Chapter 3 Programming

3-1 Programming Method and Tools

Programming is performed by connecting a Windows PC through a connection cable. Software is publicized on the net. (Downloadable for free.)

Hardware

▪ PC	Windows PC (USB is available with W2K or later.)
▪ Connection (programming) cable	USB-RS (USB-SERIAL conversion) Or Cable DOS/V (DSUB 9 pin RS-232C cable)

Software

FTMW
Terminal software
Connects to MPC to perform editing, debugging, and reading/storing from a PC. This is an indispensable application for MPC development.
The main text applies to Ver. 6.38z or later.
(Although the file name is “FTMW32.EXE”, it is called “FTMW” in the main text.)

MPCED
Offline editor
An offline editor dedicated for MPC. Color-codes control statements, labels, and comments.

SYSLD2000
System loader
Used for version-upgrading MPC. A system loader inside a flash ROM is rewritten.

F2KCheck
Simple program checker
Checks the correspondence of IF ~ END_IF and DO ~ LOOP which tends to become hard to keep, duplicate labels, and the like.

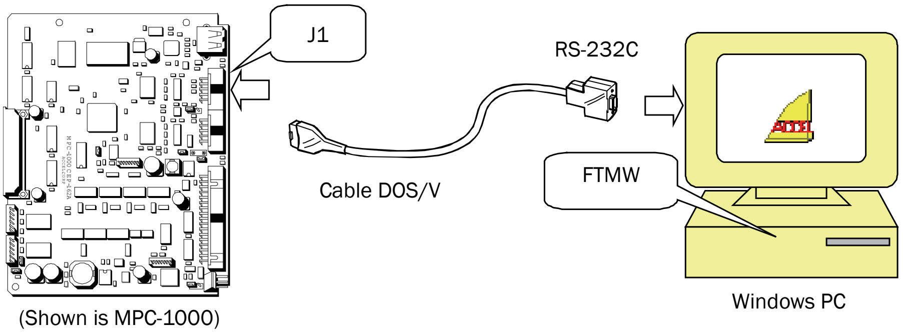
CUMON
CUnet monitor
A tool for confirming and changing CUnet global memory and checking CUnet mail sending and receiving.

ACTERM
RS-232C general use terminal software
Used for debugging of MPC communication programs, checking the operation of connected equipment, and the like.

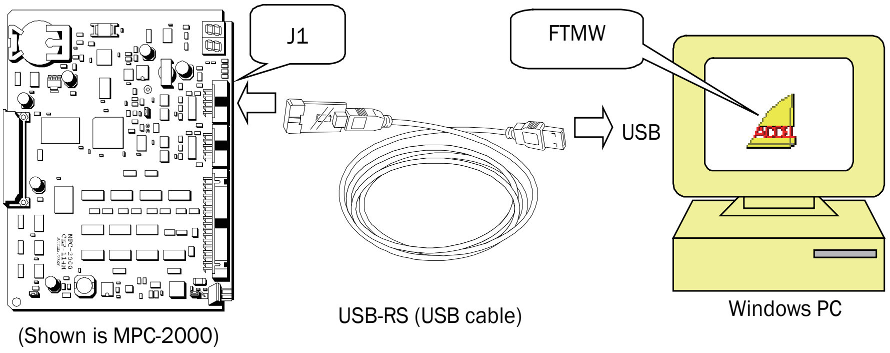
- The MPC development environment is installed with a Accel_Setup_eng.msi
- The Accel_Setup_eng.msi can be downloaded free from our company’s home page.
- The Accel_Setup_eng.msi should be used for the first installation. Later updating can be performed by replacing executive files (*.EXE). (The most recent versions can be downloaded from the web.)
- The default setup folder is C:\Program Files\ACCEL.

3-2 Connection between MPC and PC

■ When using the standard COM port of PC (connection via Cable DOS/V)

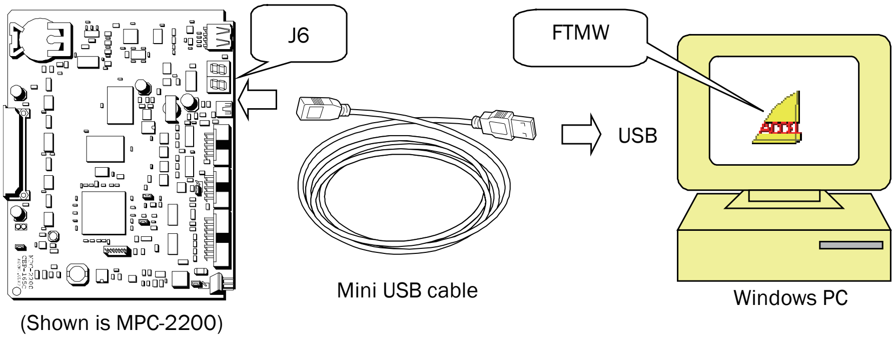


■ When using a USB port of PC (an example of connection via USB-RS)



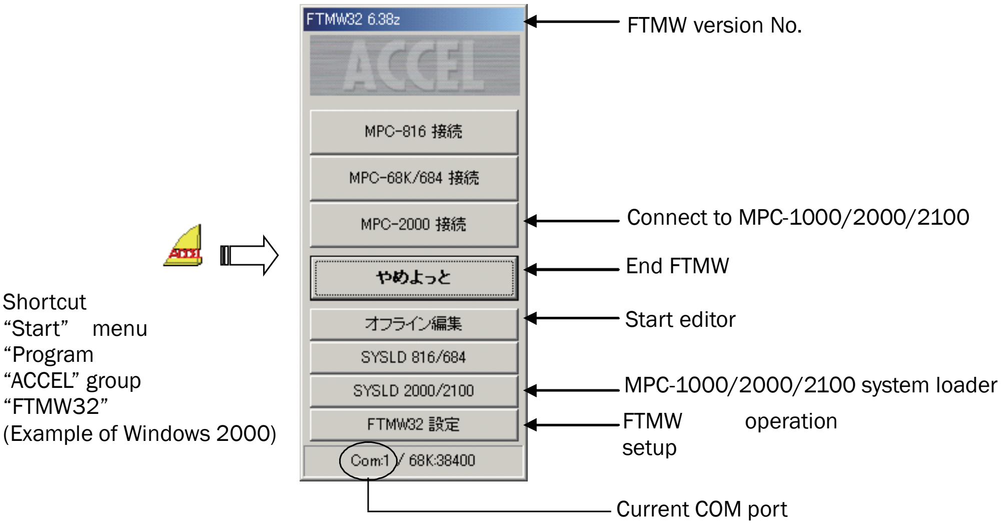
* The use of USB requires installation of a device driver.

■ Case of MPC-2200



3-3 Starting FTMW

1) Click on FTMW shortcut icon. The following window appears.



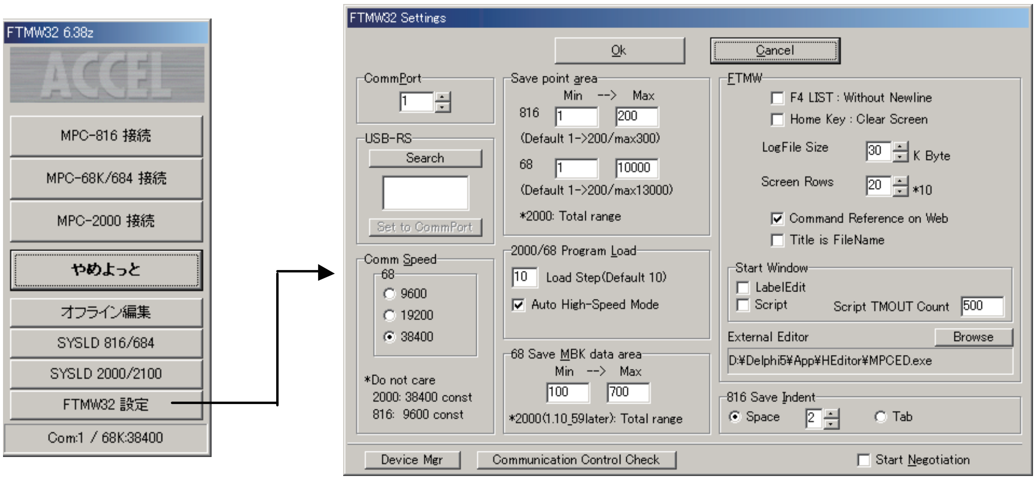
■ "FTMW32 setup" screen

* Communication Port:

When using the standard COM port of a PC, it should be one of 1~4.
When using USB-RS, it can be detected by "USB-RS" > "Search"
If the port number is unknown, start the device management by "Device Mgr" and check it by "Port (COM and LPT)". The communication speed of MPC-2000 is fixed to 38400 bps.

