, see IMAGE statement.

Up to 232 bytes can be effective when output from the line controller.

An IMAGE statement must be used to include a literal (text enclosed within quotation marks) in the format character string.

PROG

● Command



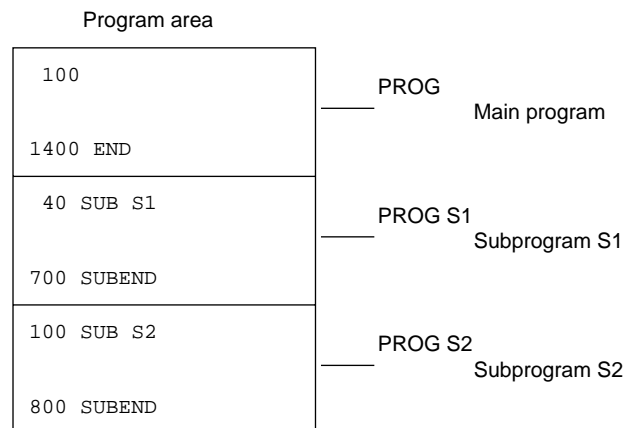
Function: Specifies the program block whose commands are to be executed.

Format: PROG [ subprogram-name]

Explanation: The main program is assumed when subprogram-name is omitted.

Subsequently executed commands apply to the program block specified by the PROG command (called the current program block).

Subprogram-name can be specified only when the named subprogram has already been written. When subprogram name is specified by EDIT command, the current program block is switched automatically.



FC030134.EPS

When subprograms are not used, this command is not needed.

QUIT (Q)

● Subcommand



Function: Quits editor and shifts to the command entry screen.

Format: QUIT or Q

Explanation: Can be used only in editor.

Shifts from the editor panel to the command entry screen.

RANDOMIZE

● Statement



Function: Initializes the random number seed.

Format: RANDOMIZE

Explanation: Initializes the random number sequence generated by a RND function. If a RND function is used to generate random numbers without executing a RANDOMIZE statement, the same random number sequence is generated each time the program is run. The occurrence of the same random number sequence can be prevented by executing a RANDOMIZE statement before generating a random number.

READ

● Statement



Function: Assigns data set by a DATA statement to variables.

Format: READ $\underline{\hspace{1cm}}$ variable-name [, variable-name ,]

Variable-name: Numeric type or character string type.

Explanation: The READ statement is used together with the DATA statement. This statement allows data set by the DATA statement to be assigned to variables on a one-to-one basis. The type of variable in the READ statement must match the type of data in the DATA statement. An error occurs when there is no data to be assigned to the variable specified in the READ statement.

Use RESTORE statement to read repeatedly the data set in the same DATA statement and to specify the DATA statement to be read.

RECOM

● Statement



Function: Specifies the start position of a common variable declared in a COM statement.

Format: (1) RECOM [$\underline{\hspace{1cm}}$ common-variable-name]

(2) RECOM $\underline{\hspace{1cm}}$ #Sn [$\underline{\hspace{1cm}}$ common-variable-name]

n (slot-number) : Numeric value. n=1 to 4.

Explanation: The RECOM statement specifies the start position of a common variable declared in a COM statement that appears later. Common-variable-names specifies as parameters must be defined in a COM statement prior to this statement.

When a common-variable-name is not specified, the start position of the common variable area is assumed.

See COM statement for formats (1) and (2) above.

REM

● Statement



Function: The REM statement is used to enter a comment. It has no effect on program execution.

Format: REM $\underline{\hspace{1cm}}$ character-string

! $\underline{\hspace{1cm}}$ character-string

Explanation: The REM statement is used for program remarks and ignored on the execution of program.

The character-string for REM statement can be any combination of alphanumeric characters and symbols.

For multiple statements, use the REM statement at the last part of a line. If it is used at the beginning or middle part of a line, the following part including colon ":" indicating multiple statement is regarded as a part of character string of the REM statement.

RENUM

- Command
- Subcommand



Function: Renumbers all lines in a program block. Line numbers in statements are also updated automatically.

Format: RENUM [_old-line-number [, new-line-number [, increment]]]

Increment: Positive integer. The default value is 10.

Explanation:

Format	Description
RENUM	Renumbers each line, starting with the currently used smallest line number and incrementing by 10.
RENUM _old-line-number	Renumbers each line, starting with old-line-number and incrementing by 10.
RENUM _old-line-number, new-line-number	Renumbers each line, starting with old-line-number and replacing it with new-line-number and incrementing by 10.
RENUM _old-line-number, new-line-number, increment	Renumbers each line by specified increment, starting with old-line-number and replacing it with new-line-number.

TC030116.EPS

An error message is displayed if old-line-number is greater than new-line-number and a statement exists on a line number smaller than old-line-number.

When FA-M3 is connected, the same error message will be displayed twice for an error of incorrect line number specifications.

Line numbers in statements are updated automatically, but line numbers in TRACE and TRACEP debugging statements are not updated.

Note that the line number in REM statement is not modified either.

RESET

- Statement



Function: Resets the specified I/O module to an initialized status.

Format: RESET _ slot-number [, port-number [, function-number]]

Slot-number: Numeric expression (integer type).

Port-number: Numeric expression (integer type). Omitted if I/O module is reset.

Function-number: Numeric expression. Register numbers.

Explanation: The statement initializes parameters, set in I/O modules I/O buffers, module states, interrupt declaration, etc.

For module reset, all are reset to the state same as start-up state.

The port number is omitted for module reset.

Specify the function number when performing buffer reset, parameter reset, releasing interrupt declaration, etc.

For I/O modules and their contents which can be initialized by RESET statement, see instruction manuals for each I/O module.

RESET STATUS

● Statement



Function: Releases SET STATUS functions.

Format: RESET $\underline{\hspace{1cm}}$ STATUS

Explanation: The RESET STATUS statement releases functions set by the SET STATUS statement. When the execution of a file access statement and an I/O statement causes an error, the SET STATUS statement stores the corresponding error code into the status information variable. Thus, no system error occurs. For more information, see SET STATUS statement.

RESTORE

● Statement



Function: Sets the pointer to the data set in a DATA statement which is to be read by the next READ statement.

Format: RESTORE $\left[\underline{\hspace{1cm}} \left\{ \begin{array}{l} \text{label} \\ \text{line-number} \end{array} \right\} \right]$

FC030135.EPS

Explanation: When a line-number or label is omitted, the pointer is set to the first DATA statement in the program block. When a label or line-number is specified, the pointer is set to the first DATA statement appearing after that label or line specified in the program block.

RETURN

● Statement



Function: Declares the end of a subroutine branched to by the GOSUB statement.

Format: RETURN

Explanation: Declares the end of a subroutine branched to by the GOSUB statement and passes control back to the line following the GOSUB statement. If a RETURN statement is executed, before a GOSUB statement is executed, an error will occur (see GOSUB statement).

RETURN RETRY

● Statement



Function: Declares the end of a subroutine branched by a GOSUB statement and returns control to the line that was being executed when control passed to the subroutine.

Format: RETURN $\underline{\hspace{1cm}}$ RETRY

Explanation: This statement can be used as necessary on return from the subroutine called by ON ERROR GOSUB.

The statement declares the end of a subroutine branched by the GOSUB statement and returns control to the line that was being executed when control passed to the subroutine.

RETURN statement passes control back to the line following the GOSUB statement, while this statement returns control to the head of the very line that was being executed.

This statement is mainly used for retry after an error occurs during I/O access and the program branches by ON ERROR GOSUB statement. When an error occurs at the line of multiple statement in branching by the ON ERROR GOSUB statement and control returns by the RETURN RETRY statement, even if the error occurs at any part of multiple statement, control returns to the head of the line in which the error occurs. An error occurs if a RETURN RETRY statement is executed before execution of a GOSUB statement.

RIGHT\$

● Function



Function: Returns a character string of specified length, starting from the rightmost character.

Format: RIGHT\$(c , m)

c: Character string expression.

m: Numeric expression.

Explanation: This function is used to isolate a specific number (m) of string characters, starting from the rightmost character in the string represented by c. When m is equal to or greater than the number of all characters in character string c, all characters in character string c are returned.

