Casio FX-870P

Casio VX-4

BASIC, CASL, C, ASM, FX Statistik, F.Com, DataBank
FX-870P = 64Kb, VX-4 = 8Kb Standart RAM

8-Bit-CPU HD61700 Cross Assembler from Hitachi

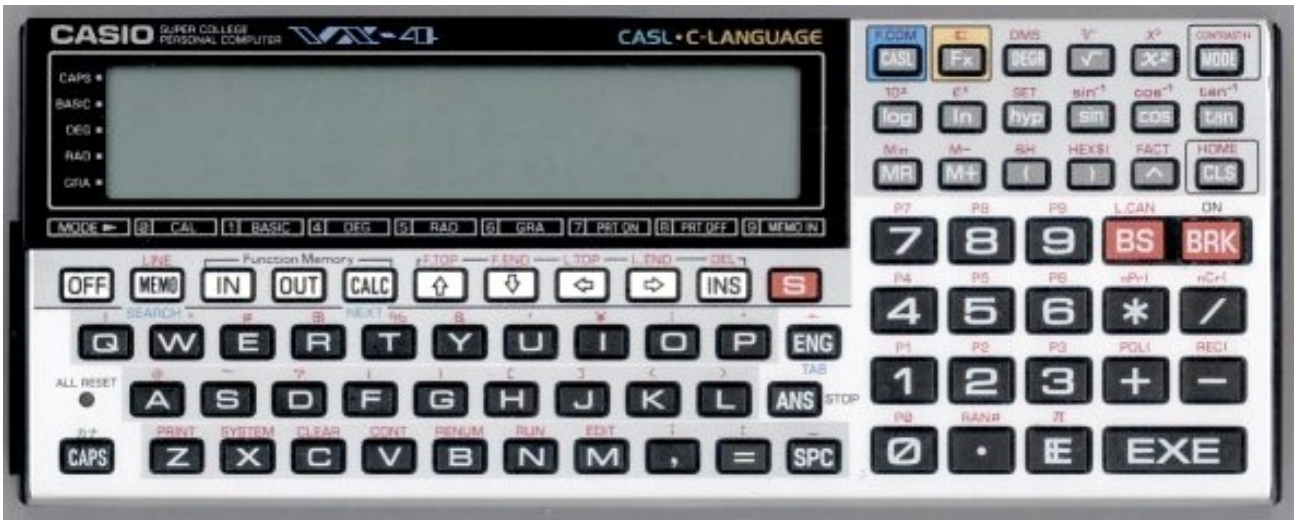


Table of Contents

Casio FX-870P Casio VX-4

Introduction about FX-870P / VX-4

FX-870P / VX-4 is a Model Developed from PB-100 Series.
8-bit CPU Hitachi's HD61700

Caution: This content is centered on the manual included with Casio Computer Co. Ltd. VX-4. Furthermore, there is no German or English manual for the Casio FX-870P and the VX-4.

!! This manual is based on the Japanese article and Pages from the original manual !!

http://luckleo.cocolog-nifty.com/pockecom/VX-4/HTML/fx-870p_manual_jp.html .
It ist only written in japanes languarge. Its was translated with Google-Translater in english and manual corrected (the german translation was to crazy. e.g. Basic words was translated incorrectly, Sentences have been translated incomprehensibly). Errors cannot be rulet out ! In some cases there is information from the original jap. operating instructions.

However, since the release date of information is often old, please avoid making inquiries to Casio Computer Co., Ltd.

① Because the internal calculation accuracy is higher than that of other companies, more accurate calculation is possible in complicated calculations and financial calculations.
② 10 program areas and 10 file areas,
③ Formula function,
④ Data Bank function,
⑤ Statistical processing function,
⑥ A relatively powerful BASIC that can use labels in other dimensions, but can use variable names of up to 255 characters,
⑦ C language interpreter,
⑧ CASL,
⑨ With an 8-bit CPU called Hitachi's HD61700 and an operating frequency of 910 KHz and many instructions of 10 to 20 clocks, the power consumption of the Pokécon is 0.08W with a processing performance of less than 0.1 MIPS. Time is secured (0.08W is estimated to be the maximum rating in the calculation)

This is a feature.
On the other hand, as a disadvantage,

① Execution of self-made machine language programs was not officially supported (executable with hidden instructions),
② The liquid crystal is 191 x 32 dots, and the dot interval is perfectly uniform and suitable for graphic display, but it does not support graphic-elated instructions. Graphics are only possible through machine language,
③ Inconvenient because labels cannot be used in BASIC,
④ Program execution speed in C language is 10x faster than BASIC, and C language can only be used for learning.
⑤ The VX-4 with only 8 Kb of memory consumes about 3.3KB in the system area, so an optional RP-33 or an additional memory upgrade is required to execute the appropriate program.

I. Basic Operation

1-1 Casio VX-4

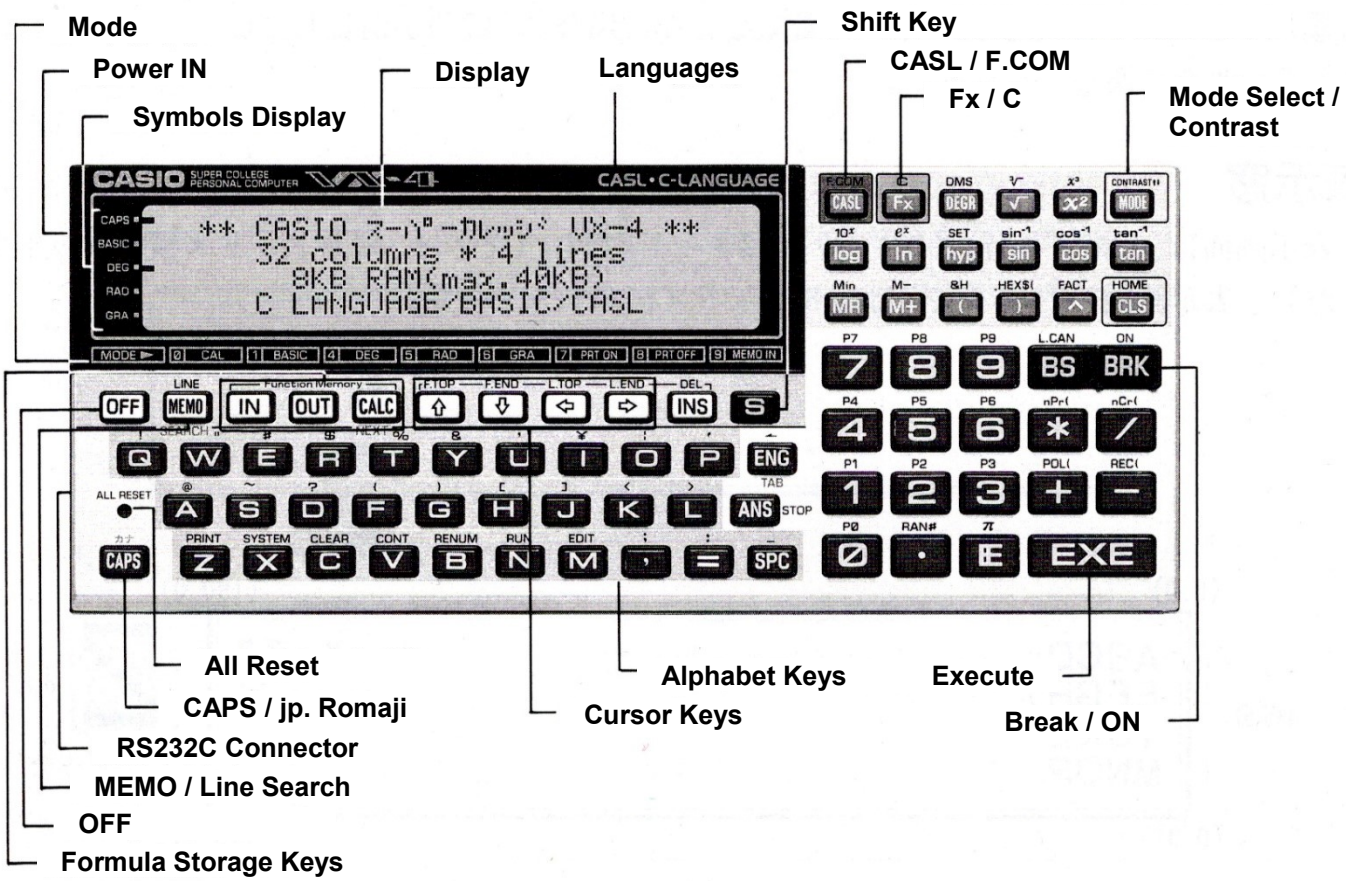


Table 1-0. FX-870P / VX-4 Modi with Mode Key

Mode Key &	Methode	Overview of Modes
0	CAL Mode	Selected when power ist switched ON
1	BASIC	10 programs writing / editing
4	DEG	angle unit = degree
5	RAD	angle unit = radians
6	GRA	angle unit = grads
7	Print ON	
8	PRINT OFF	
9	MEMO IN	Data Bank function

1-2 Battery Replacement

The battery used by FX-870P / VX-4 is

Battery for operation : 4x AA Batteries
Battery for data storage : 1x CR1220 Backup Batterie

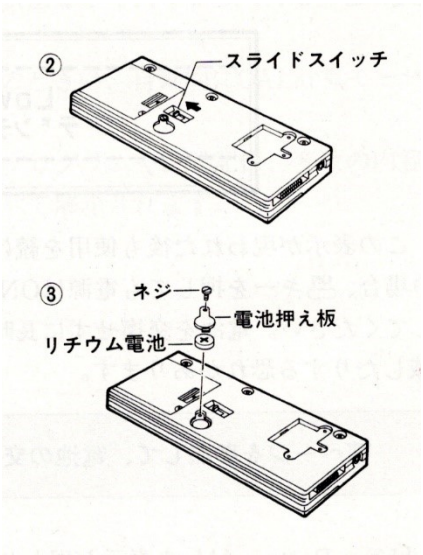
The battery does not start when the ON key ("BRK" key) is pressed , or the battery needs to be replaced when a low battery message is displayed after the ON key is pressed.
As a precaution when replacing batteries,

- **If the operating battery and data storage battery are removed at the same time, data such as programs will not be saved.**
- **When the operating battery and data storage battery are removed at the same time, it is necessary to press the P button on the back of the main unit and the ALL RESET button on the front of the main unit in turn with a thin stick like a toothpick.**

Is mentioned.

Replacing the operating battery (AA batteries; AA batteries)

1. Turn the three metal pig screws of the main unit with a coin to remove the metal pig.
2. When you remove the metal pig, there is a slide switch engraved on and off on the back of the main unit. Turn that switch off.
3. Slide the battery slide pig while pressing ▼ to remove the battery slide pig.
4. Take out the old battery and set four AA batteries (AA batteries) according to the instructions on the inner electrode.
5. Refit the battery slide pig.
6. Set the slide switch to ON.
7. Insert a metal pig and tighten the three screws.