* Communication Control Check:

Checks whether the communication control installed in Windows is the Japanese version or the English version. The Japanese version of communication control must be installed on Japanese Windows. If the English version is set up by mistake and the Japanese version is set up again, it should be checked (requires ccc.exe.).



CommPort: 1

USB-RS

Comm Speed
68
○ 9600
○ 19200
● 38400
*Do not care
2000: 38400 const
816: 9600 const

Save point area
Min --> Max
816 1 200
(Default 1->200/max300)
68 1 10000
(Default 1->200/max13000)
*2000: Total range

2000/68 Program Load
10 Load Step(Default 10)
☑ Auto High-Speed Mode

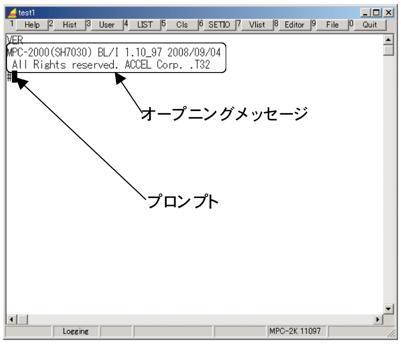
68 Save MBK data area
Min --> Max
100 700
*2000(1.10_59later): Total range

FTMW
□ F4 LIST : Without Newline
□ Home Key : Clear Screen
LogFile Size 30 K Byte
Screen Rows 20 *10
☑ Command Reference on Web
□ Title is FileName
Start Window
□ LabelEdit
□ Script Script TMOUT Count 500
External Editor
D:¥Delphi5¥App¥HEditor¥MPCED.exe

816 Save Indent
● Space 2 ○ Tab

□ Start Negotiation

2) Turn on the MPC power and press the “MPC-2000 connection” button.
Connection is normal if an opening message is displayed as a prompt on an editing screen.



```
VER
MPC-2000(SH7030) BL/I 1.10_97 2008/09/04
 All Rights reserved. ACCEL Corp. .T32
#
```

■ Meaning of the opening message
This message is displayed at the time of FTMW connection or by VER command.

- Case of MPC-2000

```
MPC-2000*(SH7030) BL/I 1.12_91 2012/01/30
 All Rights reserved. ACCEL Corp. .T32
```

- Case of MPC-2200

```
MPC-2200L(SH7211) BL/I 1.12_91 2012/01/30
 All Rights reserved. ACCEL Corp. .T32
```

- Case of MPC-2200 with USB port ON

```
#on_usb
#ver
MPC-2200L(SH7211) BL/I 1.12_91 2012/01/30
 All Rights reserved. ACCEL Corp. .T32
  +The USB Activated on TASK_29+
```

3-4 Command Input

If a command is entered after the prompt and Enter is pressed, it is instantly executed, in what is known as direct command execution. Although the majority of commands can be either executed directly or stated in a program, there are commands such as those for maintenance and editing which can be used only as a direct command, and commands such as control statements which can be written only in a program.

Usable in both	Direct only	Program only
ON 0 OFF 0 PRINT A MOV L and the like.	LIST MPCINIT ERASE RUN and the like.	GOTO GOSUB IF ~ FOR ~ NEXT and the like.

```
#ON 0<Enter>            /* Direct execution.  10 ON 0<Enter> makes it a program.
#GOTO 100<Enter>        /* Even if this is directly executed, nothing will happen.
#10 MPCINIT<Enter>      /* If this command is programmed, the program will disappear.
```

* <Enter> in this text indicates pressing down the Enter key of PC keyboard.

■ Execution example with a training kit (XY03)

```
#ON 0                 /* Front panel green LED lit
#ON 1                 /* Yellow LED lit
#PRINT SW(192)        /* Green SW status check
0                     /* 0 = OFF
#PR SW(192)         /* Execute while pressing Green SW (PR is an abbreviation of PRINT)
1                     /* 1 = ON
#PR IN(24)            /* Parallel input with DSW set to '3'
48
#PRX IN(24)           /* PRX is displayed in HEX 10 (Dec) = 30 (Hex)
00000030
#SETIO                /* Output all OFF
#
```

3-5 Program Editing in FTMW

Explained here are operations which are frequently used in editing a program currently stored in MPC by FTMW.

LIST display

The most frequently used is LIST command.

```
  * Format
 LIST [arg1 arg2]
  arg1: Start statement number or start label
  arg2: Number of lines to display
```

- LIST can be executed without any argument (① below). In that case, a continuation of the last time is displayed.
- The display start position can be specified as the first argument in the statement number of label (② and③ below).
- The number of lines to display can be specified as the second argument (④ below). Thereafter, this number of lines is held.
- LIST 0 will display from the beginning (⑤below).

①
```
#LIST   /* Start position, number of lines not specified
10      GOSUB   *READ_DSW
20      _RET_VAL   D
30      PRINT   D
40      END
50     *READ_DSW
#
```

②
```
#LIST 40   /* Start position = statement number specified
40      END
50     *READ_DSW
60      DSW_=IN(24)/16
70      RETURN   DSW_
#
```

③
```
#LIST *READ_DSW   /*  Start position = label specified
50     *READ_DSW
60      DSW_=IN(24)/16
70      RETURN   DSW_
#
```

④
```
#LIST *READ_DSW 2
          /*  Start position = label specified, number of
                lines to display specified
50     *READ_DSW
60      DSW_=IN(24)/16
#
```

⑤
```
#LIST 0 5
          /* Start position = top, number of lines to
              display specified
10       GOSUB   *READ_DSW
20       _RET_VAL   D
30       PRINT   D
        40       END
50     *READ_DSW
#
```

Inserting a line

```
10        *LOOP
20        ON 0
30        OFF 0
40        GOTO *LOOP
```

Move the cursor to Step 30, then Ctrl+N

```
10        *LOOP
20        ON 0
25        TIME 50<Enter>
30        OFF 0
40        GOTO *LOOP
```

```
10        *LOOP
20        ON 0
30        OFF 0
40        GOTO *LOOP
25 TIME 50<Enter>
```

Describe with the statement number to insert attached.

```
LIST
10        *LOOP
20        ON 0
25        TIME 50
30        OFF 0
40        GOTO *LOOP
```

Deleting a line

```
LIST
10        *LOOP
20        ON 0
25        TIME 50
30        OFF 0
40        GOTO *LOOP
```

Move the cursor to a line to be deleted, then Ctrl+Y

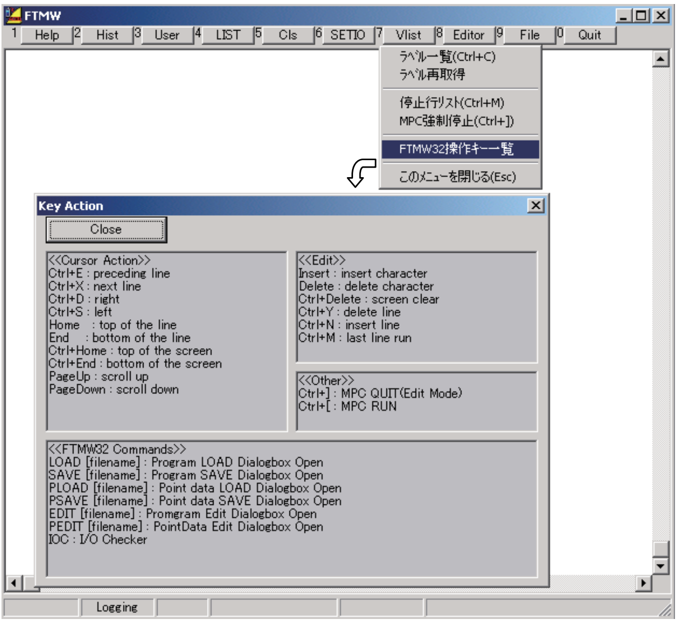
```
LIST
10        *LOOP
20        ON 0
30        OFF 0
40        GOTO *LOOP
```

```
LIST 0
10        *LOOP
20        ON 0
25        TIME 50
30        OFF 0
40        GOTO *LOOP
25<Enter>
```