Both a standard character and a large character are handled as one character.

If a character string is to be handled on the standard character basis, use HRIGHT\$ function.

RND

● Function



Function: Returns a pseudo-random number.

Format: RND(x)

x: Numeric expression. ($x \geq 0$)

Explanation: For x more than 0, a pseudo-random number more than 0 and less than x is generated.

For x below 0, a pseudo-random number exceeding x and below 0 is generated.

If 0 is assigned, 0 is always given.

When the value of x is the same, the same random number sequence is generated on each run of this function. When the RND function is executed after executing a RANDOMIZE statement, a different random number sequence is returned.

RNPAR

● Function



Function: Returns parameter (integer) used when the program was initiated.

Format: RNPAR

Explanation: Program initiation parameters are as shown below :

Program Initiation	RNPAR
RUN command	0

TC030117.EPS

If an FA-M3 controller is connected to a personal computer, the program initiation parameters are always 0.

ROTATE

● Function



Function: Rotates (shifts) bits in 16-bit integer.

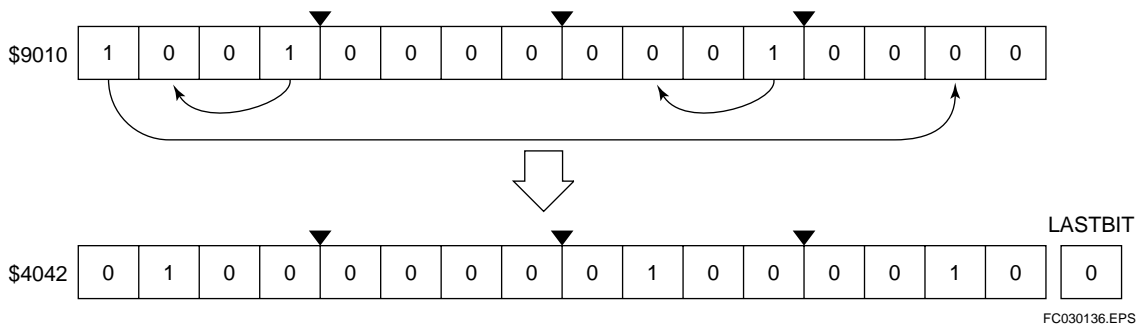
Format: ROTATE(m , n)

m , n: Numeric expression (integer).

Explanation: This function is used to shift bits of a numeric value represented by m in a rotational manner by n bits. When n is positive, the bits are rotated right. When n is negative, the bits are rotated left.

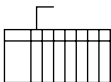
Last bit removed from a bit string is assigned to the LASTBIT function. When n is 0, rotation is not performed and the value of the LASTBIT function is 0.

Example: For ROTATE(\$9010 , -2)



RUN

● Command



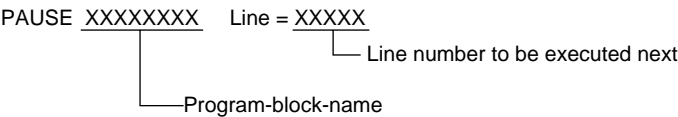
Function: Starts execution of the program stored in a user program area, beginning with its smallest line number.

Format: RUN

Explanation: This command is used only when FA-M3 is connected to a personal computer.

The command executes the program stored in a user program area, beginning with its smallest line number. As the RUN command initializes all work areas before starting program execution, all variable types and values previously defined in the work areas are deleted. However, common variables (defined by the COM statement) are not initialized.

If the [ESC] key is pressed while a program is being executed by a RUN command, the program is suspended after execution of the current line number, and the BASIC system enters the “waiting for command” state after displaying the following message on the screen:



FC030137.EPS

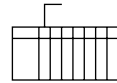
When the program block name is not program-loaded from the disk, ***** will be displayed for the program-block-name.

The CONT command can resume the execution at the suspended line only when neither program editing nor program block change is done (see CONT command).

The program can be executed line-by-line with a STEP command (see the STEP command).

SAVE

● Command



Function: Saves a program from user program area into auxiliary storage.

Format:

- (1) SAVE program-file-name
- (2) SAVE program-file-name, ON
- (3) SAVE program-file-name, start-line-number, end-line-number
- (4) SAVE program-file-name; C

Explanation: A program-file-name is input in the following format.

[device-specification:] [path \] file-name [.extension]

(Character strings are used.)

When [device-specification] is omitted, the current drive is assumed. Upper-case alphabetic characters A through Z (one byte code) and numeric numbers can be used for file names. A file name must start with an upper-case alphabetic character (the same as the specification measures of file names in personal computers).

If an attempt is made to store a file in a directly that contains a file with the same name and the same extension, the following message appears.

File name "XXXXXXXX" exists. Overwrite (Y/N) ?

If you do not wish to overwrite the file, input N. Save the file using another file name.

Saving programs in format (1)

The entire program block in the user program area is saved with format (1) above: when subprograms are contained in the program in the user program area, the program is saved along with subprograms, only subprograms in user program areas if any, are also saved.

A program file is saved as source file. A source file is saved in character code format in the specified disk.

To identify the files from those of intermediate language, it is recommended to name the files with extensions "SA".

Only source files can be APPENDED or MERGED so save programs to be APPENDED or MERGED as source files.

When a program includes a library, it cannot be stored as a source file. When a program is being debugged or for utility program, save them in source file format.

Saving programs in format (2)

The designation ON is valid when the program in a user area includes subprograms. Once ON is designated, only the program block which was last designated by a PROG command is saved as a source file (see format (1) above). To save subprograms, save them after changing blocks with a PROG command.

To identify the files from those of intermediate language, it is recommended to name the file with extensions "SA".

To save subprograms, save them after changing blocks with a PROG command.

Saving programs in format (3)

Format (3) above is used for saving program blocks designated by the last PROG command in the program block in the user program area.

Programs from the start-line-number to the end-line-number are saved in source file format (refer to format (1) above). End-line-number cannot be omitted. When start-line-number \geq end-line-number, only the program line indicated by the start-line-number is saved.

To identify the files from those of intermediate language, it is recommended to name the file with extensions "SA".

Use this format when a part of the program which has been created can be utilized for another program.

Saving programs in format (4)

All program blocks in the current user area are saved with this format.

When subprograms are included in a program in the user program area, they are also saved. The program file is saved in the disk as an intermediate language file.

To identify the files from those of source files, it is recommended to name the file with extensions "UN".

Intermediate languages are translated codes ("tokens") that are simple for the BASIC interpreter to handle. An intermediate file uses less memory space in the disk than a source file. In addition, it takes less time to load such a program with the LOAD command. When the library is included in a program, the program can be saved only in format (4) (designating C).

Only programs saved in an intermediate language file can be initiated with a START statement. Programs saved as intermediate language files cannot be APPENDED or MERGED. Intermediate language files are used to execute debugged programs in real mode.

TIP

When subprograms are to be converted into subroutines, format (3) above is useful.

The only way to delete a once-entered SUB statement in a program is to delete the entire subprogram. Therefore, when subprograms are to be converted into subroutines as shown in the example below, proceed as follows:

(1) Enter PROG _S1, then press the ENTER key

(2) Enter SAVE _SAMPLE, 20, 30 then press the ENTER key.

Subprograms with the line numbers 20 to 30 are saved in a source file.

(3) Enter DEL _S1, then press the ENTER key.

Subprogram S1 is deleted from the user area.

(4) Enter PROG, then press the ENTER key.

Assume that the current program block is "main program".

(5) Enter APPEND _SAMPLE, 100, then press the ENTER key.

The program saved in SAMPLE is added to the main program from line number 100.

(6) Enter GOSUB _100 as line number 30, then enter STOP for line number 40.

(7) Enter RETURN for line number 120 and END for line number 130.

Example:

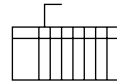
10 INPUT "A=?" ; A	10 INPUT "A=?" ; A
20 INPUT "B=?" ; B	20 INPUT "B=?" ; B
30 CALL S1(A,B)	30 GOSUB 100
40 END	40 STOP
10 SUB S1(A,B)	100 PRINT "A=" ; A
20 PRINT "A=" ; A	110 PRINT "B=" ; B
30 PRINT "B=" ; B	120 RETURN
40 SUBEND	130 END



FC030138.EPS

SCRATCH

- Command
- Statement



Function: Resets trace functions.