Replacement of data storage battery (CR1220) Since the life of the data storage battery is 24 months, it must be replaced once every two years.

1. Turn the three metal pig screws of the main unit with a coin to remove the metal pig.
2. When you remove the metal pig, there is a slide switch engraved on and off on the back of the main unit. Turn that switch off.
3. Loosen the small screw that is tightened and remove the retainer plate, because the bottom of the circular retainer plate with a diameter of about 1 inch (2.54 cm) near the slide switch is where the CR1220 is set.
4. Set the battery with the + electrode of CR1220 facing up (the side closer to the pressing plate when the pressing plate is fitted).
5. Fit the holding plate and tighten the small screw.
6. Set the slide switch to ON.
7. Insert a metal pig and tighten the three screws.

As a precaution when replacing AA batteries (AA batteries) or CR1220, leave the slide switch OFF during the replacement.

Note that if the FX-870P and VX-4 fail to start normally before replacing the battery, for example because they have not been used for a long time, the P button on the back of the main unit and the front of the main unit Press the ALL RESET button sequentially with a stick with a thin tip like a toothpick. After pressing ALL RESET,

```
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
                 A l l   r e s e t / M e m o r y
                 R A M   :     8 K B   +     0 K B
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

When the "BRK" key is pressed and the above message disappears, all memories are initialized and all stored data can be used after being erased. In the above message, the first number following "RAM:" is the RAM capacity of the main unit, and the latter number is the capacity of the additional RAM such as RP-33. Check the memory capacity of FX-870P / VX-4. it can.
The ・ P button on the back of the main unit

- I was shocked by strong static electricity,
- Executed machine language and run out of pocket

Used when it does not operate normally due to the above.
• The P button and ALL RESET button are not obtained and analyzed, so they are speculated from other sources, but they seem to be CPU reset and key interrupt, respectively. Therefore,

- The P button resets the CPU, performs the CPU initialization routine, and performs the minimum CPU settings (for example, assigning constants to registers $ 30 and $ 31), but does not initialize the RAM.
- Pressing the ALL RESET button is detected by the key matrix standard input routine, and the RAM initialization routine is executed.

I guess it is a two-stage configuration.
Low battery display When the battery is depleted and the battery needs to be replaced, a low battery message is displayed as shown below. In that case, AA batteries (AA batteries) must be replaced as described above.

```
- - - - - - - -  -   - - - - - -   - - - - - -  - -   -   -   - - - - - - - -
                L   o w   B a t   t e r y   ! !   !
             Te The N H   ヲ   The C K N S Te K Data The The Y
- - - - - - - -  -   - - - - - -   - - - - - -  - -   -   -   - - - - - - - -
```

It can be used even if this display appears, but the power is forcibly turned off after about 1 hour. In that case, the FX-870P / VX-4 does not turn on when you press the ON key, so you should replace the battery as soon as the low battery indicator appears. Leaving the battery without replacing it may cause battery leakage or data corruption.

If an error occurs during programming and the battery is depleted, "Low Battery !!!" appears and then an error message is displayed.

(note)
The button battery model number is determined by the international standard IEC60086 so that the battery specifications can be understood. In the case of CR1220, C means that the battery system is a manganese dioxide / lithium battery (nominal voltage: 3.0V), and R means round. 1220 represents a diameter of 12 mm and a thickness of 2.0 mm.

1-3 Power ON / OFF and Contrast Adjustment

Power on

The right "BRK" key on the right also serves as the ON key, so press this button. If it starts normally, the CAL mode is entered, the cursor blinks and the input is waited.
If there is no response when pressed, the possibilities other than failure and the corrective actions are as follows.

- It is operating normally, but the LCD contrast is 0 and the display is not visible. → Adjust the LCD contrast.
- Continued use with Low Battery, or the system is in some sort of runaway state. → Press the P button on the back of the main unit with a stick with a thin tip such as a toothpick. If this happens, the program or file may be safe. If you are worried, press the ALL RESET button again to initialize the RAM.
- The AA battery for operation has run out. → Replace the AA batteries and, in some cases, replace the CR1220 for data storage.

Power off

Press the OFF button in the upper left to turn off the power.
In addition, if the computer is left waiting for input, that is, when FX-870P / VX-4 is not performing calculations, the power is automatically turned off in a fixed time (several minutes).
This is called an **auto power off function** . **When the power is turned off with auto power off, all mode settings such as the number of digits are cancelled, but files such as programs, mathematical formula storage, and materialized variable values remain saved.**

Contrast adjustment

- Press the "CONTRAST" key, that is, the "SHIFT" key and then the "MODE" key.
- Contrast up with the "↑" key just below the LCD and contrast down with the "↓" key. When you want to finish, press any other key.

If this still does not display correctly, it is likely that the battery has run out.

1-4 VX-4 - FX-870P - Modi

FX-870P / VX-4 has 6 modes besides CAL mode like scientific calculator.
Table 1-1 shows how to enter each mode and a brief description of each mode.

Table 1-1. FX-870P / VX-4 Modes and Transition Methods

Mode	Migration method	Overview of Modes
CAL	Default mode when power is turned on. Press "SHIFT" (red "S" key), then press "0".	Scientific calculator-like function and formula memory calculation function. see FX-880P manual ...
Databank / Memo-in Mode	Press "SHIFT" (red "S" key), then press "9".	Data (memo) input and search. see FX-880P manual ...
FX (Statistical Calculation)	Press the "FX" key at the top.	Statistical calculation and training board (not covered).
F.COM	Press the "F.COM" key, that is, the "SHIFT" key and then the "CASL" key.	Input / output of files including BASIC programs to external devices. File operations such as editing / deleting.
BASIC	Press the "MODE" key in the upper right and then press "1". When the numeric key (0-9) is pressed after pressing "SHIFT" (red "S" key) in CAL mode, if a BASIC program exists in the program number of the pressed number, execute it. Start.	BASIC mode. No grafical Funktions inside
C Language	Press the “¢” key, that is, the “SHIFT” key and then the “FX” key.	C language mode. 10x faster than BASIC see Z-1GR and PB-2000 C-manual ...
CASL	Press the “CASL” key.	CASL mode. Only japanese manuals. see also Sharp PC-G850V manual ...
Formular Storage function	Keys „IN“, „OUT“, „CALC“	Store often used formulas in memory for calculation. This funktion is applied in CAL-Mode. see FX-880P manual ...

CALモード

電源をONにすると、常にCALモードになります。
他のモードからCALモードに入るには、MODEキーに続けて⊘キーを押します。

CAPS
BASIC
DEG
RAD
GRA

データバンク/メモインモード

```
CAPS
BASIC
DEG
RAD
GRA
-
                                    〈      1〉
```

Fx(統計計算)モード

```
CAPS
BASIC
DEG
RAD
GRA
                    〈 Fx menu 〉
1:STAT(x)
2:STAT(x,y)
3:Training Board
```

F. COMモード

```
CAPS
BASIC
DEG
RAD
GRA
P 0 1 2 3 4 5 6 7 8 9    [RS232C]
F 0 1 2 3 4 5 6 7 8 9      3355B
P0>Save / Load / Merge / Copy
   Edit / New  / Print / Device
```

BASICモード

```
CAPS
BASIC
DEG
RAD
GRA
P 0 1 2 3 4 5 6 7 8 9      3355B
Ready P0
-
```

C言語モード

```
CAPS
BASIC
DEG
RAD
GRA
                    〈 C 〉
F 0 1 2 3 4 5 6 7 8 9      3355B
F2>Run/Load/Source/Cal
```

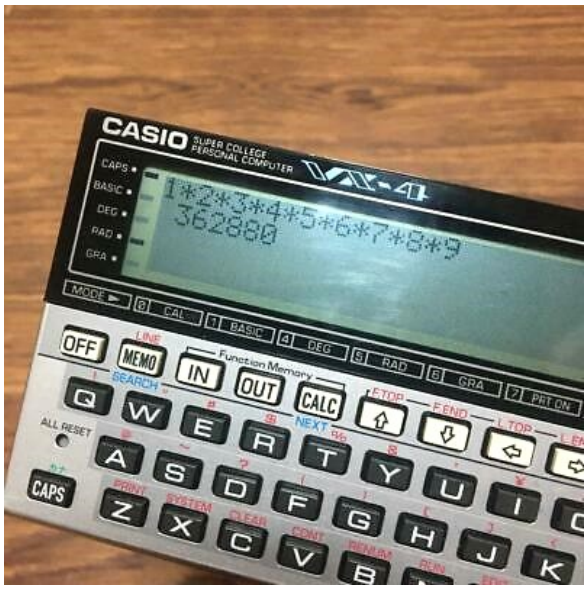
CASLモード