Input only the statement number to be deleted.

```
LIST
10        *LOOP
20        ON 0
30        OFF 0
40        GOTO *LOOP
```

Other key operations



```
<<Cursor Action>>
Ctrl+E : preceding line
Ctrl+X : next line
Ctrl+D : right
Ctrl+S : left
Home   : top of the line
End    : bottom of the line
Ctrl+Home : top of the screen
Ctrl+End : bottom of the screen
PageUp : scroll up
PageDown : scroll down
```

```
<<Edit>>
Insert : insert character
Delete : delete character
Ctrl+Delete : screen clear
Ctrl+Y : delete line
Ctrl+N : insert line
Ctrl+M : last line run
```

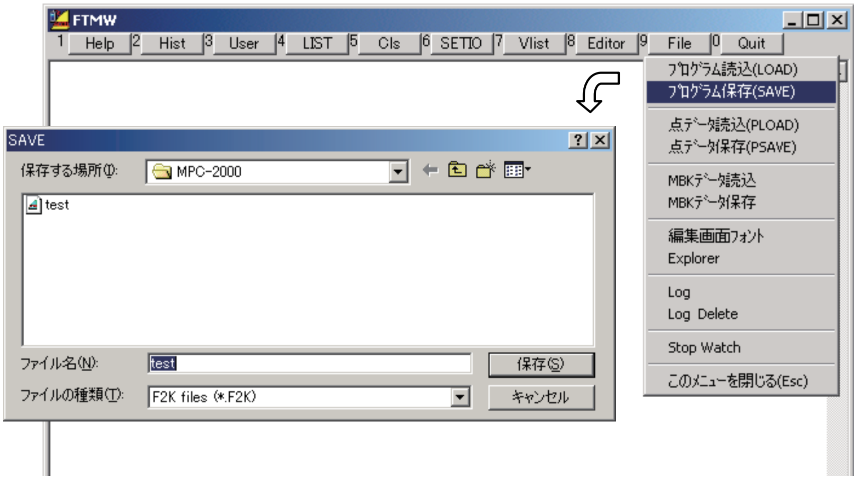
```
<<Other>>
Ctrl+] : MPC QUIT(Edit Mode)
Ctrl+[ : MPC RUN
```

```
<<FTMW32 Commands>>
LOAD [filename] : Program LOAD Dialogbox Open
SAVE [filename] : Program SAVE Dialogbox Open
PLOAD [filename] : Point data LOAD Dialogbox Open
PSAVE [filename] : Point data SAVE Dialogbox Open
EDIT [filename] : Promgram Edit Dialogbox Open
PEDIT [filename] : PointData Edit Dialogbox Open
IOC : I/O Checker
```

3-6 Saving and Loading a Program

Saving

The F9 “Save program” saves the program in a PC, with the extension ‘F2K’. The saved program does not have statement numbers.



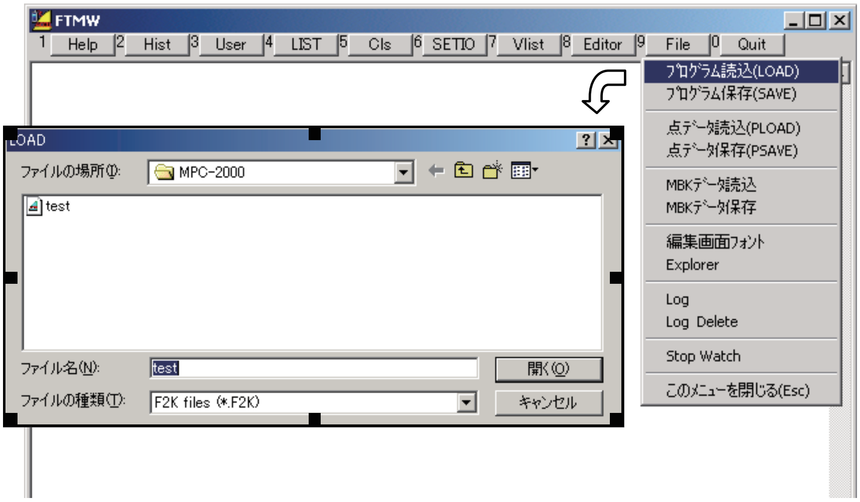
保存する場所(I): MPC-2000

ファイル名(N): test

ファイルの種類(T): F2K files (*.F2K)

Loading

The F9 “Load program” loads a program from a PC.
Statement numbers have intervals of 10 in the initial condition. If 60000 steps is exceeded, they are automatically renumbered at intervals of 5.



ファイルの場所(I): MPC-2000

ファイル名(N): test

ファイルの種類(T): F2K files (*.F2K)

Offline creation

- Because FTMW directly operates on data in MPC, it cannot be used without connection. However, programs can be created offline.
- An editor should be prepared. Although MPCED is dedicated for MPC, a general-use editor or a word processor can be used. If so, the program is saved as text data with the extension “F2K”.
- Program errors cannot be detected until loaded to MPC and executed.

Printing

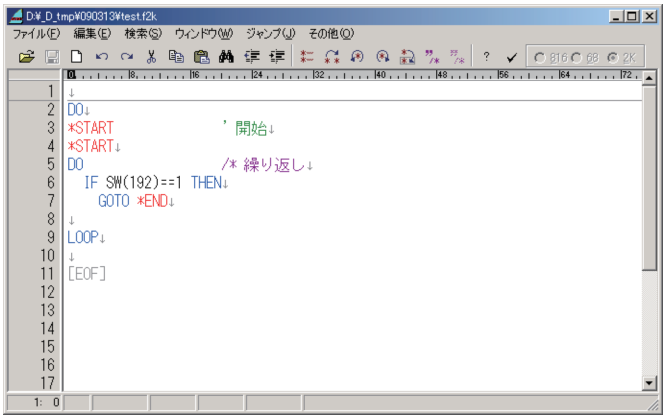
FTMW has no printing function. Use should be made of an editor such as MPCED or word processor software to print files saved in PC.

3-7 Offline Program Creation/Editing

- Because FTMW is used while connected to MPC, a program saved in a PC cannot be edited or created. An editor is used for offline programming. The editor is generally known as a text editor, which include commercially marketed products, freeware, and Windows Notepad.

- “MPCED” is prepared as an editor dedicated for MPC, and has such functions as color-coding statements, label jumping, and batch commenting/uncommenting.

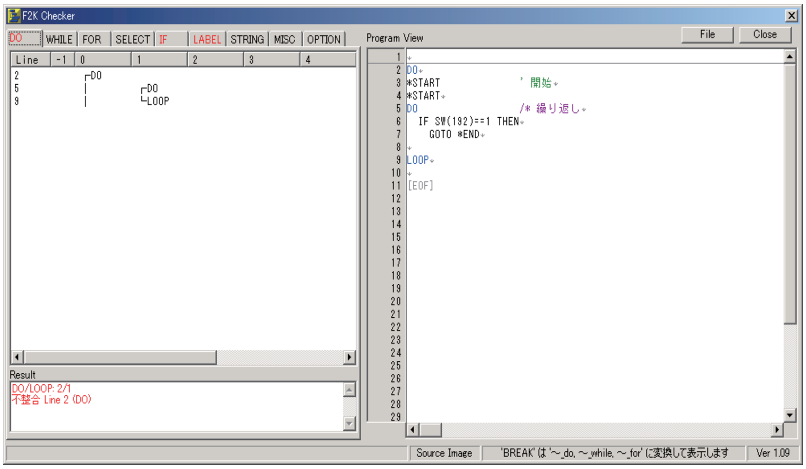
- In an Internet-connected environment, the command reference on the MPC-2000 site can be directly referred to by placing the cursor on a command and pressing the F1 key (in the same manner as referring to HELP).



```
↓
DO↓
*START              ' 開始↓
*START↓
DO                  /* 繰り返し↓
  IF SW(192)==1 THEN↓
    GOTO *END↓
↓
LOOP↓
↓
[EOF]
```

- As a program becomes larger, the correspondence of DO ~ LOOP, IF ~ END_IF, and the like tend to become difficult to tell. In such a case, try using a simple checker “F2KCheck”. It starts by the Check button of MPCED.