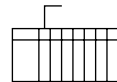
Format: SCRATCH [\leftarrow { subprogram-name }]

FC030139.EPS

Explanation: Resets the program branch trace functions (outputting the program branch) set by the TRACE statement. For subprogram names and ALL, see TRACE statement.

SCRATCHP

- Command
- Statement



Function: Releases temporarily suspended functions.

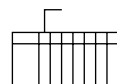
Format: SCRATCHP [\leftarrow { subprogram-name }]

FC030140.EPS

Explanation: Release temporarily suspended functions set by a TRACEP statement. For subprogram names and ALL, see TRACEP statement.

SCRATCHV

- Command
- Statement



Function: Releases variable trace functions set by the TRACEV statement.

Format: (1) When used as a command:

SCRATCHV [\leftarrow { subprogram-name }]

FC030141.EPS

(2) When used as a statement:

SCRATCHV

Explanation: This statement cancels variable trace functions set by a TRACEV statement. For subprogram names and ALL, see TRACEV statement.

SEQACTV

● Statement



Function: Starts/ends a ladder program in the sequence CPU in a program block.

Format: SEQACTV slot-number, block-number ; start/end-specification

Slot-number: Numeric-expression (integer type).

The number of slots (1 to 4) where the CPU in which a ladder program is operating to be started/stopped is defined.

Block-number: Numeric expression (integer type). The block numbers which can be specified are as follows:

Sequence CPU	Block number
F3SP21	1 to 32
F3SP25	1 to 128
F3SP35	1 to 1024

TC030118.EPS

Start/end-specification: { E; Program block end }
 { S; Program block start }

FC030142.EPS

Explanation: This statement starts/ends a ladder sequence program in the sequence CPU in a program block.

For program block start/stop-specification,

{ E: End }
 { S: Start } are used.

FC030143.EPS

In end-specification, it does not take more than one scan to end a ladder program from execution of this statement. If the entire program is not started, this statement is ignored even if starts a ladder program in a program block.

SET STATUS

● Statement



Function: Stores, if the execution of an I/O statement causes an error, the corresponding error code into variables without generating a system error message.

Format: SET STATUS status-information-variable

Status-information-variable: Simple numeric variable.

Explanation: If an error occurs when an I/O statement (see below) is executed, system error (program execution stop or branching by ON ERROR statement) is not assumed, the SET STATUS statement stores the corresponding error code into the status-information-variable (if a detail error occurs, the corresponding error code is stored into the ERRCE function) without causing system error (program execution stop or branch by ON ERROR statements). Then the execution proceeds to the next statement.

When the SET STATUS statement is executed, even if an error occurs, the corresponding error code is not stored into the ERRC function. When no error occurs, the content of the status-information-variable is still 0 (zero). To release the SET STATUS statement, use the RESET STATUS statement. The following statements allow status-information-variables to be used with the SET STATUS statements :

CONTROL,ENABLE,INTR,ENTER,HALT,OUTPUT,RESET,SET,TIMEOUT,STATUS

SET TIMEOUT

● Statement



Function: Sets the specified input/output monitoring time for the I/O module.

Format: SET _ TIMEOUT _ slot-number [, port-number] ; limit-time

Slot-number: Numeric expression (integer type).

Port-number: Numeric value or variable.

Limit-time: Numeric value or variable in milliseconds (resolution : 10 ms).
0 to 604800000ms (7 days).

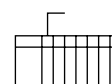
Explanation: Sets the specified input/output monitoring time (in milliseconds) for an I/O module. The time is set in units of 10 milliseconds. If 0 is set to the limit time, the default value for each I/O module is applied.

If an error occurs in SET TIMEOUT statement when the status information variable is designated in SET STATUS statement, the error code is stored in the status information variable.

In this case, no system error occurs.

SETMD RES

● Command



Function: Declares/cancels a BASIC program to be resident in the user area.

Format: (1) SETMD _RES _ON

(2) SETMD _RES _OFF

This command can be executed only in the debug mode.

Explanation: Format (1)

This format specifies residence of a BASIC program in the memory (user area).

The residence specification holds the BASIC program in the user area without loss even if power is turned off.

Format (2)

This format resets the program residence set by the format (1). To reset the status using another command, NEW command can be used. However, NEW command also deletes the program in memory.

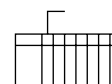
Notice in specifying program residence

- Residence specification when BASIC programs are not present in the user area causes an error.
- When a program residence is specified, commands or statements which load another program into the user area, such as LOAD command, cause an error and thus cannot be executed.

If these commands or statements are to be executed, first reset the residence specification.

SETMD RUN

● Command



Function: Specifies/cancels BYE&RUN mode which executes resident programs in real mode when debug mode has been ended by the BYE command.

Format: (1) SETMD _RUN _ON
(2) SETMD _RUN _OFF

Explanation: Format (1)

This format specifies BYE&RUN mode that executes resident programs in real mode when the debug mode is ended by the BYE command (mode).

Format (2)

This format cancels BYE&RUN mode specified in format (1) above.

Notice for usage

- By default, BYE&RUN mode is canceled.
- Although SETMD RUN ON command specifies BYE&RUN mode, the program cannot be executed if the program is not resident.

In addition, the program cannot be executed even if BYE&RUN mode has been specified, when the debug mode ends by using [CTRL]+[C] keys.
- If either of the following commands is executed, BYE&RUN mode is canceled to return to the default state.
 - NEW command
 - LOAD command

SGN

● Function



Function: Returns sign data.

Format: SGN(x)

x: Numeric expression.

Explanation: This function checks the sign of the numeric value or variable represented by x and returns the following:

- | | | |
|----|---------------------------|-------|
| 1 | when x is greater than 0. | (x>0) |
| 0 | when x is equal to 0. | (x=0) |
| -1 | when x is less than 0. | (x<0) |

SHIFT

● Function



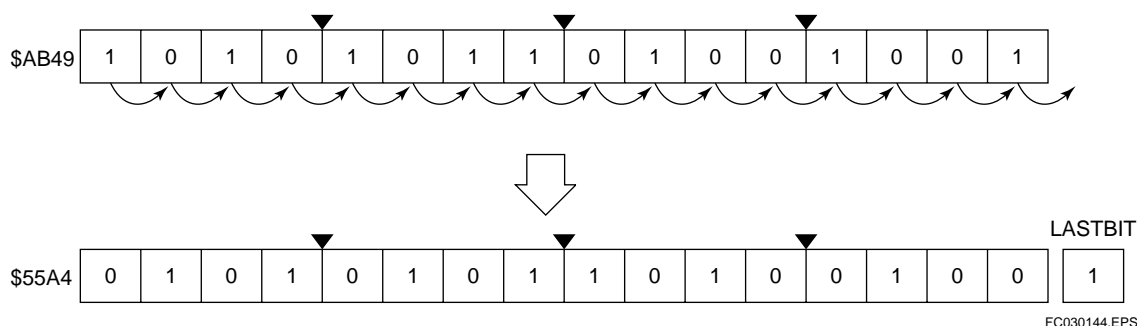
Function: Shifts bits in 16-bit integer.

Format: SHIFT(m , n)

m , n: Numeric expression (integer type).

Explanation: This function is used to shift bits represented by m (numeric value or variable) by n bits in a 16-bit integer. When n is positive, the bits are shifted right. When n is negative, the bits are shifted left. Last bit removed from a bit string is assigned to the LASTBIT function.

Example: For SHIFT (\$AB49, 1).



SIN

● Function



Function: Returns the sine of x.

Format: SIN(x)

x: Numeric expression.

Explanation: This function returns the sine of the numeric value or variable represented by x in radians.

SPC

● Function



Function: Outputs the number of specified blank spaces.

Format: SPC(x)

x: Numeric expression.

Explanation: This function determines the positive integer n obtained by rounding-off the value x and output n spaces

When the value of x is 0 or negative, no spaces are returned.

SQR

● Function



Function: Returns the square root of x.

Format: SQR(x)

x: Numeric expression (≥ 0).