```
CAPS
BASIC
DEG
RAD
GRA
                    〈 CASL 〉
F 0 1 2 3 4 5 6 7 8 9      3355B
F1>Assemble/Source/Cal
```

1-5 Calculation in CAL- or RUN-Mode

In FX-870P / VX-4, for example, you can calculate in CAL mode or use PRINT statement in BASIC.

PRINT A
Even if is executed, the mantissa part displays only a maximum of 10 digits, and values in the range of 0 and $\pm 1 \times 10^{-99}$ to $\pm 9.999{,}999{,}999 \times 10^{99}$ seem to be the limit, but **FX-870P / VX The numerical value of -4 is expressed and calculated internally by BCD with 13 digits for mantissa and 2 digits for exponent (0 and $\pm 1 \times 10^{-99}$ to $\pm 9.999{,}999{,}999{,}999 \times 10^{99}$). Saved.**



For example,
A = 1.123456789012 and enter
PRINT A
Even if you do
1.123456789

(The SET statement only drops the number of display digits). However, using the PRINT USING statement,
PRINT USING "#. ############"; A

If you execute
1.123456789012
Is output, confirming that the internal precision is up to 13 digits.

Also,
PRINT USING "#. ################"; 1/9
Run
0.111111111111100
It is confirmed that the internal accuracy is 13 digits.

FX-870P / VX-4 performs rounding after the four arithmetic operations by default (at initialization) and MODE10 . The rounding method is

- When the 11 to 13 digit number is 049 or less,
- Round up when 11 to 13 digits are 950 or more

It is. Also, rounding after the four arithmetic operations can be disabled by MODE11 .
To check the rounding process, first enable the rounding process with MODE10,

A = 1.123456789049
PRINT USING "#. ############"; A
And run continuously. At this time, A = 1.123456789049 is displayed, and it can be confirmed that the **constant substitution is not rounded even if the rounding after the four arithmetic operations is valid** .
next,
A = A * 1
PRINT USING "#. ############"; A
And running continuously,
A = 1.123456789000

Is displayed and it can be confirmed that rounding has been performed by four arithmetic operations.
Here, the following table shows a summary of operation examples in the vicinity of the rounding threshold.

Table. Example of Calculation results when Rounding is enabled after four Arithmetic Operations (FX-870P / VX-4)

Assigned value to A	A value by A = A * 1 after MODE10	Rounding
1.123456788 049	1.123456788 000	Round down
1.123456788 050	1.123456788 050	No rounding
1.123456788 949	1.123456788 949	
1.123456788 950	1.12345678 9000	Round up

Here, in order to clarify the effect of rounding up and avoid confusion, the value of the 9th decimal place of the numerical value substituted before the calculation of the table is 8.
Also, whether the FX-870P / VX-4 is enabled or disabled can be checked by the value of RNDFL (old name: MODED, address: & H1133) in the system area. If the value of this address is 0, the rounding process at the time of arithmetic operation is valid, and if it is 1, the rounding process at the time of arithmetic operation is invalid. In particular,

DEFSEG = 0
PEEK (& H1133)

You can check the contents. However, the command DEFSEG = 0 is not necessary unless a DEFSEG instruction has been issued.
For details on the format of numeric variables, refer to A-2. BCD floating-point format and internal format in 12. FX-870P / VX-4 Internal Information.
Finally, the **successor FX-890P / Z-1 has a different rounding method,**

- If the 11-13 digit number is less than 007,
- Round up when the 11 to 13 digit number is 990 or more

And the rounding conditions are getting stricter. The rounding conditions for models prior to FX-870P / VX-4 are unknown because the authors do not have them.

(note)
In the case of Sharp's pocket computers, PC-E500 series models such as PC-E650 support double precision and can store 20 digits with 24 digits of computation (basic is single-precision and almost the same as conventional models), but most The model is 12 digits for computation and 10 digits for storage. The 11th and 12th digits are rounded, and the last byte of the 8 bytes stored in the memory as a variable value is 0, and the information is damaged (PC-1350 and PC-G850V have been confirmed to work). For this reason, Sharp's pocket computers are designed to easily accumulate errors when performing complex calculations. This is in contrast to Casio's Pokémon, which basically stores 13 digits of precision as described above.