For example, the program above does not have correct correspondence for DO ~ LOOP and IF ~ END_IF. This is viewed with Checker as follows. A red tab indicates a warning.



```
Line  -1  0        1        2   3   4
2         ┌DO
5         |        ┌DO
9         |        └LOOP
```

Program View:

```
↓
DO↓
*START              ' 開始↓
*START↓
DO                  /* 繰り返し↓
  IF SW(192)==1 THEN↓
    GOTO *END↓
↓
LOOP↓
↓
[EOF]
```

Result

```
DO/LOOP: 2/1
不整合 Line 2 (DO)
```

'BREAK' は '~_do, ~_while, ~_for' に変換して表示します

* These are based on simple number matching. Normal behaviors of the program are not guaranteed.

3-8 Initialization

The initial setup and runtime parameters of MPC are stored in flash ROM and S-RAM. Malfunctions occur if the parameters are disturbed by trial-and-error activities in development or the application of static electricity during transportation (especially as a stand-alone board), Regular initialization should be performed in the following cases.

* When a board has been transported alone.
It may be damaged during transport.
If a program is stored in a standalone board when transported for maintenance, for example, beware of static electricity, short-circuiting or dropping a battery, damaging the parts, condensation, and the like. Be sure to use an antistatic bag.

* If a malfunction occurs while debugging.
If something has gone wrong while doing many different things.
If a program does not run although it has nothing wrong. (The possibility of a bug should also be pursued.)
Etc.

* After updating the system.

■ Initialization commands

MPCINIT	Initializing the RAM (switches to the English mode)
ERASE	Erasing the flash ROM

* These two should be executed as direct commands.

ENG	Switches to the English mode.

Execution example:

```
#MPCINIT
#ERASE
*
#ENG
#
```

■ Precautions in initialization

Initialization will clear the program, point data, and variables.

They should be saved in the PC or recorded if necessary.

3-9 I/O Check

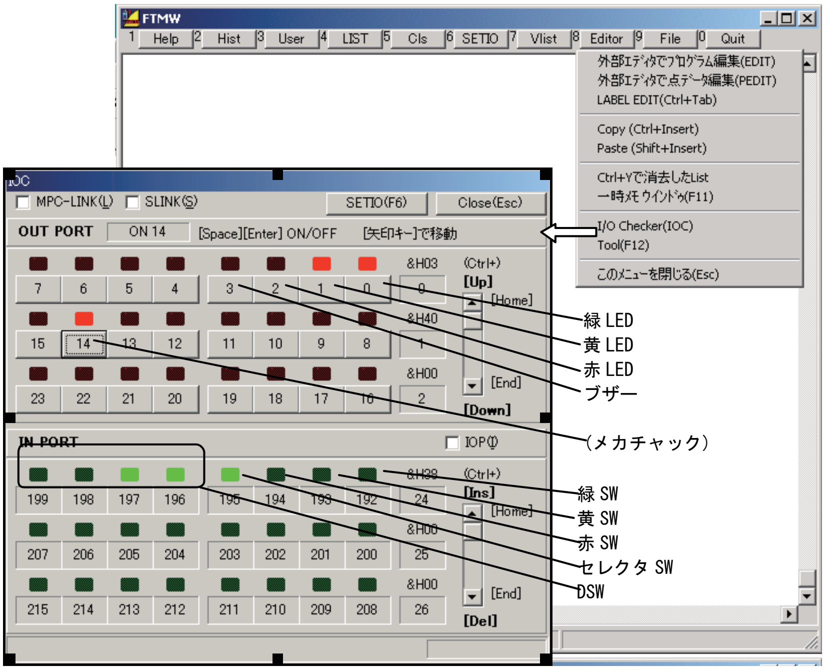
Checking by commands

These are examples of I/O checks by direct commands.

```
#ON 0           /* Front panel Green LED lit
#ON 1           /* Yellow LED lit
#PRINT SW(192)  /* Green SW status check
0               /* 0 = OFF
#PR SW(192)     /* Execute while pressing Green SW (PR is an abbreviation of PRINT)
1               /* 1 = ON
#PR SW(195)     /* Selector SW on the left side
0
#PR SW(195)     /* Selector SW on the right side
1
#PR IN(24)      /* Parallel input with DSW set to '3'
56
#PRX IN(24)     /* PRX is displayed in HEX 56 (Dec) = 38 (Hex)
00000038
#SETIO          /* Output all OFF
```

Checking by I/O Checker

For viewing all in one screen, start I/O Checker by pressing [F8] I/O Checker.
Typing IOC<Enter> also starts I/O Checker.



3-10 Language Specification

Integer BASIC

The MPC-2000 language is multitasking integer BASIC. Although there is no multitasking in normal BASIC, BL/1 (Basic Language 1) of MPC-2000 can simultaneously execute up to 32 programs through time-sharing processing, and can handle complex actions of devices.

As variables, 4-byte integer is set as the standard.
In the same manner as in the normal BASIC, variables can be used without defining them.
Variable labels must be within 15 characters. Floating-point operations necessary for measurements and the like are realized through the FLOAT command. Only integers are ordinarily used to prevent reduction in the processing speed and ambiguity in processing.

Although floating-point operations can deal with a wide range of numerical values, in some programs unintended errors may accumulate and malfunctions may occur as a result.

Multi-statements

BL/1 consists of a command or formula in each line. Although the execution unit is one line, multiple commands can be described in one line by delimiting the commands with a ':' (colon).

```
ON   1 : TIME   100 : OFF   1 : TIME   100
```

The description of an IF statement can also be in one line by limiting with ':'.

```
IF   i%10==1 THEN
 USB_DEL   FILE$
 END_IF     →          IF   i%10==1 THEN  : USB_DEL   FILE$ : END_IF
```

Comments

A string array starting with a ‘ (single quote) becomes a comment and will not be executed. However, the interpreter consumes a slight amount of time even with a comment.

Try to use a ‘ (single quote) comment in an unexecuted section (such as the top of a program).

Multitasking

In actual device control scenes, various actuators must be concurrently controlled. However, in the general software language, multitasking requires a complicated procedure and even if workable, the processing speed is generally slow. This is because multi-tasking was developed in the field of information processing in general and assumes its operation in a large-scale system.

On the other hand, the multitasking in BL/1 is specialized in device control and assumes its operation in a small-scale system. Therefore, it is designed to be used through very simple procedures.

1) FORK

In a normal BASIC-type interpreter, a program is run by sequentially executing a command list. There is only one order of execution. For example, commands in the range of 10~30 continuously repeated in the following example.

```
10     DO
20      ON   1 : TIME   100 : OFF   1 : TIME   100
30     LOOP
```

However, in MPC-2000 multiple execution paths can be started by the FORK command.
FORK is a command corresponding to RUN and means that RUN *TASK is executed as Task 1.
Tasks 1~31 can be specified, and 31 programs can be simultaneously executed.
The program executed first by RUN becomes Task 0.

```
10     FORK   1 *TASK
20     DO
30      ON   1 : TIME   100 : OFF   1 : TIME   100
40     LOOP
50    *TASK
60     DO
70      ON   2 : TIME   100 : OFF   2 : TIME   100
80     LOOP
```

2) Task management

A task whose execution has already begun can be stopped in the middle or resumed.

QUIT n	Command to quit a task.
PASUE n	Command to pause a task.
CONT n	Command to resume a paused task.

In addition, there may be needs to obtain the status of another task or one’s own task number.

TASK()	Check whether another task is being executed or stopped.
TASKn	One’s own task number can be always obtained with a reserved variable.