Explanation: This function returns the square root of the numeric value or variable represented by x. The value of x must be equal to or greater than 0.

STATUS

● Statement



Function: Reads the specified module status.

Format: (1) STATUS slot-number, port-number [, register-number] ; numeric-variable

(2) STATUS slot-number, 1 ; reference-variable

Slot-number: Numeric expression (integer type)

Port-number: Numeric expression. Can specify two or more numbers.

Register-number: Numeric expression.

Reference-variable: Numeric variable.

1 : Stopping status

2 : Running status

Negative values : Error-occurring state

Explanation: Format (1)

Format (1) is used for I/O modules.

Reads parameters set in the module or module-status values and store them in the specified variable. In other words, reads module-specific information or special parameters, and save them in the specified variable.

Format (2)

Format (2) is used for sequence CPU modules.

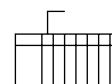
Reads the running / stopping status or error-occurring state of the sequence CPU module.

If an error occurs in STATUS statement when the status information variable is designated in SET STATUS statement, the error code is stored in the status information variable.

In this case, no system error occurs.

STEP

● Command



Function: Executes the next line of the program suspended by a PAUSE, TRACEP statement or the [ESC] key.

Format: STEP

Explanation: The STEP command is normally used during debugging. It cannot be used when BASIC is activated on a stand-alone personal computer.

When the STEP command is executed after the variables are checked or rewritten with the program set in a PAUSE status using PAUSE, TRACEP statement or the [ESC] key, only the next one line is executed and the program returns to a PAUSE status again.

This command is useful for inspecting or checking the program operations line by line.

If the STEP command is executed at other than execution suspension time, an error may occur.

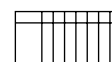
The STEP command cannot be executed even in execution suspended time,

- When the program list is corrected (including EDIT) or
- When the program block is changed (using PROG command).

Further, when the execution is suspended by a STOP or END statement, the STEP command cannot be executed. In addition, the STEP command cannot be executed at the head of the program.

STOP

● Statement



Function: Stops the execution of the program.

Format: STOP

Explanation: The STOP statement may be omitted when the program is to be stopped after execution of the line with the largest line number of the main program. (END statement has the end declaration function). When the STOP statement is executed, all files that have been declared open are closed.

When the STOP statement is executed, the message given below is displayed in the display.

```

STOP ***** Line=XXXXX
      |           |
      |           |----- Line-number of STOP statement
      |
      |----- Program-name
  
```

FC030145.EPS

The STOP statement can be used only in a main program.

STR\$

● Function



Function: Returns a character string representing a numeric value or variable.

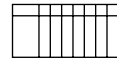
Format: STR\$(x)

x: Numeric expression.

Explanation: This function converts the value of a numeric value or variable represented by x into a character string. When a character string is to be converted into a numeric value, use the VAL function.

SUB

● Statement



Function: The SUB statement declares a subprogram.

Format: (1) SUB _ subprogram-name

(2) SUB _subprogram-name(formal-argument [, formal-argument...])

Explanation: Format (1) is used to declare a subprogram requiring no arguments.

Format (2) is used to declare a subprogram requiring arguments.

For the types and usage of available arguments that can be used, see the CALL statement.

When a formal-argument is an array variable, a DIM or COM statement for the formal argument need not be declared. When branching using ON....., format (2) cannot be used.

Differences between subroutine and subprogram are described below.

When a variable (formal argument) is used, a subroutine must have the same variable name as that for the main routine. When a subprogram is used, different variable names may be used provided that they have the same variable type. The execution time for each subprogram should be 1 second or longer. When the execution time is less than 1 second, perform processing in the main program or set it to a subroutine. If a subprogram has an execution time of less than 1 second, the overall program processing time may be longer.

Notices in creating a program

- Main programs and subprograms share the following items. Note this fact in program creation.

Card-slot-number and timer-number.

SUBCOM

● Statement



Function: Specifies the start position of a subprogram common area.

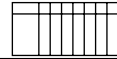
Format: SUBCOM [_ common-variable-name]

Explanation: The SUBCOM statement is used within a main program. The start position of the common area specified by a COM statement in a subprogram is set to the position of the common variable specified in the preceding SUBCOM statement.

When a common-variable-name is not specified or the SUBCOM statement itself is omitted, the subprogram common area is positioned at the start of the main program common area.

SUBEND

● Statement



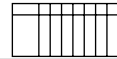
Function: Designates the end of a subprogram.

Format: SUBEND

Explanation: The SUBEND statement, which is entered at the end of a subprogram, cannot be omitted. When the SUBEXIT statement is omitted, the SUBEND statement is also used to return to the main program from a subprogram.

SUBEXIT

● Statement



Function: Returns control from a subprogram.

Format: SUBEXIT

Explanation: The SUBEXIT statement return control to the statement following the CALL statement which called the subprogram.

The SUBEXIT statement may be omitted when terminating subprogram execution at the end of the subprogram (The SUBEND statement is provided with a return declaration function).

The variable area ensured in a subprogram is released when SUBEXIT statement is executed.

SUBEXIT RETRY

● Statement



Function: Returns from a subprogram to the line in which the branch occurred.

Format: SUBEXIT $\underline{\hspace{1cm}}$ RETRY

Explanation: As a rule, the SUBEXIT RETRY statement is used to return from a subprogram accessed with an ON ERROR CALL when is necessary.

The SUBEXIT RETRY statement declares a return from a subprogram accessed with the CALL statement to the line in which the branch occurred.

While the SUBEXIT statement returns to the next line following the line in which the branch occurred, the SUBEXIT RETRY statement returns to the line in which the branch occurred.

Hence, the SUBEXIT RETRY statement is mainly used when an error occurs during I/O access and a branch is taken with an ON ERROR CALL

SWAP

● Statement



Function: Swaps the values of two variables.

Format: SWAP $\underline{\hspace{1cm}}$ variable-name , variable-name

Explanation: Values can be swapped between variables of any type, as long as the variables are of the same variable type (integer, long integer, single-precision real number, double-precision real number, or character string). If the variables are of the same variable-type, values can be swapped between array variable and simple variable, or between array variables. A whole array cannot be swapped with SWAP statement.

TAN

● Function



Function: Returns the tangent of x.

Format: TAN(x)

x: Numeric expression.

Explanation: This function returns the tangent of the numeric value or variable represented by x in radians.

TIME\$

● Function



Function: Returns the time.

Format: TIME\$

Explanation: This function returns the current time of day in the form a character string hh:mm:ss, where hh represents hours, mm represents minutes, and ss represents seconds.

To change the time, use the BASIC Programming Tool M3 for Windows.

For details, see the Instruction Manual for BASIC Programming Tool M3 for Windows (IM 34M6Q22-02E).

TIMEMS

● Function



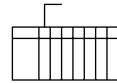
Function: Returns the elapsed time in ms (millisecond) since 0 hour 0 minute.

Format: TIMEMS

Explanation: This function is used to return a measure of the elapsed time in ms since 0 hour 0 minute (24-hour system). The resolution is 10 ms. TIMEMS has a single-precision real numeric value.

TRACE

- Command
- Subcommand



Function: Outputs information on program branches occurring during program execution.

Format: TRACE $\left[\begin{array}{l} \text{subprogram-name [; start-line-number , end-line-number]} \\ \text{start-line-number , end-line-number} \\ \text{ALL} \end{array} \right]$

FC030146.EPS

Explanation: This statement is valid only in debug mode and ignored in real mode.

The TRACE statement checks for program branches occurring in a specified range of lines in a specified program block. If the YEWMAC line computer is connected, the trace output device can be specified in a LISTDEV statement independently of the output device specified in a PRINT statement in the program. The program block in which program branches are traced can be selected by specification of parameters in the following formats.

(1) When a TRACE statement is directly entered as a command:

Format	Program Block
Subprogram-name specified	Specified subprogram.
Subprogram-name omitted	<ul style="list-style-type: none"> Current program block (When a TRACE statement is used in pause status). Main program block (When a TRACE statement is used in other than pause status).
TRACE ALL	Entire program blocks.