Therefore, **Casio's pocket computer seems to be superior to Sharp in terms of calculation accuracy.**

use ENG

例題　1234567890と0.123456789を指数で求めなさい。

操作		表示	
①	1 2 3 4 5 6 7 8 9 0	①	1234567890_
②	EXE	②	1234567890
③	ENG	③	1.23456789E+09
④	0 · 1 2 3 4 5 6 7 8 9	④	0.123456789_
⑤	EXE	⑤	0.123456789
⑥	ENG	⑥	123.456789E-03

例題　0.12345×0.00001の結果を指数で求めなさい。

操作		表示	
①	0 · 1 2 3 4 5 * 0 · 0 0 0 0 1	①	0.12345*0.00001_
②	EXE	②	0.0000012345
③	ENG	③	1.2345E-06

また、仮数部の小数点の位置を変えるには次のように操作します。

操作		表示	
④	ENG	④	1234.5E-09
⑤	ENG	⑤	1234500E-12
⑥	ENG	⑥	1234500000E-15
⑦	SHIFT ←	⑦	1234500E-12
⑧	SHIFT ←	⑧	1234.5E-09
⑨	SHIFT ←	⑨	1.2345E-06
⑩	SHIFT ←	⑩	0.0012345E-03
⑪	SHIFT ←	⑪	0.000001234E+00
⑫	SHIFT ←	⑫	0.000000001E+03

Angle Modes

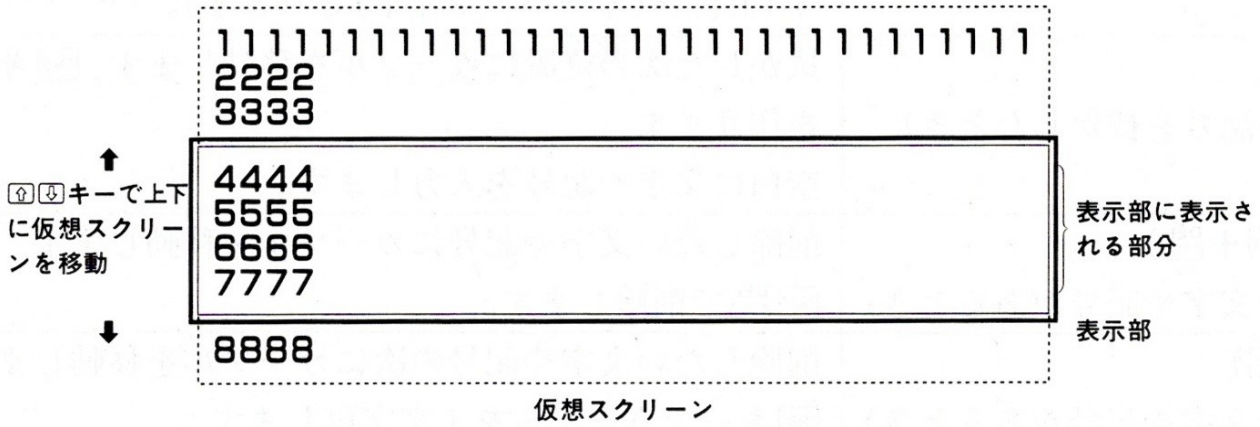
名　称	シンボル	数学記号	操　作	
度数法(デグリー単位)	DEG	°	MODE 4	A N G L E 0 EXE
孤度法(ラジアン単位)	RAD	rad	MODE 5	A N G L E 1 EXE
グラッド単位	GRA	grad	MODE 6	A N G L E 2 EXE

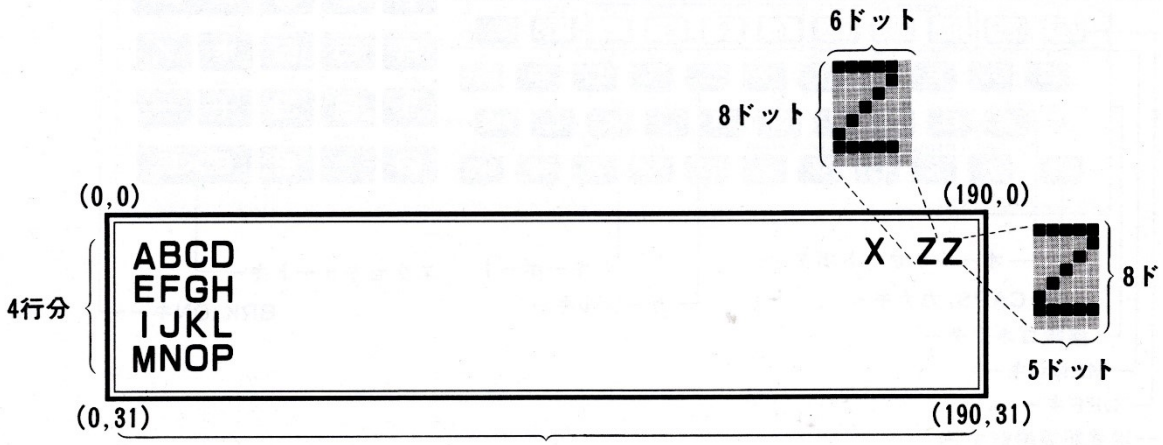
コラム●各度法の比較対照表●

名　称	例
度数法(デグリー単位)	0° 30° 45° 60° 90° 120° 135° 150° 180° 270° 360°
孤度法(ラジアン単位)	0 $\pi/6$ $\pi/4$ $\pi/3$ $\pi/2$ $2\pi/3$ $3\pi/4$ $5\pi/6$ π $3\pi/2$ 2π
グラッド単位	0 100/3 50 200/3 100 400/3 150 500/3 200 300 400

1-6 Display

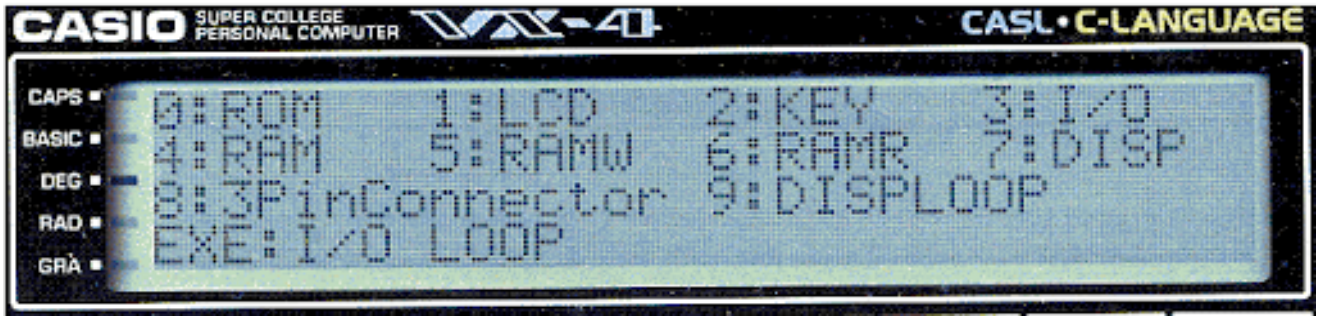
Display 4 Lines and virtuell Display 8 Lines



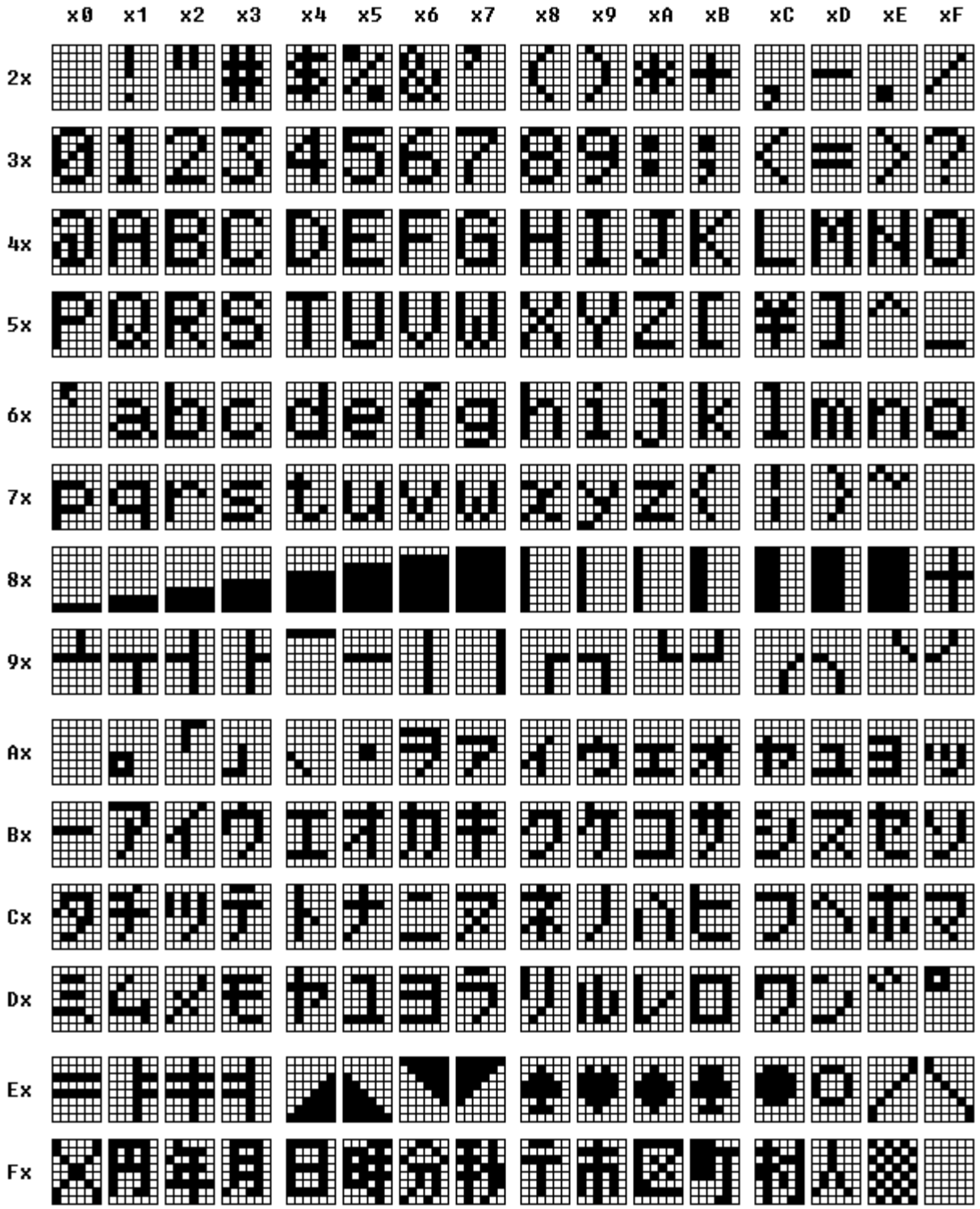


Selftest:

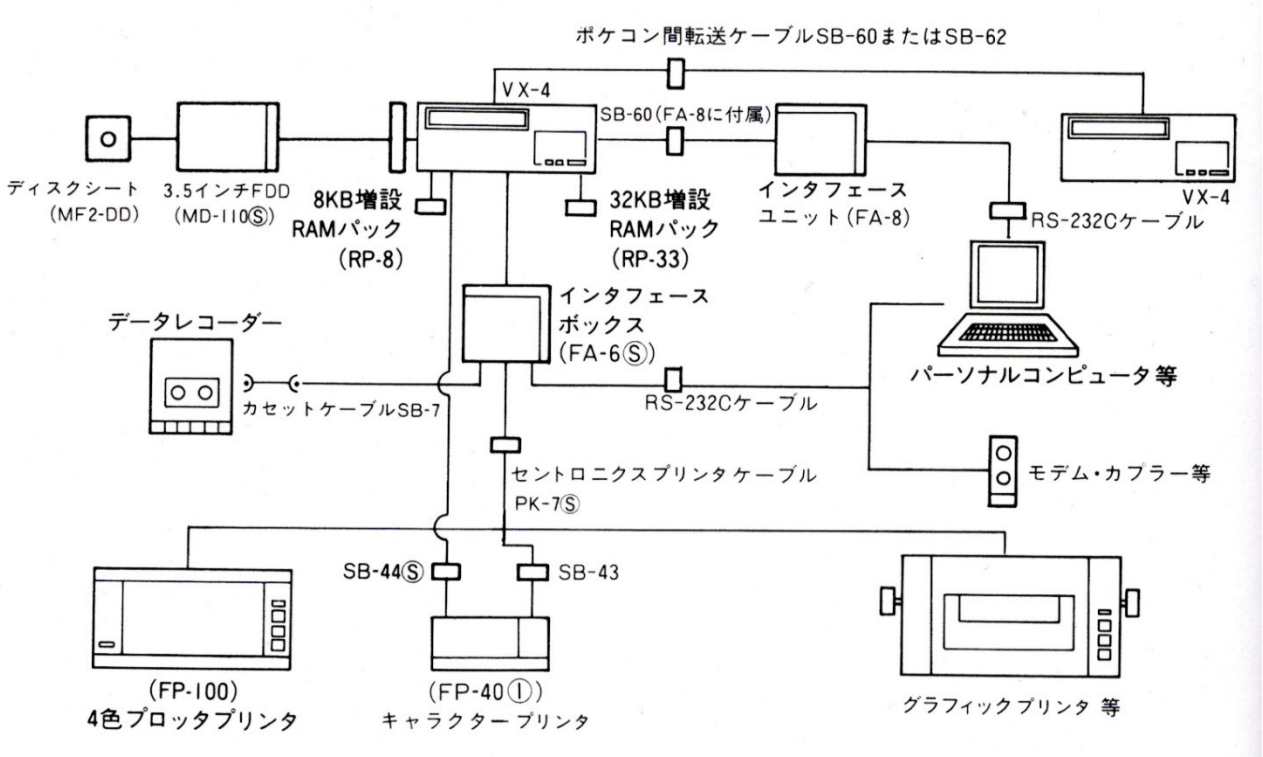
(BASIC) SYSTEM* / ENTER



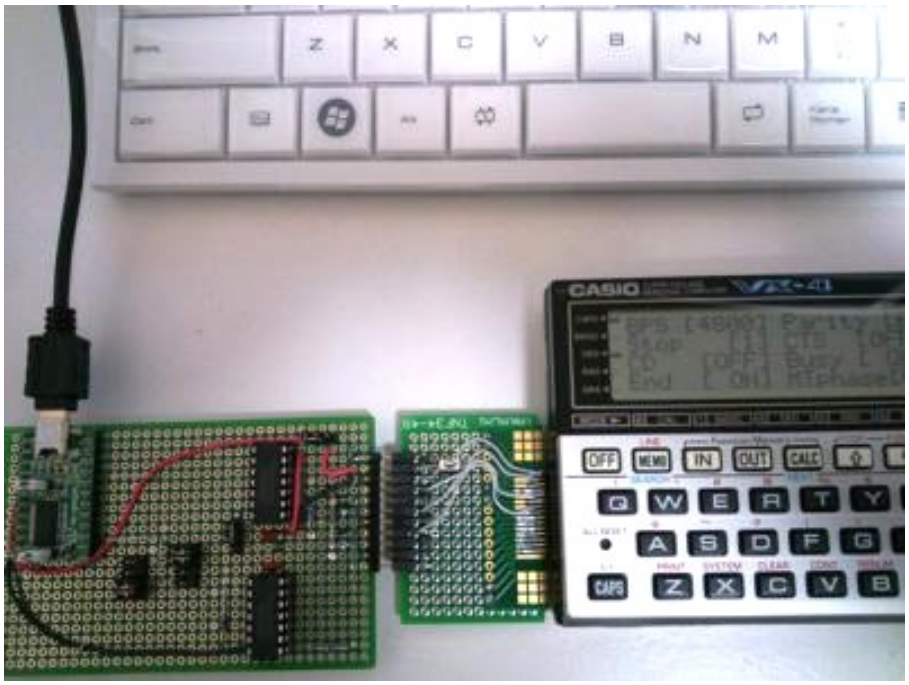
<u>only an Example</u> for a Char-Set with 5x7 Pixel



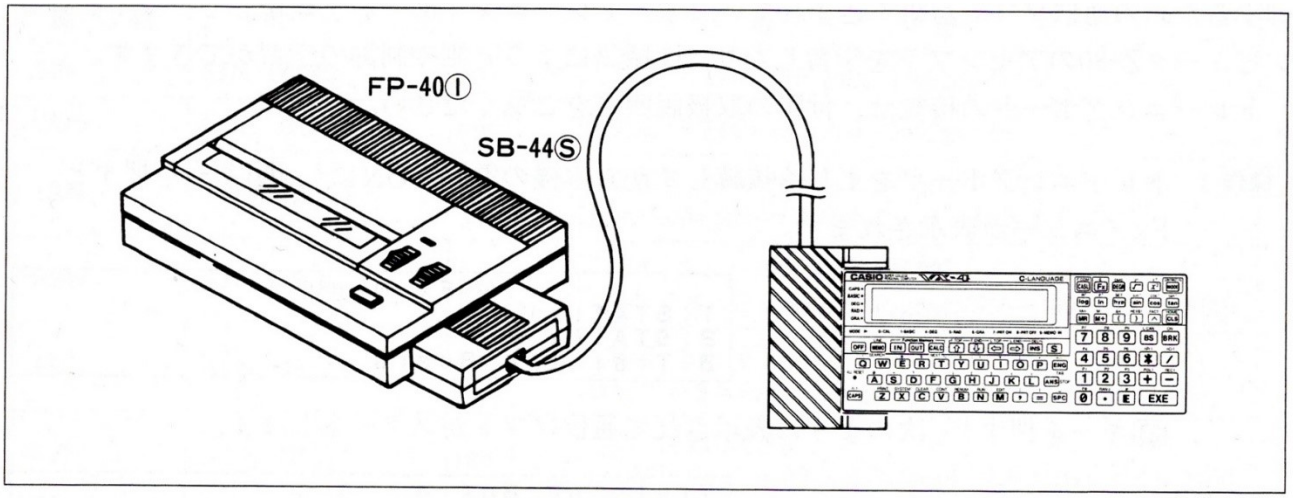
1-7 Accessories for the FX-870P / VX-4

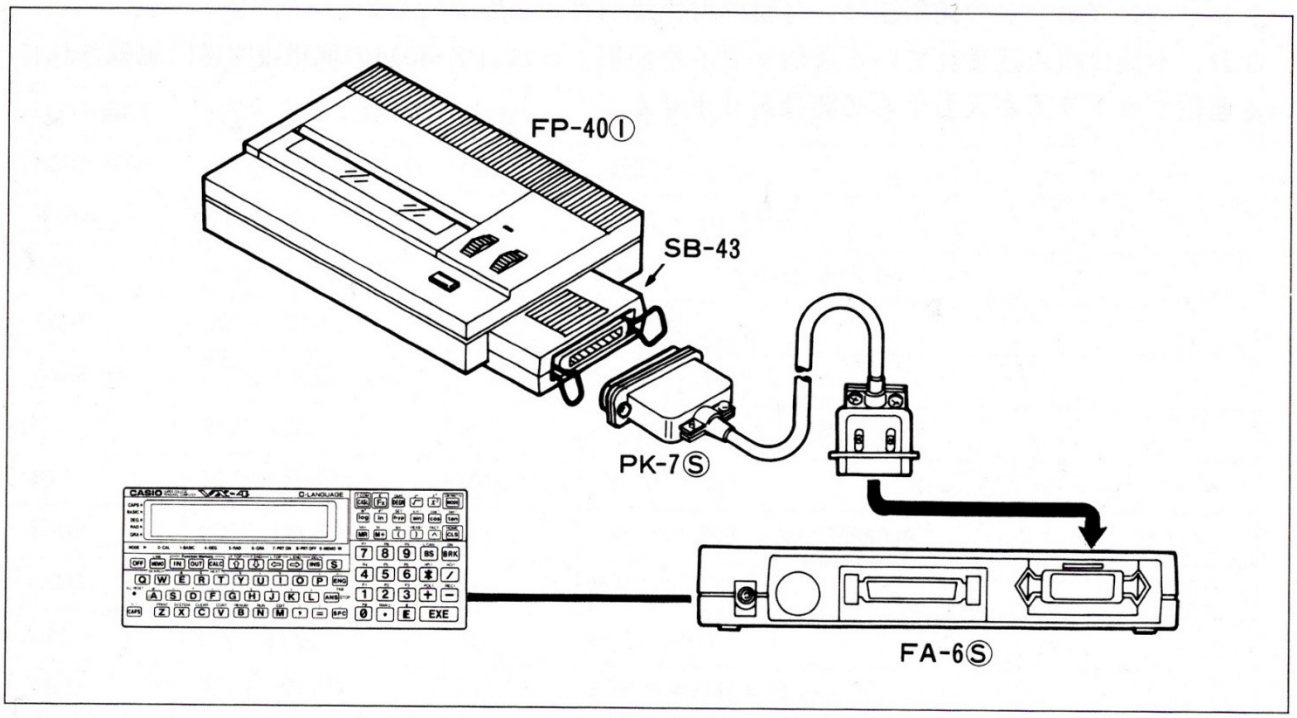


Kyoros Room Blog:

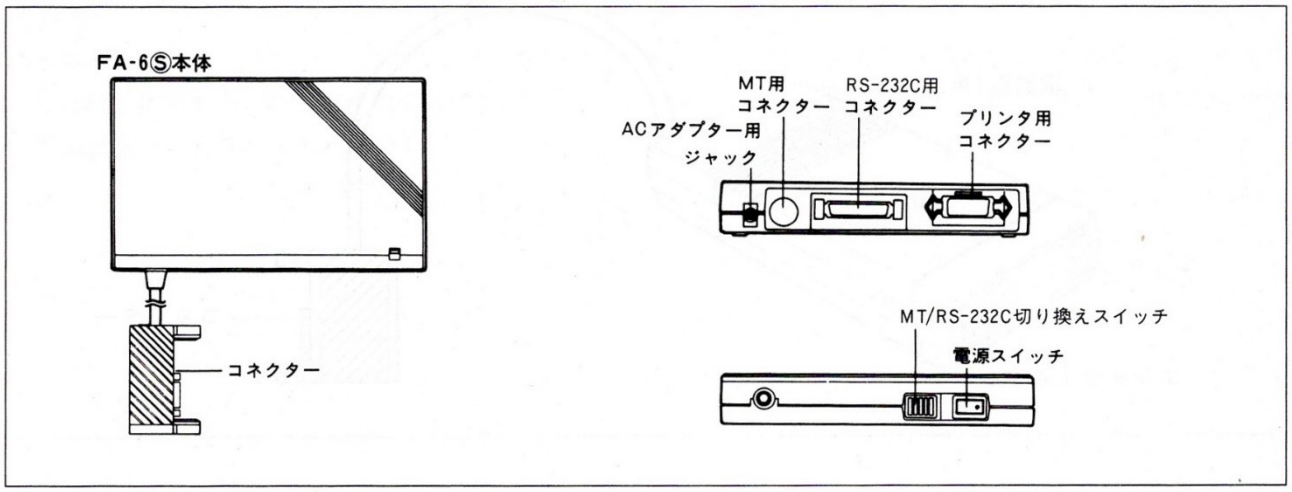


FP-40:

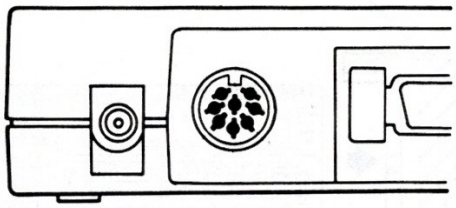

FP-40Ⓘ
SB-44Ⓢ


FP-40Ⓘ
SB-43
PK-7Ⓢ
FA-6Ⓢ

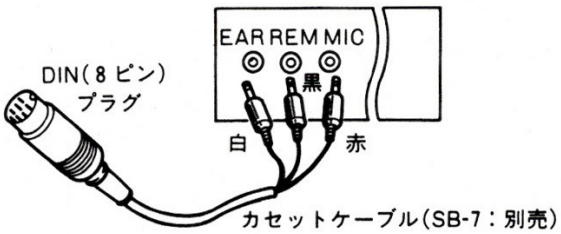
FA-6:


FA-6Ⓢ本体
コネクター
MT用
コネクター
RS-232C用
コネクター
ACアダプター用
ジャック
プリンタ用
コネクター
MT/RS-232C切り換えスイッチ
電源スイッチ

FA-6Ⓢ



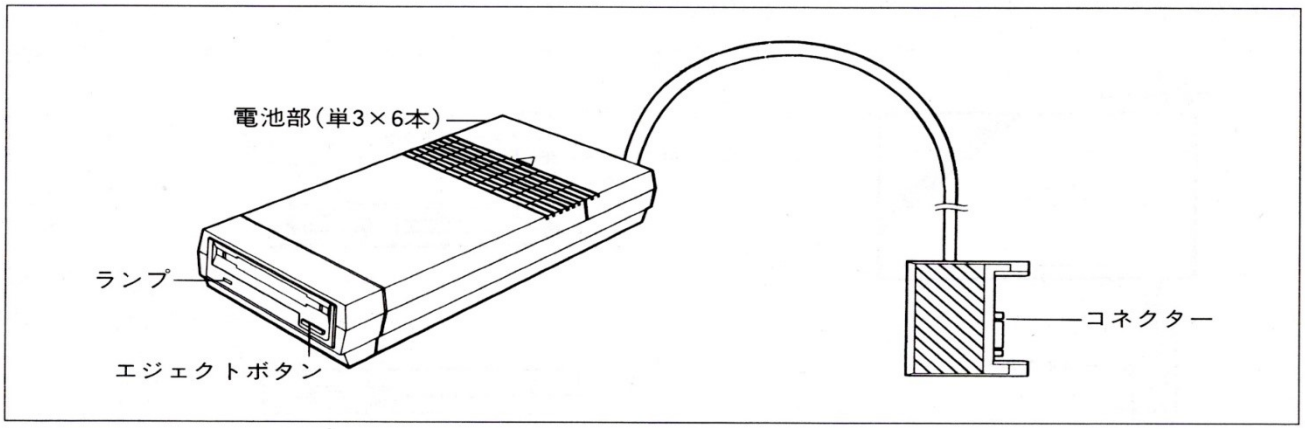
カセットテープレコーダー端子

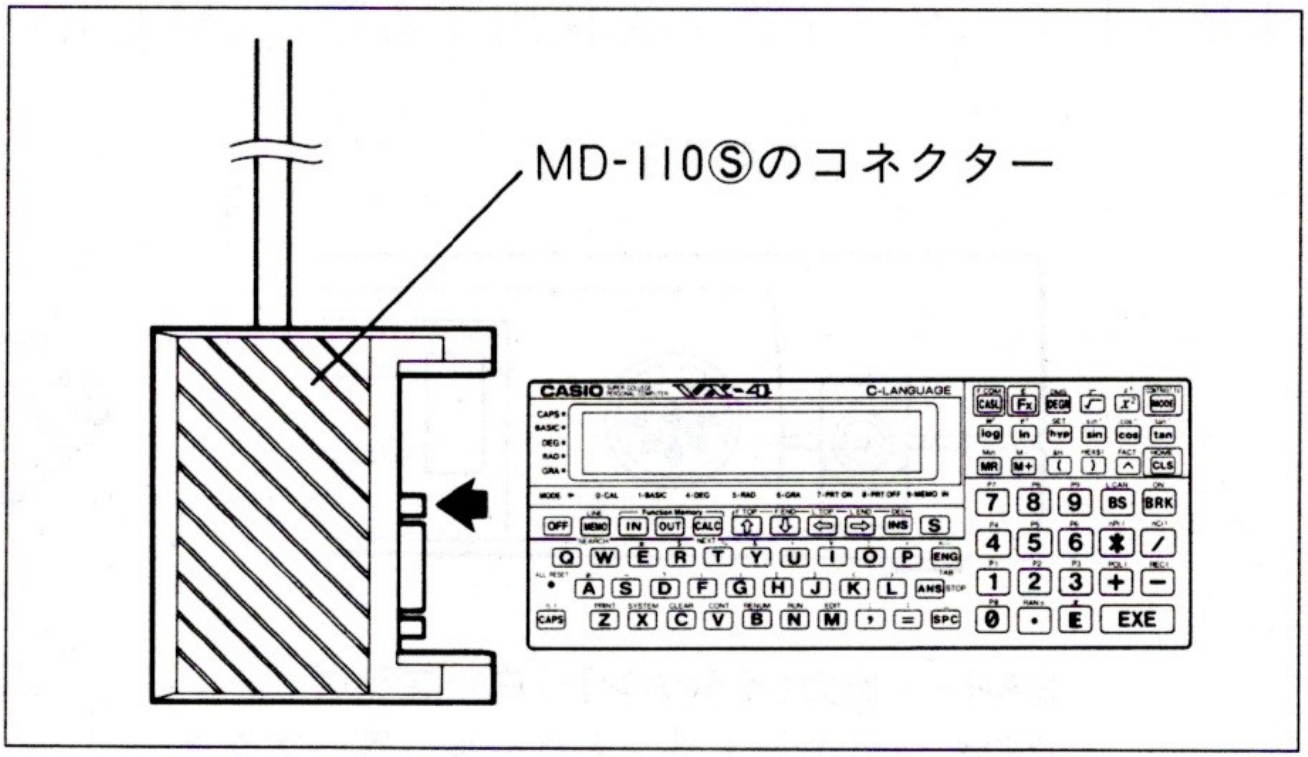


EAR——出力(イヤホン)…白いプラグ
REM——リモートコントロール…黒いプラグ
MIC——入力(マイク)…赤いプラグ

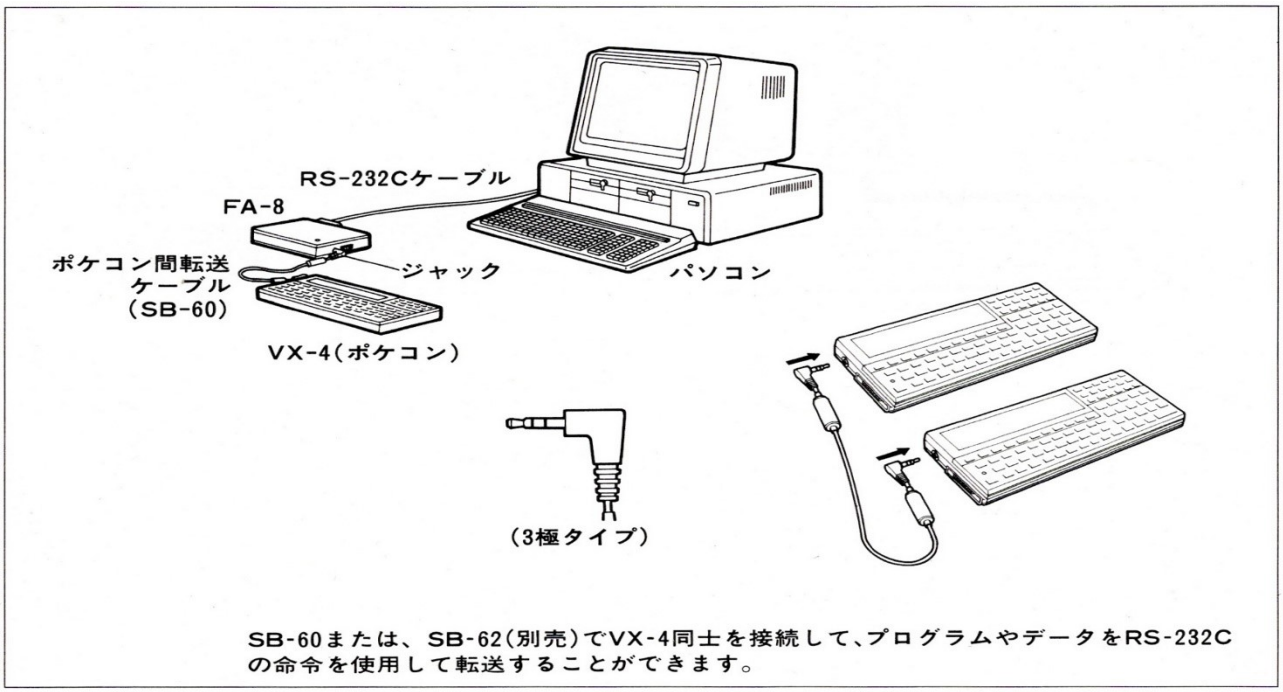
注) リモート端子のないカセットテープレコーダーの場合は、リモートプラグはどこにもつなぎません。

MD-110

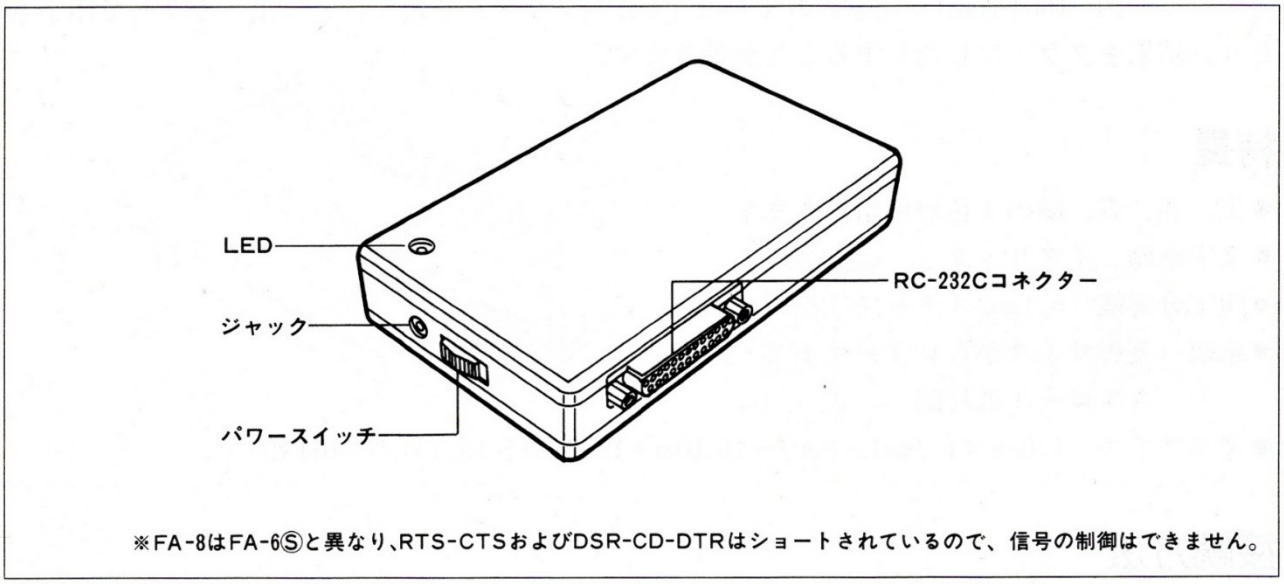




FA-8:

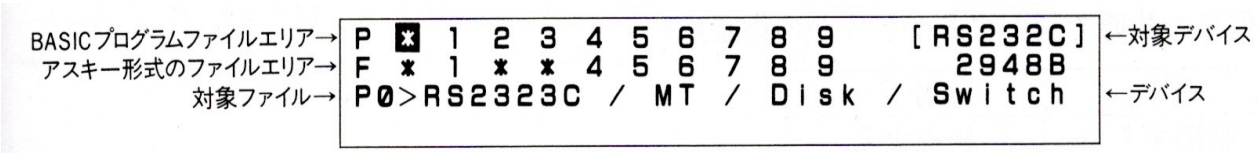


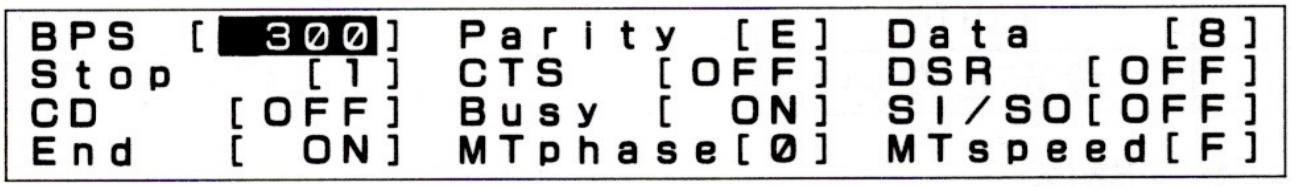
SB-60または、SB-62(別売)でVX-4同士を接続して、プログラムやデータをRS-232Cの命令を使用して転送することができます。



※FA-8はFA-6Ⓢと異なり、RTS-CTSおよびDSR-CD-DTRはショートされているので、信号の制御はできません。

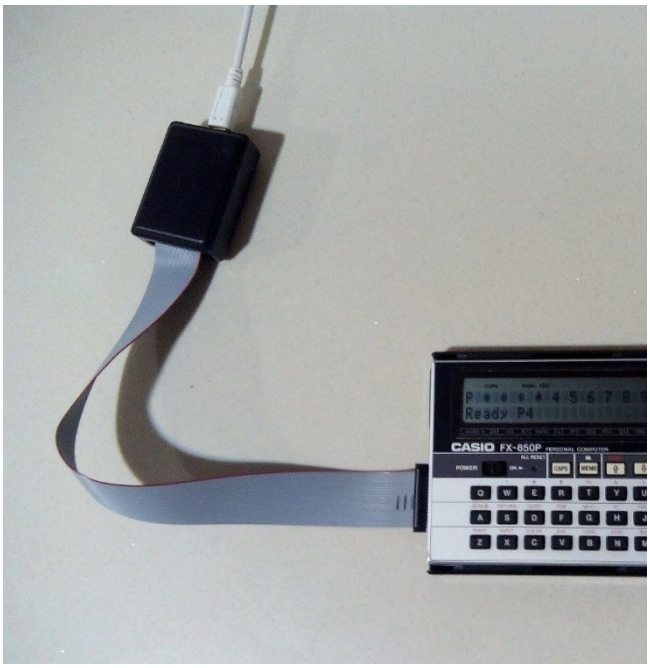
RS232C:

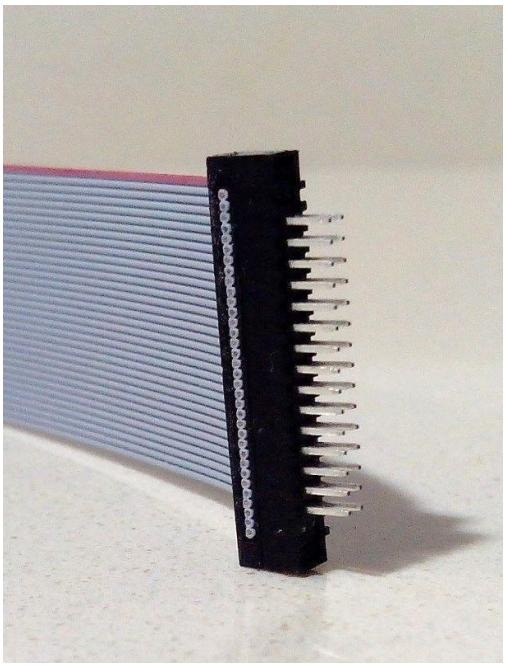




RP-8 = 8Kb, RP-33 = 32Kb RAM Speicher:

USB-Interface-Kabel for FX-850P to VX-4 (Inet 2020)





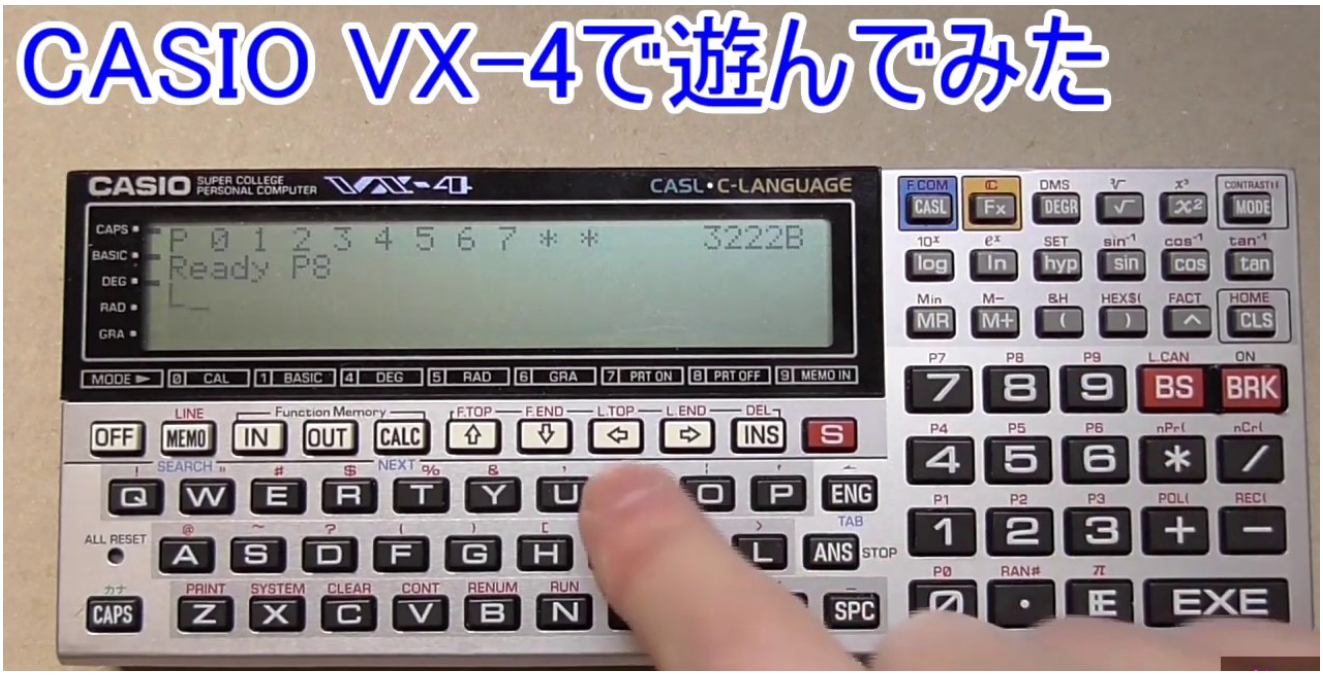
1-8 Romaji – Tabellen (Shift CAPS & …)

	ア列	イ列	ウ列	エ列	オ列
ア行	ア A	イ I	ウ U	エ E	オ O
カ行	カ KA CA	キ KI	ク KU CU	ケ KE	コ KO CO
ガ行	ガ GA	ギ GI	グ GU	ゲ GE	ゴ GO
サ行	サ SA	シ SI SHI CI	ス SU	セ SE CE	ソ SO
ザ行	ザ ZA	ジ ZI JI	ズ ZU	ゼ ZE	ゾ ZO
タ行	タ TA	チ TI CHI	ツ TU TSU	テ TE	ト TO
ダ行	ダ DA	ヂ DI	ヅ DU	デ DE	ド DO
ナ行	ナ NA	ニ NI	ヌ NU	ネ NE	ノ NO
ハ行	ハ HA	ヒ HI	フ HU FU	ヘ HE	ホ HO
バ行	バ BA	ビ BI	ブ BU	ベ BE	ボ BO
パ行	パ PA	ピ PI	プ PU	ペ PE	ポ PO
マ行	マ MA	ミ MI	ム MU	メ ME	モ MO
ヤ行	ヤ YA	イ YI	ユ YU	イェ YE	ヨ YO
ラ行	ラ RA LA	リ RI LI	ル RU LU	レ RE LE	ロ RO LO
ワ行	ワ WA	ウィ WI	ウ WU	ウェ WE	ヲ WO
ン行	ン N, X				
キャ行	キャ KYA	キィ KYI	キュ KYU	キェ KYE	キョ KYO

	ア列	イ列	ウ列	エ列	オ列
ギャ行	ギャ GYA	ギィ GYI	ギュ GYU	ギェ GYE	ギョ GYO
クァ行	クァ QA	クィ QI	クゥ QU	クェ QE	クォ QO
シャ行	シャ SYA SHA	シィ SYI	シュ SYU SHU	シェ SYE SHE	ショ SYO SHO
ジャ行	ジャ ZYA JA JYA	ジィ ZYI JYI	ジュ ZYU JU JYU	ジェ ZYE JE JYE	ジョ ZYO JO JYO
チャ行	チャ TYA CYA CHA	チィ TYI CYI	チュ TYU CYU CHU	チェ TYE CYE CHE	チョ TYO CYO CHO
ヂャ行	ヂャ DYA	ヂィ DYI	ヂュ DYU	ヂェ DYE	ヂョ DYO
テャ行	テャ THA	ティ THI	テュ THU	テェ THE	テョ THO
デャ行	デャ DHA	ディ DHI	デュ DHU	デェ DHE	デョ DHO
ニャ行	ニャ NYA	ニィ NYI	ニュ NYU	ニェ NYE	ニョ NYO
ピャ行	ピャ PYA	ピィ PYI	ピュ PYU	ピェ PYE	ピョ PYO
ヒャ行	ヒャ HYA	ヒィ HYI	ヒュ HYU	ヒェ HYE	ヒョ HYO
ビャ行	ビャ BYA	ビィ BYI	ビュ BYU	ビェ BYE	ビョ BYO
ファ行	ファ FA	フィ FI		フェ FE	フォ FO
フャ行	フャ FYA	フィ FYI	フュ FYU	フェ FYE	フョ FYO
ミャ行	ミャ MYA	ミィ MYI	ミュ MYU	ミェ MYE	ミョ MYO
リャ行	リャ RYA LYA	リィ RYI LYI	リュ RYU LYU	リェ RYE LYE	リョ RYO LYO
ヴ行	ヴァ VA	ヴィ VI	ヴ VU	ヴェ VE	ヴォ VO