3) Semaphore

The most difficult multitasking in an actual scene uses one actuator or output by multiple tasks. For example, two tasks output character strings to RS-232C CH1 in the following example.

```
10     FORK   1 *TASK1
20     FORK   2 *TASK2
30     END
40    *TASK1
50     DO
```

```
60       FOR   i=&h0041 TO &h004A
70        PRINT#   CHR$(i)
80       NEXT
90       PRINT#   "\r\n"
100       TIME   500
110      LOOP
120    *TASK2
130      DO
140       FOR   j=&h0030 TO &h0039
150        PRINT#   CHR$(j)
160       NEXT
170       PRINT#   "\r\n"
180       TIME   500
190      LOOP
```

The result becomes as follows, wherein outputs from two tasks are intermingled.

```
[RS-232C output]
ABCDEFGH0123456IJ
789
ABCDEFGH0123456IJ
789
```

Then, an interlock between tasks known as a semaphore is added. Examples are WAIT ON(-1) and OFF -1 in the following program.

```
10       FORK   1 *TASK1
20       FORK   2 *TASK2
30       END
40     *TASK1
50       DO
55        WAIT   ON(-1)
60        FOR   i=&h0041 TO &h004A
70         PRINT#   CHR$(i)
80        NEXT
90        PRINT#   "\r\n"
95        OFF   -1
100        TIME   500
110       LOOP
120     *TASK2
130       DO
135        WAIT   ON(-1)
140        FOR   j=&h0030 TO &h0039
150         PRINT#   CHR$(j)
160        NEXT
170        PRINT#   "\r\n"
175        OFF   -1
180        TIME   500
190       LOOP
#
```

The result is an organized output as follows.

```
[RS-232C]
0123456789
ABCDEFGHIJ
0123456789
ABCDEFGHIJ
0123456789
```

A semaphore means a wooden-bar signal which was used for railroads to prevent collisions. If tasks are likened to multiple railroads, a semaphore (wooden-bar signal) prevents trains from colliding at an intersection of railroads (tasks). Although memory I/Os such as WAIT ON(-1) are used as semaphore, any output port may do if ON() function covers the I/O area.

4) SWAP command

When a program is executed and stopped by CTRL_A, the following display may be output.

```
        *0!   [20]
       ! is a time-wasting task.
```

Indicated after the task number is that ! is wasting time.
Multitasking may look to human eyes as if multiple programs are running at the same time, for the CPU it is simply executing tasks sequentially by time-sharing. BL/1 adopts a simple time-sharing multitasking called the round-robin scheme. Each task is switched at every 3 msec. However, if there is a condition-wait command such as TIME, WAIT, and SW(), the task is forced to be switched. It is because if the condition is not met, executing that task would be a waste of time.
Among condition-waiting, the following program generates a waste of time.
Therefore, if a is 0, forced task switching should be generated.

```
10      DO
20       IF   a==1 THEN  : BREAK   : END_IF
30      LOOP
#run

        *0!   [20]
      ! is a time-wasting task.
#
```

For this, SWAP command should be added as follows.

```
10      DO
20       IF   a==1 THEN  : BREAK   : END_IF
25       SWAP
30      LOOP
#run

        *0    [25]
#
```

SWAP command is a command to generate a forced task switching.
Device control is a collection of processes which take certain actions if some conditions are met. Therefore, unless those conditions are met, nothing is performed, in which case SWAP is added to suppress a waste of time. Alternatively, if more time is expected to pass until all the conditions are met, of if there is no need to respond at a high speed in the first place, a timer command such as TIME 100 is used instead of SWAP. TIME also performs forced task switching, and further has the task sleep for a specified time. Time not spent by
that particular task is effectively used by other things or tasks.

Debugging

1) BREAK_POINT

In BL/1 a program can be stopped at up to eight specified statement numbers by a BREAK_POINT command. (Label specification is also possible.)
If a program number is specified as follows, the specified line is displayed.
Afterwards, the statement number of the specified line is displayed in reverse.
As break points, statement numbers are specified in order. Releasing a specified statement number is done by inputting the same number. To check which statement numbers are registered, execute BKP command with no argument. In addition, to release all break points, enter BKP 0.

```
30      FORK   2 *bb
40      END
110    *bb
120      DO
130       FOR   i_=8 TO 15
```

```
140       ON  i_ : TIME  50 : OFF  i_150      NEXT
160      LOOP
#bkp 110 140

110    *bb
140      ON  i_ : TIME  50 : OFF  i_
#bkp
BREAK_POINT 0=110
BREAK_POINT 1=140
#bkp 110

110    *bb
#bkp
BREAK_POINT 0=140
#
```

①When break points are actually specified and RUN is executed, execution is stopped at specified points.
Then, the line where it is stopped and the task number are displayed. By pressing n<ENTER> execution is resumed up to the next break point. In this program, a break occurs each time the statement number 30 is passed (before execution).

②To perform a stepwise forwarding (continual line-by-line execution), press t<ENTER>.
To release the stepwise forwarding, press the <ENTER> key.

③While stopped by a break, the value of a variable or function can be referred to.
Press ‘p’ and subsequently enter the variable name or function name.

④Break points can also be added.
Press ‘b’ and enter a statement number to add a break point.

⑤To release a break point while in a break, enter “u”.

⑥To end program execution, press ‘e’.

```
     #list *aa
     50     *aa
     60      DO
     70       FOR  i_=0 TO 7
     80        ON  i_ : TIME  200 : OFF  i_
     90       NEXT
     100      LOOP
     #bkp 100

     100     LOOP
     #run *aa
     50-##
①   100      LOOP      <00>
②   #t
     60        DO       <00>
     #t
     70        FOR  i_=0 TO 7    <00>
     #t
     80        ON  i_ : TIME  200 : OFF  i_    <00>
③   ?pi_
     #PR i_-> 0
     #
     100      LOOP      <00>
     ?b80
④   #BKP 80->
     80       ON  i_ : TIME  200 : OFF  i_
     #
```

```
   80       ON   i_ : TIME   200 : OFF   i_    <00>
   ?p i_
   #PR  i_-> 0
   #
   80       ON   i_ : TIME   200 : OFF   i_    <00>
   #
   80       ON   i_ : TIME   200 : OFF   i_    <00>
   ?p i_
   #PR  i_-> 2
   #
   80       ON   i_ : TIME   200 : OFF   i_    <00>
⑤ ?u
   #
   100      LOOP      <00>
   #
   100      LOOP      <00>
⑥ ?e
   ##
```

2) If a FOR statement is set as a break point, ...

This is a precaution for a case where 20 is set as a break point in the following program. The execution order becomes 20 -> 30 -> 40 -> 30 -> 40 -> 30 -> 40 -> 40, and the FOR statement is executed only once in the FOR loop.

```
10      DO
20       FOR   i=1 TO 3
30        PRINT   i
40       NEXT
50      LOOP
```

This is because the FOR statement includes an initialization formula. At the time of the initial compilation, the system embeds a place after TO of the FOR statement for the NEXT statement. The NEXT statement evaluates the limit value and the STEP value of the FOR statement at every execution, and if it is looped, moves the control to immediately after the FOR statement. Therefore, the FOR statement itself is not executed in the loop.

3) BREAK_POINT in multitasking

In a program such as the one below, break points can be set after execution.
Once set, the break points become immediately effective. In other words, debugging can be started with a program under execution. The rest of its usage is the same as in single tasking.

```
LIST
10     AAA=111 :  B=123
20      FORK   1 *aa
30      FORK   2 *bb
40      END
50     *aa
60      DO
70       FOR   i_=0 TO 7
80        ON   i_ : TIME   200 : OFF   i_
90       NEXT
100      LOOP
110     *bb
120      DO
130       FOR   i_=8 TO 15
140        ON   i_ : TIME   50 : OFF   i_
150       NEXT
160      LOOP
#run
#bkp 80
80      ON   i_ : TIME   200 : OFF   i_
##
```

```
80       ON   i_ : TIME   200 : OFF   i_    <01>
#
80       ON   i_ : TIME   200 : OFF   i_    <01>
#t
90      NEXT       <01>
#t
80       ON   i_ : TIME   200 : OFF   i_    <01>
#t
90      NEXT       <01>
?p   i_
#PR i_-> 2
```

Below is a case where break points are set in different tasks. While in a break, a break of another task occurs, a waiting state is entered. Therefore, if <ENTER> execution is repeated, alternate break processes of Task 1 and Task 2 occur.
In each break, if the value of i_ is referred to, each task shows a different value.
In addition, if 'u' is entered in the middle, break number 80 is released, and afterwards only Task 2 will have breaks.

```
#bkp 80 140

80      ON   i_ : TIME   200 : OFF   i_

140      ON   i_ : TIME   50 : OFF   i_
##
80       ON   i_ : TIME   200 : OFF   i_    <01>
#
140       ON   i_ : TIME   50 : OFF   i_    <02>
#
80       ON   i_ : TIME   200 : OFF   i_    <01>
#
140       ON   i_ : TIME   50 : OFF   i_    <02>
#
80       ON   i_ : TIME   200 : OFF   i_    <01>
?p i_
#PR  i_-> 6
#
140       ON   i_ : TIME   50 : OFF   i_    <02>
?p i_
#PR  i_-> 15
#
80       ON   i_ : TIME   200 : OFF   i_    <01>
?u
#
140       ON   i_ : TIME   50 : OFF   i_    <02>
#
140       ON   i_ : TIME   50 : OFF   i_    <02>
?
```

4) SLOW_RUN

Operating a device for the first time requires considerable precaution. In such a case, SLOW_RUN slows down the execution speed of a program. For example, the following command inserts a timer of 1000 msec at every line in executing Task 10.

```
SLOW_RUN 10 1000
```

The arguments are the task number and wait time for each line. A maximum of 4000 msec can be specified.