TC030119.EPS

(2) When a TRACE statement is entered in a program:

Format	Program Block
Subprogram-name specified	Specified subprogram.
Subprogram-name omitted	A program block containing a TRACE statement.
TRACE ALL	Entire program blocks.

TC030120.EPS

The range of lines in the program block in which program branches are traced can be selected in the following formats:

Format	Explanation
Both start-line number and end-line-number omitted	All lines in the program block
Both start-line-number and end-line-number specified	Lines ranging from start-line-number to end-line-number

TC030121.EPS

Labels cannot be used to specify line numbers in a TRACE statement. Trace output data includes both the line numbers at which program branches occur and the line numbers to which control is subsequently passed. For branches to subprograms, however, the line numbers from which control is passed and to which control is returned are output.

To cancel the TRACE function, use the SCRATCH statement.

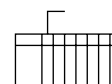


CAUTION


The TRACE statement line number cannot be changed by the RENUM command.

TRACEP

- Command
- Statement



Function: Suspends the execution of a program immediately before executing the specified line.

Format: TRACEP [ subprogram-name ;] line-number [, number-of-times]

Line-number: Numeric constant (Labels cannot be used as line numbers).

Number-of-times: Numeric constant.

Explanation: This statement is valid only in debug mode and ignored in real mode.

The TRACEP statement pauses the execution of a program immediately before executing a specified line in a selected program block the designated number of times. (This program state is similar to the PAUSE statement.) The TRACEP statement must be described before the line specified by the line-number. When the number-of-times is omitted, the program pauses each time before executing the specified line. The TRACEP statement is valid only in debug mode.

In real mode, the TRACEP statement only generates a message without pausing.

Only one TRACEP statement can be specified for one program block. When several TRACEP statements are specified, the last TRACEP statement is valid. The TRACEP statement line number is not changed with an RENUM command. To cancel the TRACEP statement, execute a SCRATCHP statement. The TRACEP statement is used in program blocks shown below:

Format	Program Block
Subprogram-name specified	Named subprogram
Subprogram-name omitted	When TRACEP is used as a command. : Current program block (specified by a PROGcommand) When TRACEP is used as a statement. : A program block containing a TRACEP statement

TC030122.EPS

To cancel the TRACEP function, use SCRATCHP statement.

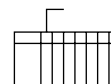


CAUTION

The TRACEP statement line number cannot be changed by the RENUM command.

TRACEV

- Command
- Statement



Function: Outputs variable changes occurring during program execution.

Format: TRACEV \square [subprogram-name ;] variable-list

Variable-list: When more than one variable is specified, separate them with commas “,”.
Up to five variables can be specified.

Explanation: This statement is valid only in debug mode and ignored in real mode.

The TRACEV statement outputs changes in specified variables (line numbers, variable names, and variable values) in the program block to a personal computer display or the YEWMAC line computer. Program blocks are specified in the same way as those for a TRACE statement. The YEWMAC line computer output device can be specified by the LISTDEV statement.

The output format is as follows:

TRACE — “program-block-name” LINE line-number variable = assigned-value

Line number in which —
the variable value is changed

Variable —

Value assigned to variable. —

FC030147.EPS

Program block names are displayed as follows:

	Program Entered from Keyboard	Program Entered from Auxiliary Memory Unit
Main program	*****	Program Name
Subprogram	Subprogram Name	Subprogram Name

TC030123.EPS

One TRACEV statement can be specified at a time for one program block. When more than one TRACEV statement is specified, the last TRACEV statement is valid.

The variables for this statement are as follows:

For simple variables: Specify variable names.

For array variables: Specify variable names without ().

Example: Specify A for A(*) or A(3).

Array variable elements cannot be specified on a one-by-one basis.

All elements of the specified array variable are output.

To release the TRACEV statement, use the SCRATCHV statement.

TRANSFER

● Statement



Function: Used for starting I/O operation of I/O modules.

Format: TRANSFER__slot-number [, instrument-number] $\left[\begin{array}{c} \text{FORMAT} \\ \text{NOFORMAT} \\ \text{BFORMAT} \end{array} \right] \text{---} \left\{ \begin{array}{c} \text{FROM} \\ \text{INTO} \end{array} \right\}$
 __character-string-variable

FC030148.EPS

Slot-number: Numeric expression (integer type).

Instrument-number: Numeric expression (Whether it can be specified depends on the I/O modules).

Character-string-variable: Character string variable to store data (character strings) to be input or output.

Explanation: This statement is used for inputs or outputs for I/O (specifically communication) modules. It is always used in a pair with ON EOT statement.

TRANSFER...FROM is used to output data. This has the same functions as an OUTPUT statement.

TRANSFER...INTO is used to input data. This has the same functions as the ENTER statement. Unlike OUTPUT and ENTER statements, the TRANSFER statement, after initiating I/O operations, advances to the next step of BASIC program regardless of completion of I/O operations.

Since completion of I/O is informed by branch with ON EOT statement, declare ON EOT statement before execution of TRANSFER statement.

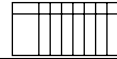
If a character string is output like OUTPUT statement, the character string preceding the null code is recognized as an effective character string. The null code cannot be output.

Note, that when a character string is initiated with a null code, output is disabled, and no branch occurs with an ON EOT statement.

Specification of formats such as NOFORMAT and others must be executed by combining with the specification of ENTER/OUTPUT formats. For details, see instruction manuals for each I/O module.

VAL

● Function



Function: Returns a numeric value represented by a character string.

Format: VAL(c)

c: Character string expression.

Explanation: This function converts a character string c into a numeric value.

Conversion of a character string is performed as follows:

- The first character in a character string c must be a numeral, a + or – sign, \$, a decimal point or a space. If other character other than those above, when used, will cause an error.
- When the first character in a character string c (excluding “\$” if the leading character is \$) can be converted into a numeric value, characters from the first character to a convertible one are converted. In this case, even if a character that cannot be converted into a numeric value is contained in the character string c, no error occurs.
- When the first character in a character string c is “\$” and the character just after “\$” is a numerical or “A” to “F”, four characters beginning with other than zero are converted into a numeric value. Zero following “\$” is ignored. However, when a character that cannot be converted is encountered, conversion to a numeric value is truncated even if the number of characters is less than 4.
- When character string c is composed of Null characters (CHR\$(0)), it is converted into zero.

WAIT

● Statement



Function: Suspends or delays the execution of a program.

Format: (1) WAIT
 (2) WAIT time-delay
 Time-delay is specified as a numeric expression.
 1 to 604800000 (7 days). Set in units of millisecond.

Explanation: Format (1).

This WAIT format stops the execution of a program. The program is released from the wait status when an event occurs (when ON ... GOSUB or ON ... GOTO statement is executed) or when the "STOP" key is pressed. When ON ... GOSUB statement is executed, control passes to the statement next to the WAIT statement after return. When ON ... GOTO statement is executed, control passes to the statement specified by GOTO. When an interruption is disabled by a DISABLE statement (except DISABLE C), if format (1) above is used, the operation is the same as that of an ENABLE statement (to cancel the DISABLE statement) is executed unconditionally. During this operation, ENABLE C is not performed. While an interruption is processed by ON ... CALL or ON ... GOSUB statement, if the format (1) is used, an error will occur.

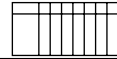
Format (2).

With this format, program execution is delayed for the specified time (milliseconds). While the program execution is delayed, if an event occurs (when the ON ... GOSUB or ON ... GOTO statement is executed), the program branches to the specified line. When a branch was caused by a ON ... GOSUB statement, control, after return, waits the remaining specified time delay before passing to the next statement after return. When a branch was caused by the ON ... GOTO statement, program execution is transferred to the specified line (without any time delay). While the program execution is delayed, if the "STOP" key is pressed, the delay is canceled and execution is suspended. At this time if a CONT command is executed, the statement following the WAIT statement is executed. This format (2) allows program execution to be delayed even when an interruption is disabled by the DISABLE statement. While an interruption is executed by the ON ... CALL or ON ... GOSUB statement, program execution can be delayed using this format (2).

When used as a multiple statement, the WAIT statement must be used at the end of the line and cannot be used at any other position of the line either in format (1) or in format (2). (An error will occur.)