II. BASIC - Referenz

Table of Contents



List of Manual Commands

LIST, LLIST	RENUM	NEW	PASS	RUN
SAVE	LOAD	MERGE	VERIFY	EDIT
DELETE	SYSTEM	CONT	LIST #	SAVE #
LOAD #	MERGE #	NEW #		

Note that the commands for VERIFY, SAVE #, LOAD #, MERGE #, and NEW # have been deleted on FX-890P and Z-1.

List of Program Commands

ANGLE	BEEP	CLEAR	DIM	ERASE
END	DATA	READ	RESTORE	FOR ~ TO ~ [STEP] ~ NEXT
GOTO	GOSUB	RETURN	IF ~ THEN ~ ELSE	INPUT
LET	ON-GOTO	ON-GOSUB	PRINT, LPRINT	PRINT USING
REM	SET	STOP	READ #	WRITE #
RESTORE #	CLOSE	CLS	DEFSEG	LOCATE
DEFCHR $	POKE	TRON	TROFF	VARLIST
INPUT #	LINE INPUT #	ON ERROR GOTO	OPEN	PRINT #
RESUME	FORMAT	FILES	KILL	NAME
CHAIN	STAT	STAT CLEAR	MODE	

There are no graphic-related commands such as LINE and DRAW, CALL, SWAP, WAIT, REV, NORM, OUT, and OUTPORT supported by FX-890P and Z-1GR.

List of Built-in Functions

SIN	COS	TAN	ASN	ACS
ATN	HYPSIN	HYPCOS	HYPTAN	HYPASN
HYPACS	HYPATN	SQR	CUR	^
EXP	LOG	LN	ABS	INT
FRAC	FIX	SGN	ROUND (	RAN #
π, PI	DEG (	REC (	POL (	FACT
NPR(	NCR (	FRE	DEGR	DMS
CNT	SUMX, SUMY, SUMX2, SUMY2, SUMXY	MEANX, MEANY	SDX, SDY, SDXN, SDNY	LRA, LRB
COR	EOX, EOY	& H	DMS $ (	LEN (
MID $ (	CHR $ (	LEFT $ (	RIGHT $ (	STR $ (
VAL (	HEX $ (	ASC (	VALF (	EOF

ERL	ERR	PEEK	DSKF	TAB
INPUT $	INKEY $			

The functions INP, INPORT, POINT, and TIMER supported by FX-890P and Z-1GR are not available.

Logical Operations

NOT	AND	OR	XOR	¥
MOD				

Although not described in the BASIC manual, it is described in the operation text, but logical operators are provided.

Error Message List

OM error	SN error	ST error	TC error	BV error
NR error	RW error	BF error	BN error	NF error
LB error	FL error	OV error	MA error	DD error
BS error	FC error	UL error	TM error	RE error
PR error	DA error	FO error	NX error	GS error
FM error	OP error	AM error	FR error	PO error
DF error				

In FX-890P and Z-1GR, LB error has been deleted.

The FX-850P, FX-870P, FX-880P, FX-890P, VX-1 to 4, Z-1 and PB-1000 Series

These machines have a new implementation of BASIC, called JIS Standard BASIC by Casio. The PB-1000 has a RAM file system while the FX and VX systems retain the ten program areas of the earlier machines. The internal encoding is ASCII but the BASIC keywords and line numbers are encoded differently (line numbers can now reach up to 65535, not only 9999.) The extended character sets differ between the PB-1000 and the other machines of the series. The PB-1000 shares the PB-700 character set with special graphics while the FX, VX and Z systems show math and science symbols instead. The Z-1 and its sibling FX-890P lack the tape interface.

All machines except the PB-1000 connect to the FA-6 interface. This interface offers a higher transmission speed of 1200 bits per second. The data block format is a variant of the PB-700 scheme but the encoding of BASIC programs is different. It is possible to load a file saved with `SAVE,A` on a PB-700 into the FX-850P, and the other way round is possible, too. You have to restrict the speed to 300 bits per second (`SAVE"(S)"` and `LOAD"(S)"` on the FX-850P.) I could only partly test the tape interface with the VX-1 or FX-870P because I could only write but not read programs or data through the FA-6 interface with these machines.

The FX-850P/FX-880P systems can read tapes from the PB-100 series with special commands (`PBLOAD`, `PBGET`).

The PB-1000 has a similar connector but mechanical and electrical differences inhibit the use of the FA-6. The PB-1000 uses the FA-7 interface which offers even higher transfer rates (up to 2400 bits per second, selectable by DIP switch on the interface.) The Z-1 and FX-890P no longer support tapes but can still be used with the bas850 source text translator and a serial or USB interface.

	Mémoire		Graph Basic	Nb Lig,	Lib	Prog			Kata Kana	Nb	RP
	Base	Max									
FX-840p	3 536	36 304	✗	2	Fx ?		ASM	CASL	✓	0	-
FX-841p	3 536	36 304	✗	2	Lib ?	STAT	TABLE		✓	1	-
FX-850p	3 536	36 304	✗	2	Lib 116				✗	1	-
FX-860p	21 456	54 224	✗	2	Lib 116				✓	0	-
FX-860pvc	21 312	54 080	✗	2	Lib 116			CASL	✓	1	-
FX-880p	21 456	54 224	✗	2	Lib 116				✗	1	-
FX-890p	51 180	83 948	✓	4	Fx 3	C	ASM	CASL	✓	1	-
Z-1	18 412	51 180	✓	4	Fx 3	C	ASM	CASL	✓	1	33
Z-1GR	18 412	51 180	✓	4	Fx 3	C	ASM	CASL	✓	1	-