Some programs such as WS0() and WS1() have a time-out function. If the execution speed of a Specific task is slowed down by SLOW_RUN, a trouble occurs in debugging due to time-out. In such a case, the following should be executed.

```
SLOW_RUN TMOUT
```

By this, the time-out time is multiplied by a factor of 10. If further room is necessary, adding an argument of 10000 multiplies it by a factor of 100. This is because the down count timer which is subtracted at every 100 msed is set to be subtracted at every 10000 msec, or 10 seconds.

```
SLOW_RUN TMOUT  10000
```

Global variables and task-local variables

Roughly speaking, BL/1 has two kinds of variables; global variables and task-local variables. Global variables are variables which can be used anywhere by any task. They can be regarded as normal variables. Task-local variables are variables unique to BL/1 and task different values in different tasks.

An example is given below. A variable such as port_ given a '_' code at the end becomes a task-local variable. In the example below, a subroutine *ON_PORT turns on a different port for each task which calls it.
When this subroutine is simultaneously used by multiple tasks, if port_ were a normal global variable, multiple tasks would end up using the same variable at the same time, which would make the processes unstable.

Because port_ can have independent values among different tasks, such process conflict due to sharing the same variable will disappear.

```
*ON_PORT
  port_=X(TASKn)
  ON port_
 RETURN
```

However, task-local variables are difficult to monitor in debugging, regarding the kind of values they have. Even if

```
print port_
```

is executed after stopping a program, only the value of port_ of an executed task, namely task 0 can be referred to. In order to solve this, BL/1 prepares pra command as follows.

```
pra port_
```

Although the pra command is usually used to display a list of array variable elements, it displays values by tasks for a task-local variable.

Reserved constants and reserved variables

BL/1 has reserved constants and reserved variables which are prepared in advance. Registered as the reserved constants are numerical values which are fixed by the system for use.

The following can be listed as an example. X-A has a value of &H80000001 and specifies the X-axis in RMVS command. In this manner, reserved constants are prepared so that command functions can be more efficiently described.

```
RMVS X_A 1000
```

There are a considerable number of reserved constants, and their use varies widely depending on commands. Refer to such information by searching with condition narrowed as [Group] -> [Reserved Constants] in the Command Reference on the web.
Reserved variables are variables such as TASKn and SYSCLK which are constantly updated by the system. Presently, there are the following reserved variables and constants.

[Reserved Variables]

Global Variables	Use
SYSCLK	Automatically incremented at every 1 msec. CPU clock reference.
TASKn	A variable which returns the self task number.
SEC	Automatically incremented at every 1 second.

PG_TASK0	A variable which returns the PG number assigned to Task 0.If the PG does not exist, -1 is returned.
MBK_ERR	The number of errors of MEWNET communication.
MBK_CMD	Command which could not be processed by MEWNET communication.If prx MBK_CMD returns 4142, it means AB.
VER$	Version character string. For the version number, see MBK(8053).
CUM_PNT CUM_SRC CUM_NUM CUM_CNT CUM_ERR CUM_TASK	Used by CUnet CU_POST.
FILE$,FILE$1,FILE$2	Used by USB memory command USB_WRITE.
CHK_SUM	Program check sum. Checking if it is the same program immediately after loading it.
V_PGA,V_PGB	Return value of MPC-1000. Current position, version, etc.
Task Variables	
timer_	Time-out processing. It is down-counted at every 0.1 second and stops at 0.
ptr_	Character string processing. Character string pointer.
rse_	Communication error status.
err_	Information on an error while using ON-ERROR.

* Although there are other reserved variables, they can be freely used if no functional conflict exists.

[Reserved Constants] "Those displayed as a list of constants" by Vlist

Data Type Specification			
Lng	Touch panel I/O	Long type (2 words) specification	S_MBK,OUT
Wrd		Word type specification	
Int		Word type specification (signed)	MBK()
NIL		0. Used to explicitly indicate 0.	
PG interrupt setup			
CMP_PLS	MPG-2314	Compare current pulse counter and COMP+.	INSET
CMP_CNT		Compare encoder counter and COMP+.	
C_MORE		Interrupt if counter >= COMP+.	INTA_ON/_OFF
C_LESS		Interrupt if counter < COMP+.	INTB_ON/_OFF
CUnet			
SA0~SA15	MPC-Cunet	I/O number corresponding to CUnet station address	ON/OFF SW()
SA0_B ~SA15_B		I/O bank number corresponding to CUnet station address	IN/OUT
Communication			
EOL	Serial communication	Receiving terminator setup	INPUT#
CHR_C		Number of received characters specification	
CLR_BUF		Receiving buffer clear	
TMOUT		Non-reception time-out specification	
LONG_PRG	Touch panel	Conversion of program number into long	S_MBK
B7N,B7E,B7O B8E,B8O		Frame parity specification	MEWNET
Ub,Lb	Touch panel	Higher, lower order byte specification	IN() * MBK I/O rea
CompoWay COMPOWAY	Serial communication	OMRON CompoWay specification	PRINT#
RS485,RTS	Serial communication	Performs RTS control. RS-485 communication.	PRINT#

Miscellaneous			
_NEXT	Control statement	Option of RESUME	RESUME
OFF	SENSE_ON/_OFF	Option of SENSE_ON/OFF	SENSE_ON/_OFF
AVOID	IO	Command invalidation	ON,OFF,OUT, PULSE_OUT
ON_USB	USB	USB enable port	MPC-1000
USB,USB0,USB1 USB2,COM		USB channel specification	USB_WRITE, INPUT#
AD7890-10	AD	At the time of replacing AD7890-10	SET_AD
AD0,AD1		AD board selection	
SET_SF	NC command	Option of GET_CODE	GET_CODE
ALLOW	Unused	Deleted in the future	
Pulse Generation			
SACL	MPG-2314/-2541	Sigmoid specification	ACCEL
VOID		Invalid argument	MOVS,MOVL
X_A		X-axis specification	PG in general
Y_A		Y-axis specification	
Z_A		Z-axis specification	
U_A		U-axis specification	
ALL_A		All-axis specification	
VOID_X		Exclude X-axis = Y_A|U_A|Z_A	
VOID_Y		Exclude Y-axis = X_A|U_A|Z_A	
VOID_Z		Exclude Z-axis = X_A|Y_A|Z_A	
VOID_U		Exclude U-axis = X_A|Y_A|Z_A	
X_E	MPG-2314	X-axis error specification	RR(X_E)==0
Y_E		Y-axis error specification	
Z_E		Z-axis error specification	
U_E		U-axis error specification	
ALL_E		All-axis error specification	
POS_L	MPG-2314/-2541	Positive large number	HOME
NEG_L		Negative large number	
XIN0~XIN3	MPG2314	HPT input specification IN1, IN2 short IN3 non-connection	HPT()
XINP,XALM			
YIN0~YIN3			
YINP,YALM			
UIN0~UIN3			
UINP,UALM			
ZIN0~ZIN3			
ZINP,ZALM			
N_SDX	MPG-2541		HPT()
N_SDY			
N_SDU			
N_SDZ			
P_SDX			
P_SDY			
P_SDU			
P_SDZ			
STP_I	MPG-2314/-2541	Immediate stop	STOP
STP_D		Slow-down and stop	