WHILE/END WHILE

● Statement



Function: Executes statements repeatedly while conditions are satisfied.

Format: WHILE expression
 to
 END WHILE

Explanation: While the condition of the expression is true ($\neq 0$), statements between the WHILE statement and END WHILE statement are executed repeatedly. When the condition of the expression is false ($=0$), program goes to the statement after the END WHILE statement. If a condition of the relational or logical expression specified by the WHILE statement is not satisfied, statements between the WHILE statement and END WHILE statement are not executed. A relational expression such as $(A>0)$ can be used. Other types of expressions can be used if numeric values must be resulted. The WHILE/END WHILE statement may be nested for multiplexing like FOR NEXT (see FOR NEXT statement). A program in the loop of the WHILE and END WHILE statements may branch outside the loop by using a GOTO statement. But an external program cannot branch into the loop of the WHILE and END WHILE statements. The WHILE and END WHILE statements cannot be used with a multiple statement.

C4. Error Code List

C4.1 YM-BASIC/FA Error Codes

Error Code	Error Message
System errors	
1	Intermediate code not interpreted
2	Parameter error
3	Timer error
Syntax error	
4	Incorrect syntax (invalid operand specified)
5	RESERVED
6	Executing TRANSFER statement (attempted to access module while executing TRANSFER statement)
Computation error	
7	Computation overflow
8	Division by zero
9	Underflow error
10	Integer overflow
11	Unmatched data type (incorrect data results)
12	Numeric value abnormal or argument (function) error
13	Unallocated variable
14	Undefined variable name
15	String concatenate overflow
16	Cannot assign
Array variable declaration error	
17	Invalid subscript value (less than 0 or above 32768) in ALLOCATE statement
18	Array size exceeded
19	Common area size exceeded
Used-defined function error	
20	Undefined function
21	RESERVED
22	Wrong number of arguments
23	RESERVED
24	Attempt to redefine a previously defined function
Reference to array variable	
25	Array subscript out of range
26	Array subscript dimension error
FOR - NEXT error	
27	Improper FOR ~ NEXT, WHILE ~ END WHILE, IF ~ ENDIF matching
28	Needs simple numeric variable in FOR statement
29	RESERVED

TC040101.EPS

Error Code	Error Message
Insufficient data	
30	Too many entries (excessive input data)
31	Too few entries (insufficient data)
32	No data to be read by READ statement
Unmatched data type	
33	Unmatched
34	RESERVED
Image data error	
35	Improper match between IMAGE and data item
36	RESERVED
37	RESERVED
38	RESERVED
Output overflow	
39	Output data overflow
40	Value overflow
41	RESERVED
Invalid line number	
42	IMAGE statement missing in a line referenced
43	Branch destination for GOTO/GOSUB statement not found
44	Program statement name not found
Sequence error of declaration statements	
45	Invalid position of OPTION BASE statement
46	Invalid position of DEFINT, DEFLNG, DEFSNG, DEFDBL statements
47	Invalid operand in DEFINT, DEFLNG, DEFSNG, DEFDBL statements
48	RESERVED
49	RESERVED
50	RESERVED
Inter-user area communications error	
51	RESERVED
52	Communications error/CPU type error
53	Communications error/CPU type error
54	Communications error/CPU type error (refer to detail error codes)
55	SIGNAL transmission error
56	RESERVED
57	RESERVED
58	RESERVED
59	RESERVED
60	RESERVE, RELEASE statement error

TC040102.EPS

Error Code	Error Message
61	Invalid numeric value in ON statement
62	Existing variable or array redeclared
63	RESERVED
64	Incorrect RETURN statement (RETURN with no GOSUB)
65	RESERVED
66	RESERVED
67	Numeric conversion error
68	Error detected in ON ERROR processing
69	RESERVED
70	Attempt to execute a statement not executable.
71	RESERVED
72	RESERVED
73	RESERVED
74	RESERVED
75	Prerun error, conflicting program
76	Conflicting error in main program
77	RESERVED
78	Syntax error in subprogram
Memory overflow	
80	Stack area is now being used (insufficient area).
81	Insufficient area for reserving variable areas
82	I/O error (see detailed error codes)
File or library error	
83	File name (program name) already exists.
84	File name (program name) not found. File name contains double-width spaces.
85	END OF FILE
86	RESERVED
87	RESERVED
88	Error generated in library 88-7□ □: □-th parameter error
92	Resident mode error
93	Resident program size too large
94	Resident program upload error
95	CHAIN statement execution error
System error	
101	Insufficient dynamic free area
102	Variable name not found in the symbol table
103	Numeric data value abnormal
104	Numeric data value abnormal
105	Source line length exceeds 306 bytes.

TC040103.EPS

Error Code	Error Message
106	Non-reserved word in program code
107	Error in analysis of a computational expression
108	Reserved word code not found
109	Coding error in coded text
110	Parameter error in subroutine call
111	Editor error
112	Editor error
113	Editor error
114	Editor error
Command error	
115	Command entry validity error, command-disabled state due to Type SS, US, or SB program
116	Syntax error
117	Statement not found
Invalid line number	
118	Line number not found
119	Invalid line number
120	Line number exceeds 65535.
Alarm	
121	Character string not found
122	Free area not more than 400 bytes
Subprogram error	
123	Subprogram not found
124	Unable to assign new subblock. Illegal replacement of SUB statement
Syntax error	
130	Unable to assign stack area
131	Invalid array variable in a statement
132	GO, GOTO, GOSUB not described in ON statement.
133	Invalid timer number
134	RESERVED
135	Variable name other than that described as operand was used.
136	Incorrect line number or comma described as operand
137	Invalid significant character string
138	Invalid program name
139	FROM or INTO operand in TRANSFER statement.
140	TO operand missing (FOR statement, etc.)
141	RESERVED
142	RESERVED
143	Invalid operand description
144	Computational expression error

TC040104.EPS

Error Code	Error Message
145	RESERVED
146	RESERVED
147	Invalid data list
148	Invalid #Tn or #Un description
149	Invalid binary constant
150	RESERVED
151	Invalid line number or label
152	Too many variable names or labels
153	Invalid variable type declaration
154	Nesting of IF statement exceeds 16 levels.
155	THEN without processing statement
156	ELSE without corresponding IF
157	ENDIF without corresponding IF
158	Statement not terminated in correct format
159	Left part or “=” does not appear in computational expression
160	Invalid DEF statement
161	Permissible numeric size exceeded
162	Invalid hexadecimal constant
163	Invalid FIND command operand
164	Undefined statement type
165	Command headed by a line number
166	Not executed with immediately executable statement
167	Six or more LF codes found between significant characters other than blank characters
168	Invalid statement number
169	Coded statement area (514 bytes) not assigned
170	Statement not allowed in multiple statement line
171	Data type error
172	Invalid subscript
173	Operand error (incorrect operator described in character expression)
174	Variable, label or line number not defined in an immediately executable statement

TC040105.EPS

C4.2 Detail Error Codes

The detail error code is output for BASIC error codes of 082 (I/O error), 054 (shared access error), 055 (SIGNAL transmission error). For details, see instruction manuals for each module.

● I/O errors

82- xx

The following describes the meaning of error codes and probable causes where errors are output when accessing the Sequence CPU Module and Contact Input/Output Module.

Where errors appear when accessing another module, see applicable instruction manual.