VX-1			✗	2						0	-
VX-2	3 392	36 160	✗	2	Fx 14			CASL	✓	1	-
FX-870p	17 179	49 947	✗	4	Fx 3	C		CASL	✓	1	-
VX-3	3 355	36 123	✗	4	Fx 3	C		CASL	✓	1	-
VX-4	4 891	37 659	✗	4	Fx 3	C		CASL	✓	1	33

2-1 The BASIC Token

ABS
ACS
ALL
AND
ANGLE
APPEND
AS
ASC
ASN
ATN

BEEP

CALC
CHAIN
CHR$
CLEAR
CLOSE
CLS
CNT
CONT
COR
COS
CUR

DATA
DEF
DEFSEG
DEG
DEGR
DELETE
DIM
DMS
DMS$
DSKF

EDIT
ELSE
END
EOF
EOX
EOY
ERASE
ERL
ERR
ERROR
EXP

FACT
FILES
FIX
FOR
FORMAT
FRAC
FRE

GOSUB
GOTO

HEX$
HYP

IF
INKEY$
INPUT
INT

KILL

LEFT$
LEN
LET
LINE
LIST
LLIST
LN
LOAD
LOCATE
LOG
LPRINT
LRA
LRB

MEANX
MEANY
MERGE
MID$
MOD

NAME
NCR
NEW
NEXT
NOT
NPR

ON
OPEN
OR
OUT

PASS
PEEK
PI
POKE
POL
PRINT
PUT

RAN#
READ
REC
REM
RENUM
RESTORE
RESUME
RETURN
RIGHT$
ROUND
RUN

SAVE
SDX
SDXN
SDY
SDYN
SET
SGN
SIN
SQR
STAT
STEP
STOP
STR$
SUMX
SUMX2
SUMXY
SUMY
SUMY2
SYSTEM

TAB
TAN
THEN
TO
TROFF
TRON

USING

VAL
VALF
VAR
VERIFY

WRITE#

XOR

2-2 How to enter BASIC Mode

① Press the MODE key and '1' in succession to enter BASIC mode. Below the LCD screen is a table showing the combinations of the MODE button and numeric keys.

② In BASIC mode, pressing the SHIFT key (red 'S' key) and the numeric key in succession selects the program number of the number that was pressed and becomes the target for editing and execution.

③ In CAL mode, if you press the SHIFT key (red 'S' key) and the number key in succession, if there is a program in the program number of the pressed number, that program is executed.

2-3 Grammar Overview

Here, basic knowledge of BASIC is omitted. The features of FX-870P / VX-4 BASIC are as follows.

- There are 10 program areas P0 to P9 that can be stored. Therefore, there is no problem even if each program has the same line number. However, there are no scoping rules for variables, and all are global variables. This is a major feature of CASIO BASIC.
- There is a data bank area (file area) F0 to F9, and BASIC allows input / output via WRITE #, READ #, etc. Therefore, the calculation results can be output to a file and saved.
- In Sharp, the label that was implemented in the initial pocket computer is not implemented, and it is specified by the line number or program number. Casio's last Pokémon FX-890P / Z-1 was the first label to be mounted.
- Sharp's pocket computer BASIC can execute machine language with the CALL instruction, where as Casio cannot execute Machine language except for a few models such as PB-1000 and FX-890P / Z-1. FX-870P / VX-4 does not officially support machine language execution, but can execute machine language routines with the hidden instruction MODE110 (execution start address).

2-2-1 Structure of sentence Each sentence (line) is composed as follows.

[line number] Command (Instruction) Operand [: Command (Instruction) Operand; [: ・・・]]]

The Line-Numbers can be 1-65535. The Line-Length was 255 Chjars.

A sentence consists of a command and an operand, separated by a colon (:). If a line number is added at the beginning of the line, it is interpreted as a program and stored in memory. If there is no line number, it is executed directly after pressing the EXE button.

2-2-2 Variables Variables are classified into four types depending on whether the data type is numeric or string, single variable or array variable.

Table 2-1. Classification of BASIC variables

		Single variable	Array variable
Data type	**Numeric**	Numeric variable	Array numeric variables
	Character (column)	Character variable	Array character variable

The naming method for variable names and array names is as follows.

① Must not contain reserved words from the beginning. Conversely, reserved words are a memory-saving specification that allows delimiters such as white space to be omitted.
② The first character string must be one of uppercase letters ('A'-'Z'), lowercase letters ('a'-'z'), or kana (ASCII code: & HA6- & HDF).
③ Except for the beginning, it must consist of uppercase letters, lowercase letters, kana, and numbers ('0'-'9').
④ The length of the string must be no more than 255 characters. The length of the **standart string A$ to Z$ must be no more than 30 characters**.

Handling of arrays is as follows.

① An array is first declared with a DIM statement.