INP_ON	MPG-2314	Valid with in-position ON	INSET
INP_OFF		Valid with in-position OFF	
INP_NO		In-position invalid	
ALM_ON		Valid with alarm ON	
ALM_OFF		Valid with alarm OFF	
ALM_NO		Alarm invalid	
NO_PHASE		Counter input	
PAHSE1		Single encoder input	
PHASE2		Double encoder input	
PHASE4		Quadruple encoder input	
UP_DWN		Up-down counter	
MD_2PLS		CW/CCW pulse output	
MD_DPLS		Direction instructing pulse output	
LMT_ON		X-LMT ~ Z-LMT ON valid	
LMT_OFF		X-LMT ~ Z-LMT OFF valid	
SLMT_ON		Soft limit valid	
SLMT_OFF		Soft limit invalid	
IN0_ON		Valid with XIN0 ~ ZIN0 ON	
IN0_OFF		Valid with XIN0 ~ ZIN0 OFF	
IN1_ON		Valid with XIN1 ~ ZIN1 ON	
IN1_OFF		Valid with XIN1 ~ ZIN1 OFF	
IN2_ON		Valid with XIN2 ~ ZIN2 ON	
IN2_OFF		Valid with XIN2 ~ ZIN2 OFF	
IN3_ON		Valid with XIN3 ~ ZIN3 ON	
IN3_OFF		Valid with XIN3 ~ ZIN3 OFF	
CW		Circular interpolation specification	SHOM/MOVT
CCW		Circular interpolation specification	
SLMTp		Soft limit + error	LMT() or PGE()
SLMTn		Soft limit – error	
LMTp		Limit + error	
LMTn		Limit – error	
EMG		EMG input error	
ALM		ALM input error	
IN0~IN3		Cause of stop	PGE()
CRL_ER		Error reset	
X_C		Counter specification	STPS
Y_C		Counter specification	
Z_C		Counter specification	
U_C		Counter specification	
VRING		Ring counter setup	Miscellaneous
PR_CHK		Operation precheck	
PGA,PGB	MPC-1000	PG active specification	MPC-1000 only

* Reserved constant can only be read out, and any attempt of setting a value will cause an error.

Data area

Separate from arbitrarily-usable variables, there are reserved arrays MBK( ), X( ), Y( ), U( ), and Z( ). MBK( ) is an array for a touch panel, if the touch panel is not used, it can be used as a general memory area.

X( ) ~ Z( ) are point data used for industrial robot-like purposes. When not used as point data, they can be used as array variables in the same manner as MBK( ).

In addition, an up a two-dimensional array variable can be defined and used by the DIM command.

Array Element	Type and Range	Purpose	Modification Method
MBK(n)	Word (2 bytes)0~8099	Shared memory with touch panel	S_MBK MBK(n)=m
X(n), Y(n), U(n), Z(n) Used as P(n) in PG command.	Long(4bytes) MPC-1000/2000 1~7000 MPC-2100 1~16000	Robot coordinate point data or data area	SETP X(n)=m
Array by DIM command	Up to 20000 in total	-	-

Character string variables

Character string variables are variables given $ at the end. Up to 128 character strings can be used, and the size of each character string is up to 255 bytes.

Character string operations are also supported, and combining can be performed using a '+' operator.

```
#a$="12345"+chr$(&h41)+"bcdef"
#pr a$
 12345Abcdef
```

As for search/editing of a character string, instead of a method centering the BASIC standard MID$, C language-like processing is made possible, using a task-local variable "ptr_" which is a pointer. (See SERCH, SERCH$, VAL, STRCPY, etc.) For extracting a numerical value in a character string, powerful VAL function is prepared. The numerical conversion process of the above character string a$ can be described as follows.

```
#pr val(a$)
 12345
#
```

A character string variable is handled as a point (actual address) in a normal arithmetic formula. Therefore, extracting a partial character string can be freely performed by the following operation for example.

```
10     a$="1234567890abcdefgABCDEFG"
30      SERCH   a$ "a"
35     s=ptr_-1 :  e=SERCH$("A") :  c=e-s-1
40     ptr_=s
50     c$=PTR$(c)
60      PRINT   c$
#run
 abcdefg
#
```

Arithmetic formula

In an arithmetic formula of BL/1, although multiplication and division are given priority over addition and subtraction, others are executed in order from the left. Other prioritized operations are enclosed with ( ).

```
a=1+2*3               a becomes 7 because execution is done in the order of 2 * 3 => 6, 1 + 6 => 7
a=(a1+a2+a3+a4)/4     Entered in a is the sum of a1 to a4 divided by 4 (average).
c=sqr(a*a+b*b)        This becomes square root of the sum of squares of a and b.
```

The length of formula (including conditional expression) is set within 102 characters, and although a considerably long formula can also be described, if there are too many ( )s, the internal memory is wasted, and a stack overflow may occur. Operations should be described simply and efficiently.

[Dyadic operator]

+	Addition	<<	Left shift (× 2n)
-	Subtraction	>>	Right shift (/2n)
*	Multiplication	,	Word synthesis
/	Division	;	Upper-order byte
%	Multiplication/division	&	Logical product
^	Exclusive logical sum	|	Logical sum

Conditional expressions

The difference between a conditional expression (logical formula) and an arithmetic formula is that although the result of an arithmetic formula becomes an integer, the result of a logical formula takes only the value of 1 (true) or 0 (false).

a==5 a and 5 are compared to obtain 1 or 0. The result is 1 (true) if they are equal, or 0 (false) if they are not equal.

A logical formula becomes an argument of IF statement and others as follows. If true, the part from immediately after THEN to ELSE or END_IF is executed.

```
IF a==5 THEN : ON 1 : ELSE : ON 2 : END_IF

a==5&(b==3)
```

In this example, a is compared with 5, and AND is operated on the result and the result of comparing b and 3 is taken. Therefore, the whole becomes true when two conditions that a is 5 and that b is 3 are satisfied. In the same manner, the following case contains OR.
The reason b==3 is enclosed with ( ) is to prioritize this comparison operator.

```
a==5|(b==3)
```

This case becomes true (1) if a is 5 or b is 3.
Whether this kind of logical formula actually becomes true or false can be checked by

```
print a==5|(b==3)
```

According to this logical rule of taking 1 or 0, a logical formula can be simplified. Simplification leads to the speed increase of processing.

For example, WAIT is a command to continue waiting until a logical formula which is its argument becomes true (1), and the condition-wait can be described as follows.

```
WAIT (SW(0)==1)& (SW(2)==1)& (SW(4)==1)& (SW(7)==1)& (SW(-1)==1)
↓
WAIT SW(0)& SW(2)&SW(4) & SW(7) & SW(-1)
```

This is a simplification which is possible because SW function outputs the value of 1 when it is true. If logical reversal is necessary, it can be described in the following manner.

```
WAIT (SW(0)==1)& (SW(2)==1)& (SW(4)==0)& (SW(7)==0)& (SW(-1)==1)
↓
WAIT SW(0)& SW(2)&@SW(4) & @SW(7) & SW(-1)
```

@SW() function is the logical reversal of SW().
In logical operations, not only simple logical comparisons but also logical formula having character strings and arrays intermingled can be described.
In such a case, the comparison formula should be enclosed with ( ) and combined with & and/or |.

```
IF (a$=="123")&(b==100)&(mbk(5)>1000)  THEN
```

In an IF statement, using AND or OR between arguments is also possible.

```
IF   a==1 AND b==2 THEN
```

In general, a complex logical operation which connects multiple elements becomes faster in processing when it is assembled into one formula, which makes it harder to read.
Grouping the formulae by their significance and connecting the logics with AND and OR in the end would make the program easy to understand.