Detailed Error Code (expressed in hexadecimal)	Error Message	Probable Cause
01	No driver exists.	
0C	Insufficient system area	
13	Driver internal error	
81	No driver exists.	
82	Invalid function	<ul style="list-style-type: none"> Attempted to execute statement which module does not support (OUTPUT statement for input module, etc.). Incorrect I/O module slot number
83	Invalid logical file number	
84	Invalid buffer length	<ul style="list-style-type: none"> Invalid number of devices
85	Invalid parameter	<ul style="list-style-type: none"> Invalid parameter
86	Invalid parameter address	
91	Invalid parameter	<ul style="list-style-type: none"> Device number out of range Invalid parameter
92	Invalid data setting	<ul style="list-style-type: none"> Data setting incorrect (data type, etc.) Specified terminal number other than 0 or 1
93	Invalid command description	<ul style="list-style-type: none"> Invalid format specified
94	Invalid module specified	<ul style="list-style-type: none"> Invalid module name assigned Module name not assigned
95	Invalid bit pattern	
9A	Number of specified processing requests exceeded	
9B	Number of specified processing requests exceeded	
9C	Internal error in ASSIGN statement	
9D	ASSIGN statement not executed, I/O not installed	<ul style="list-style-type: none"> Incorrect module name assigned Module name not assigned Sequence CPU module not found

TC040201.EPS

Detailed Error Code (expressed in hexadecimal)	Error Message	Probable Cause
A1	Incorrect slot number	
B0	Invalid access procedure	
B1	Invalid module designation or data number	• Incorrect device name
B2	Data high/low limit overflow	
B3	Invalid device specification	• Incorrect read/write units
B4	Improper number of devices accessed	• Incorrect device name
B5	Invalid data or code	
B6	Invalid module designation	
BB	Invalid interruption code or number	
BC	Interruption code already requested.	
C1	Buffer overflow	
C7	I/O reset detected	
CD	Insufficient area (driver work area)	
D1	Device error	• Faulty module
D2	Data verification error	
D4	Receiving data error	
D5	Communication error	
D6	No terminator in received text	
D7	Communication error	
D8	Hardware error during data transmission	
D9	Received text header information invalid	
DA	Program initiation text received	
DB	Invalid transmission mode	
DC	Buffer overflow	
E1	DEVICE NOT READY	• Module not installed • Incorrect slot number • Faulty module
E2	DEVICE BUSY	• Sequence CPU module cannot receive a BASIC statement.
E3	Data error	
E6	Timeout	• Received statement not processed within the given time.
E8	Checksum error	
EA	Data overrun	
F1	Statement-execution check error	• Sequence CPU module not in statement execution status
F2	Improper internal status	• Statement executed, but sequence CPU module is not in normal status
F3	Internal error	
F4	Internal error	
F6	Internal error	
FE	Device not configured	

TC040202.EPS

● Shared access errors

54- xx or 55- xx

The following describes the meaning of error codes and probable causes where errors appear at the time of shared accessing.

Detailed Error Code (expressed in hexadecimal)	Error Message	Probable Cause
03	Destination BASIC task not found	
05	Event-receive intermediate buffer overflow in destination BASIC	
82	Internal error	
84	Incorrect buffer length specification	<ul style="list-style-type: none"> Common variables are not of integer type or long integer type.
88	Incorrect destination UNIT type (incorrect configuration)	
89	Incorrect own UNIT type	
91	Invalid parameter	<ul style="list-style-type: none"> Invalid parameter
92	Invalid data setting	<ul style="list-style-type: none"> Set data do not match with configuration.
9D	Sequence CPU not installed	<ul style="list-style-type: none"> Incorrect module name assigned Sequence CPU not assigned Sequence CPU module not found
A7	Unconnected UNIT no. specified	
B1	Internal error	
B2	Internal error	
B3	Internal error	
B4	Internal error	
B5	Internal error	
B6	Internal error	
B7	Internal error	
BA	Internal error	

TC040203.EPS

Detailed Error Code (expressed in hexadecimal)	Error Message	Probable Cause
C1	Destination memory access error	
C3	Internal error	
C4	Internal error	
D1	Error found in a communication line	
D2	Error found in a communication line	
D3	Error found in a communication line	
D4	Error found in a communication line	
D5	Error found in a communication line	
D6	Error found in a communication line	
D7	Error found in a communication line	
D8	Error found in a communication line	
D9	Error found in a communication line	
DA	Error found in a communication line	
DB	Error found in a communication line	
DC	Error found in a communication line	
E1	DEVICE NOT READY	<ul style="list-style-type: none"> • Module not installed • Invalid slot number • Faulty module • Subunit turned OFF/ON at the time of Shared accessing
E6	Timeout	
FE	Device not configured	Invalid slot number

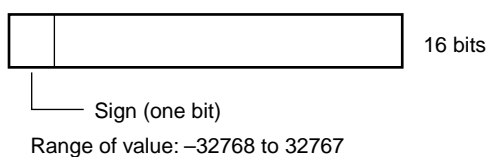
TC040204.EPS

Appendix 1. Listing of Internal Codes

Appendix 1.1 Data Formats

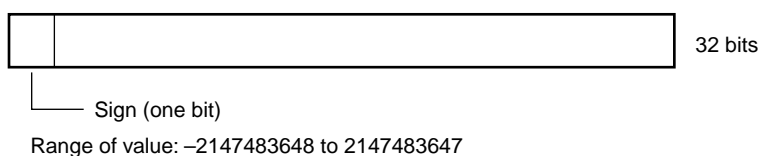
The YM-BASIC/FA programming language has four data types: an integer, long integer, single-precision real number, and double-precision real number. These data types differ in format from one another. You do not have to pay any regard to the data format, however, unless you use such arithmetic functions as those for handling binary digits.

■ Integers



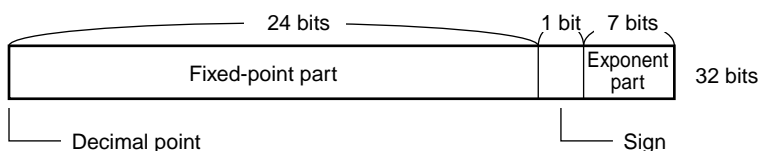
FAP0101.EPS

■ Long Integers



FAP0102.EPS

■ Single-precision Real Numbers



FAP0103.EPS

Notes 1: The exponent part is shifted at 2^6 . For example, it is 2^0 if the least-significant 7 bits is \$40.
2: The fixed-point part is always represented as a whole number.

Examples:

0.0	00 ----- 0	0	00 ----- 0
0.5	10 ----- 0	0	10 ----- 0
-0.5	10 ----- 0	1	10 ----- 0

FAP0104.EPS

Range of value in decimal number representation

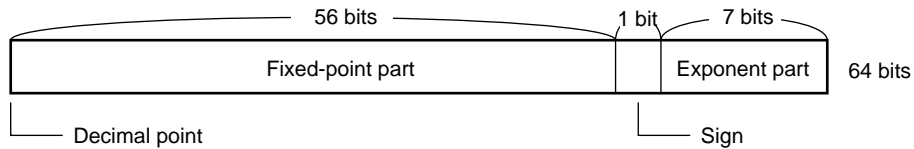
$9.223372 \times 10^{18} \geq \text{Positive value} \geq 5.421010 \times 10^{-20}$;

$-9.223372 \times 10^{18} \leq \text{Negative value} \leq -2.710505 \times 10^{-20}$; and

0 (zero)

where, the number of significant digits is 7.

■ Double-precision Real Numbers



FAP0105.EPS

Note: A double-precision real number shares the same format with a single-precision real number, except that its fixed-point part is 32 bits longer.

Range of value in decimal number representation

$$9.223372036854776 \times 10^{18} \geq \text{Positive value} \geq 5.421010862427522 \times 10^{-20};$$

$$-9.223372036854776 \times 10^{18} \leq \text{Negative value} \leq -5.421010862427522 \times 10^{-20}; \text{ and}$$

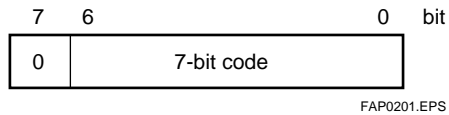
0 (zero)

where, the number of significant digits is approximately 16.

Appendix 1.2 Character Code Format

Each alphanumeric character of the YM-BASIC/FA programming language is represented by a character code of one byte, which equals 8 bits, as shown below.