② The array subscript is an integer greater than or equal to 0, and the fractional part is truncated.
③ The dimensions of the array are written in the CASIO manual and the range allowed by the internal stack, but in reality, 255 dimensions is the maximum in terms of work area representation. (Note 2)
④ The maximum value of the subscript is the range allowed by the storage capacity.

The used memory size of the variable is listed in (Note 1) at the end of this chapter.

Notes on variables and arrays are as follows.

① Variables and arrays are commonly used for all programs (P0-P9).
② Variables are reserved for their first use.
③ An array variable cannot be used unless an array declaration is made in the DIM statement.
④ Character variables are stored in the character data area specified by the CLEAR statement.
⑤ **Uppercase and lowercase letters are recognized as different characters.** For example, A and a are separate variables.
⑥ **Numeric variables, character variables, array numeric variables, and array character variables with the same variable name can exist simultaneously.** For example, DIM A (10) and A $ (10) can be used while using A and A$.

Care must be taken because these can cause bugs. VARLIST is a useful command for debugging because it lists the names and types of variables that have substance when executing programs.

2-2-3 Valid only in a comparison operator program. The result is -1 if true, 0 if false. Since comparison of character strings is complicated, Table 2-2 shows the operation of comparison operators depending on the data type.

Table 2-2. Comparison operator behavior

Data type	Action	Example of use	result
Numeric	Compare numerical values.	PRINT 123> 45	-1 (true)
		PRINT 123 <45	0 (false)

String	The character code sizes are compared in order from the beginning.	PRINT "ABC" <"ABD"	-1 (true)
		PRINT "DEF" <"ABC"	0 (false)
	When the character string is the same from the beginning and one is included in the other, the shorter character string is considered smaller.	PRINT "ABC"> "ABCD"	0 (false)

2-2-4 Character operators Only + (plus) of the four arithmetic operations are valid for string operations. + Performs the operation of combining left and right strings, and the result must be within 255 characters. For example, "A" + "B" results in "AB".

(Note 1) **Variable memory usage** Numeric variables and character variables are allocated to memory when they are used for the first time. The bytes used at that time are as follows.

Numeric variable:	(Variable name length + 12) bytes are secured from the work area.
Character variable:	(Variable name length + 4) bytes are secured from the work area, and (String length + 1) bytes are secured from the character area.

Array variables are allocated in memory when they are defined with a DIM statement. The bytes used at that time are as follows.

Array numeric variables:	((Variable name length + 4) + (array size * 8) + dimension * 2 + 1) bytes are secured from the work area.
Array character variable:	(Variable name length + 4) bytes are allocated from the work area, and ((array size) + dimension * 2 + 1) bytes are allocated from the character area. When a character string is assigned, the character area is used for the length of the character string.

Refer to “2-3. Variable data storage format” in “12. FX-870P / VX-4 internal information” for details of the variable storage method.

(Note 2) **Maximum number of dimensions of array variable**
The dimension of the array variable is stored in the +1 term in the used memory size of both array variables in (Note 1), that is, 1 byte. Therefore, the maximum number of dimensions of an array variable is 255. However,

- It is impossible to define 255 dimensions because of the restriction that must be declared in a DIM statement with 255 characters per line.
- In principle, 255 dimensions can be realized by directly manipulating the BASIC work area using PEEK and POKE statements. However, in order to declare DIM A $ (1,1,1, ..., 1), a huge memory of 2^{255}するには 5.8E + 78 bytes (in short, 255 bits) is required. .