[Logical operator]

==	Coincide with	<	Smaller than
!= {<>}	Not coincide with	>=	Equal to or greater than
>	Greater than	=<	Equal to or smaller than

Control statements

1) Repetition and condition-wait

WHILE conditional expression ~ WEND	Conditional repetition
DO ~ LOOP, FOR ~ NEXT	Repetition, sequential processing
BREAK	Escape from repetition
WAIT	Conditional expression

- Infinite loop

```
DO
 ON 0
 TIME 500
 OFF   0
 TIME 500
LOOP
```

- Repetition for 10 times

```
FOR CNT=1 TO 10
 ON 0
 TIME 500
 OFF   0
 TIME 500
NEXT CNT                                  /* Return to FOR by incrementing CNT by 1
```

- Escape from repetition by BREAK

```
CNT=0
DO
 CNT=CNT+1
 IF CNT>10 THEN : BREAK : END_IF          /* Escape from DO~LOOP
 ON 0
 TIME 500
 OFF   0
 TIME 500
LOOP
```

- Escape from repetition by GOTO

```
 CNT=0
 DO
  CNT=CNT+1
  IF CNT>10 THEN : GOTO *PASS : END_IF    /* Escape from DO~LOOP
  ON 0
  TIME 500
  OFF   0
  TIME 500
 LOOP
*PASS
```

* Escape from FOR ~ NEXT by BREAK or GOTO is also possible

- Condition-wait

```
        WAIT SW(192)==1   /* Wait for SW(192) to become ON
```

2) Conditional branch

IF ~ THEN ~ [ELSE] ~ END_IF	Branch
SELECT_CASE ~ END_SELECT	Branch by a numerical value

- IF statement, suitable for simple conditional judgment.

```
        IF SW(195)==1 THEN          /* If the front panel selector SW is ON,
         GOTO *MANU
        ELSE                        /*  Otherwise
         GOTO *AUTO
        END_IF
       *MANU
        OFF 0 : ON 1
        PRINT "MANUAL MODE"
        END
       *AUTO
        ON 0 : OFF 1
        PRINT "AUTO MODE"
        END
```

- SELECT_CASE statement. When there are multiple conditions.

```
        OFF 0 : OFF 1 : OFF 2       /*Turning off LED
        DSW=IN(24)/16               /* Reading out the front panel DSW
        SELECT_CASE DSW             /* Examining the DSW value
         CASE 0                              /* If DSW = 0
          ON 0 : OFF 1 : OFF 2
         CASE 1                              /*  If DSW = 1
          OFF 0 : ON 1 : OFF 2
         CASE 2                              /* If DSW = 2
          OFF 0 : OFF 1 : ON 2
         CASE_ELSE                  /* Otherwise
          ON 3 : TIME 10 : OFF 3
        END_SELECT
```

- When VOID is specified as the argument of SELECT_CASE, a CASE statement proprietary logical expression is evaluated and executed. When multiple conditions must be examined, it is more efficient than listing those IF statements.

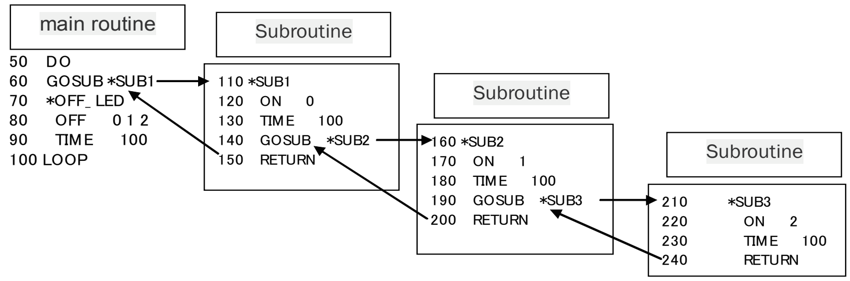
```
        SELECT_CASE VOID
         CASE SW(192)==1          /* Green SW ON -> Green LED ON
          ON 0
          OFF 1 2
         CASE SW(193)==1          /* Yellow SW ON -> Yellow LED ON
          ON 1
          OFF 0 2
         CASE SW(194)==1          /* Red SW ON -> Red LED ON
          ON 2
          OFF 0 1
         CASE SW(195)==1          /* Selector SW right side -> LED OFF
          OFF 0 1 2
         CASE_ELSE
        END_SELECT
```

3) Subroutines

GOSUB,RETURN	Jump to a subroutine, return from a subroutine
_VAR,_RET_VAL	An argument to a subroutine, a returned value from a subroutine

Making subroutines for individual work units and calling them from the main routine will

make the program easy to read.
If there is only GOSUB but no RETURN, a “Stack overflow” error will occur.



```
50   DO
60   GOSUB *SUB1
70   *OFF_ LED
80     OFF    0 1 2
90     TIME     100
100 LOOP
```

```
110 *SUB1
120   ON    0
130   TIME    100
140   GOSUB   *SUB2
150   RETURN
```

```
160 *SUB2
170   ON    1
180   TIME    100
190   GOSUB   *SUB3
200   RETURN
```

```
210       *SUB3
220         ON    2
230         TIME    100
240         RETURN
```

- An argument to a subroutine can be given to GOSUB. In the subroutine, the value is obtained by the _VAR command.
- A return value from a subroutine can be given after RETURN. Obtaining the return value is by _RET_VAL.
- In combination with a local variable, sharing a subroutine among tasks becomes possible.

```
10      OFF   0 1 2
20      TIME   500
30      GOSUB   *ON_LED 0 1 2 /* Calling a subroutine with an argument
40      END
50     *ON_LED
60      _VAR   A_ B_ C_      /*  Receiving an argument (one with _ is a local variable.)
70      ON   A_
80      TIME   500
90      ON   B_
100      TIME   500
110      ON   C_
120      RETURN
```

```
 10      GOSUB   *READ_DSW
20      _RET_VAL   D        /*  Receiving a return value
30      PRINT   D
40      END
50     *READ_DSW
60     DSW_=IN(24)/16       /*  Reading in the front panel DSW
70      RETURN   DSW_      /* DSW_ is a return value.
```

ON_ERROR

In BL/1, if an error occurs during a program execution, execution of the program stops. Because an error is usually fatal, the program should be modified based on the error so that no error will occur.

Although that would be sufficient while developing a program, once the actual operation is started, stopping the program is not preferable. ON_ERROR can capture an error during execution and avoid error processing program. In the ON_ERROR process, the error code and the statement number where the error occurred are stored in a task variable err_.
The error code can be obtained by err_>>24.

If an error further occurs in the error processing program, the error is only displayed but no jump to the error processing program occurs. The error jump prohibited state is released only when GOTO command or RESUME command has been executed.
For error codes, see the Error Code Table at the end of the volume.

1) Recoverable case

In external equipment such as USB memory, runtime errors can occur due to defects, degradation, and/or insufficient reliability. In such a case, a treatment such as RST_USB is performed to have the process retried. In this case, as a command to restore the program control to the original position, RESUME is available.

RESUME	Return to the command wherein an error occurred
RESUME _NEXT	Return to the line following the command where an error occurred

2) Unrecoverable case

Errors occurring due to mistaken I/O numbers or variable setup during operation require the program to be modified later. In such a case, a process to notify the touch panel or the like of the error character string and the location of occurrence is performed.

```
 ON_ERROR  *err
FILE$="TEST.TXT"
 DO
  USB_WRITE   "TEST\n"
  OUT   0 -10000
 LOOP
*err
 SELECT_CASE   err_>>24
  CASE   53
  CASE   54
  CASE   55
  CASE   56
  RST_USB   : TIME   500 : INC   usb_err : RESUME
  CASE_ELSE
   S_MBK   err_>>24 100 : S_MBK   err_&&H00FFFFFF 101 : S_MBK   ERR$(err_) 102 40
   ON   PATRIGHT
 END_SELECT
 END
#run
#pr mbk(100)
 9
#pr mbk(101)
 50
#pr mbk$(102,40)
      I/O range is exceeded.
##
```

Usage of SELECT_CASE VOID

SELECT_CASE has an expanded description method.
Usually, control is performed by categorizing a variable as shown below.

```
DO
SELECT_CASE   A
CASE   100 : FORK 1 *SHORI :  WAIT SW(192)==0
CASE   101 : FORK 1 *SHOR2 :  WAIT SW(193)==0
CASE   102 : FORK 1 *SHOR3 :  WAIT SW(194)==0
CASE_ELSE
END_SELECT
LOOP
```

However, if an argument is set as VOID as follows, CASE statement executes an independent logical evaluation. Because an exclusive processing can be performed out of parallel conditions, a clean description becomes possible by eliminating a complex IF_ELSE syntax.

```
DO
SELECT_CASE   VOID
CASE   SW(192)==0 : FORK 1 *SHORI :  WAIT SW(192)==0
CASE   SW(193)==1 : FORK 1 *SHOR2 :  WAIT SW(193)==0
CASE   SW(194)==0 : FORK 1 *SHOR3 :  WAIT SW(194)==0
CASE_ELSE
END_SELECT
LOOP
```