■ Alphanumeric Character



Appendix 1.3 Listing of Alphanumeric Character Codes

									0	0	0	0	0	0	0	0
									0	0	0	0	1	1	1	1
									0	0	1	1	0	0	1	1
									0	1	0	1	0	1	0	1
b8	b7	b6	b5	b4	b3	b2	b1		0	1	2	3	4	5	6	7
				0	0	0	0	0	NUL	DLE	SP	0	@	P	'	p
				0	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
				0	0	1	0	2	STX	DC2	"	2	B	R	b	r
				0	0	1	1	3	ETX	DC3	#	3	C	S	c	s
				0	1	0	0	4	EOT	DC4	\$	4	D	T	d	t
				0	1	0	1	5	ENQ	NAK	%	5	E	U	e	u
				0	1	1	0	6	ACK	SYN	&	6	F	V	f	v
				0	1	1	1	7	BEL	ETB	'	7	G	W	g	w
				1	0	0	0	8	BS	CAN	(8	H	X	h	x
				1	0	0	1	9	HT	EM)	9	I	Y	i	y
				1	0	1	0	A	LF	SUB	*	:	J	Z	j	z
				1	0	1	1	B	VT	ESC	+	;	K	[k	{
				1	1	0	0	C	FF	FS	,	<	L	\	l	
				1	1	0	1	D	CR	GS	-	=	M]	m	}
				1	1	1	0	E	SO	RS	.	>	N	^	n	~
				1	1	1	1	F	SI	US	/	?	O	_	o	DEL

Appendix 2. Listing of Reserved Words

ABORTIO	COPY	DP	GLOAD	LCOPY
ABS	COS	DU	GMODE	LEFT\$
ALL			GO	LEN
ALLOCATE			GOSUB	LET
AND	DATA	EDIT	GOTO	LHEX\$
APPEND	DATE\$	ELLIPSE	GPOSITION	LINE
ARNAM	DBADD	ELSE IF	GPRINT	LINKLIB
ASC	DBCLOSE	ELSE	GRAPHIC IS	LINPUT
ASSIGN	DBCOND	ENABLE	GSAVE	LINPUT#
AT	DBDEF	ENABLE INTR	GSELECT BT	LIST
ATN	DBDEL	END	GSELECT NT	
ATTR\$	DBDELM	ENDIF		LISTDEV
AUTO	DBDROP	END WHILE		LOAD
	DBEND	ENTER	HALT	LOCAL
	DBFIND	EOL	HEX\$	LOCAL LOCKOUT
BATCH	DBFLD	ERASE	HINSTR	LOF
BATCHOFF	DBFLUSH	ERLIST	HLEFT\$	LOG
BCD	DBGET	ERRC	HLEN	LROTATE
BEEP	DBGETM	ERRCE	HMID\$	LSHIFT
BFORMAT	DBINIT	ERRCS	HRIGHT\$	
BINAND	DBOPEN	ERRL		
BINNOT	DBORDER	ERRM\$		MAINTENANCE
BINOR	DBPUT	EXOR	IF	MASTER
BINXOR	DBPUTM	EXP	IMAGE	MERGE
BIT	DBRLS		INIT	MID\$
BLEN	DBUPD		INPUT	MOD
BOOT	DEF	FIELD#	INPUT#	MOVE
BOX	DEFAULT OFF	FILES	INPUT\$	
BOX FILL	DEFAULT ON	FIND	INSTR	
BYE	DEFDBL	FOR	INT	NAM
	DEFFILE	FORMAT	INTO	NEW
	DEFGCUR	FRAME	IOLOAD	NEWL
CALL	DEFINT	FREE	IOSAVE	NEXT
CALLLIB	DEFLNG	FROM	IOSIZE	NL
CHAIN	DEFSHORT	FUSING		NOFORMAT
CHAR	DEFSNG			NOT
CHG	DEFVOL		KEY IS	
CHR\$	DEL	GCOLOR	KEY LABEL	
CIRCLE	DEL#	GCREADX		OFF COMINT
CIRCLE FILL	DELF	GCREADY		OFF EOF
CLOSE	DELP	GCURSOR OFF	LASTBIT	OFF EOT
COL	DIM	GCURSOR ON	LBCD	OFF ERROR
COLOR	DISABLE	GDISPLAY OFF	LBINAND	OFF EVENT
COM	DISP	GDISPLAY ON	LBINNOT	OFF EXEVENT
COM IS	DISP USING	GERASE	LBINOR	OFF GRAPHIC
CONT	DIV	GET#	LBINXOR	OFF INPUT
CONTROL	DJLINE	GINIT	LBIT	OFF INT

TAP0201.EPS

OFF KEY	PI	READ IO	SCREEN	TAN
OFF SEQEV	POINT	RECEIVE	SET MARKER	THEN
OFF SYSEV	POKE	RECOM	SET PAINT	TIME\$
OFF TIME	POLYGON	RELESE	SET PEN	TIMEMS
OFF TIMEOUT	POLYGON FILL	REM	SET STATUS	TO
OFF TIMER	POSITION	REMOTE	SET TEXT	TRACE
ON	POSITION#	RENAME	SET TIMEOUT	TRACEP
ON COMINT	POSX	RENUM	SETDAY	TRACEV
ON EOF	POSY	REPEAT	SETMD RES	TRANSFER
ON EOT	PR	RESERVE	SETTIME	TRIGGER
ON ERROR	PRCSR	RESET	SGN	
ON EVENT	PRCSWT	RESET STATUS	SHIFT	
ON EXEVT	PRELEASE	RESTORE	SIGNAL	UN TIL
ON INPUT	PRESERVE	RETRY	SIN	USING
ON INT	PRINT	RETURN	SLOAD	
ON KEY	PRINT BFORMAT	RIGHT\$	SPC	
ON SEQEV	PRINT NOFORMAT	RND	SQR	VAL
ON SYSEV	PRINT USING	RNPAR	SSAVE	VIEWPORT W
ON TIME	PRINT#	ROTATE	START	VOL\$
ON TIMEOUT	PRINTER IS	RUN	STATUS	VOLUMES
ON TIMER	PROG		STEP	
ON UNIT	PSET		STOP	
OPEN	PU	SAVE	STR\$	WAIT
OPTION BASE	PUT#	SCRATCH	SUB	WHILE
OR		SCRATCHP	SUBCOM	WINDOW
OUTPUT		SCRATCHV	SUBEND	WRITE IO
	QUIT	SEND	SUBEXIT	
		SEQACTV	SWAP	
PAINT		SET BLINK		
PAUSE	RANDOMIZE	SET CHAR		
PEEK	READ	SET LINE	TAB	

TAP0202.EPS

Appendix 3. Listing of MS-DOS Special Editing Functions

Table 3.1 lists the MS-DOS (DOS/V) special editing functions. The term “template” here refers to a special area reserved by MS-DOS (DOS/V) where you can temporarily store data, for example.

Press the return (↵) key after typing a character string following the prompt (BSC, bsc or >). The data of the typed character string is stored in a template. For an EDIT command or statements, the character strings of their execution results are supported in terms of template storage. Only character strings of no more than 252 bytes are included in template storage. If you specify a character string of greater than 252 bytes, characters of the extra bytes are excluded from template storage and therefore are discarded. The data in the template is overwritten each time you press the ↵ key.

Table 3.1 Special Editing Functions

Key	Editing Function	
[F1] [→]	COPY 1	Copies one character from the template to the command line.
[F2]	COPY UP	Copies characters of up to one immediately preceding the specified character from the template to the command line.
[F3]	COPY ALL	Copies all characters remaining in the template from the template to a command line.
[F4]	SKIP UP	Skips (excludes from copying) the template's characters of up to one immediately preceding the specified character.
[F5]	NEW LINE	Copies data in the command line to the template (creates a new template if the ↵ key is not pressed).
[↓]	VOID	Cancels the current entry in the command line and feeds a line. Data in the template is not updated, however.
[Ins]	INSERT MODE	Places the system in the insert mode.
[Del]	SKIP 1	Skips (excludes from copying) one character among characters in the template.

TAP0301.EPS

Note that these editing functions are case-sensitive and any lower-case letter is not converted to an upper-case letter. If you delete a line by specifying the line number only, the template is supplied with the code “DEL _ (line number).” If you view the program with a LIST command, the template is supplied with the data of the program's last line.

Revision History

Edition	Date	Revised Item
1st	Oct. 1999	New publication

Written by Product Marketing Section, PLC Center
 Industrial Automation Business Head Quarter.
 Yokogawa Electric Corporation
Published by Yokogawa Electric Corporation
 2-9-32 Nakacho, Musashino-shi, Tokyo 180-8750, JAPAN
Printed by Yokogawa Graphic Arts Co., Ltd.