- Declaration equivalent to DIM A $ (0,0,0, ..., 0) can be realized if the memory usage is taken into consideration. However, the number of elements in an array variable is 1, the substance is just a variable, and the declaration itself is meaningless. In addition, since a 255-dimensional index is calculated for accessing variables, the loss is large in terms of calculation speed. However, FX-870P / VX-4 can declare 0 (though meaningless) as the maximum DIM index.

2-4 BASIC Manual Commands

- Manual commands cannot be executed in the program.
- {} Indicates one of them. However, when executing with BASIC, {} itself is not entered.
- [] Can be omitted. However, when executing in BASIC, [] itself is not input.
- Commands marked with * can also be used in CAL mode.

Table 3. Manual Commands

Command Name	Format	Function	Example of Use

LIST
LLIST
	{LIST, LLIST}
	{
	• [Start number] [-[End number]]
	• .
	• ALL
	}
		Display all or part of the program contents on the screen. When LIST is LLIST, output from the screen is output to the printer.	1. LIST: 'Display from the top
			2. LIST 30: 'Display line number 30
			3. LIST 20-80: 'Display line numbers 20-80
			4. LIST 20-: 'Display line number 20 and later
			5. LIST -80: 'Displays from the first line to line number 80
			6. LIST.: 'Display last line processed
			7. LIST ALL: 'Display programs in all program areas

RENUM	RENUM [new line number] [, [old line number] [, incremental]]	Renumber lines at regular intervals. The default values for the new line number, old line number, and increment are 10, the first line number, and 10, respectively.	1. RENUM 100,10,10: 'Set line number 10 as new line number 100, and then renumber line numbers at intervals of 10
NEW	NEW [ALL]	Erase the program in the currently specified program area. When ALL is specified, all programs in the program area are deleted.	8. NEW: 'Erase program in specified program area 9. NEW ALL: 'Erase programs in all program areas (P0-P9)
***** **PASS**	PASS "Password"	Sets or cancels all program areas and all file areas.	10. PASS "CASIO": 'When executed first, operations such as LIST and EDIT are disabled for each area. It is canceled by executing PASS "CASIO" again.
RUN	RUN [line number]	Execute the program from the first line or specified line.	11. RUN: 'Run the program from the first line 12. RUN1000: 'Run program from line number 1000

SAVE	SAVE [ALL] " File descriptor " [, A]	Outputs the program to the file specified by the file descriptor. The target program is the program in the currently specified program area, or the program in all program areas when ALL is specified. However, ALL-designated output destinations are limited to cassette tapes. When ", A" is added, the output is ASCII. The ALL specification is not available for FX-890P and Z-1 BASIC.	13. SAVE "0: DEMO1.BAS": 'Output the program with the file name "DEMO1.BAS" in the floppy disk. 14. SAVE "CAS0: (S) DEMO2.BAS", A: 'Output the program in ASCII format with the file name "DEMO1.BAS" at normal phase and slow transfer speed (300bps) on the cassette tape. 15. SAVEALL "CAS1: (F) P09": 'Outputs the program in the entire program area with the file name "P09" with reverse phase to cassette tape and high transfer speed (1200bps)
LOAD	LOAD [ALL] " file descriptor " [, A]	Reads the program of the file specified by the file descriptor. The reading destination is the currently specified program area, or the entire program area when ALL is specified. However, ALL specification is limited to LOAD from cassette tape. When ", A" is added, ASCII format program is read. The ALL specification is not available for FX-890P and Z-1 BASIC.	1. LOAD "5, E, 8,1, N, N, N, B, N": 'Load the program from RS-232C. Refer to the file descriptor for the RS-232C settings.
MERGE	MERGE " file descriptor "	The program of the file specified by the file descriptor is mixed with the currently specified program area.	16. MERGE "0: TEST.BAS": 'Read the program of the file "TEST.BAS" in the floppy disk and mix.
VERIFY	VERIFY " file descriptor "	Check the file recorded in the cassette file. In FX-890P, Z-1, this command has been deleted.	17. VERIFY "CAS0: TEST": 'Verify that the file "TEST" on the cassette tape is recorded correctly.

EDIT	EDIT {
	• [line number]
	• .
	}
		Displays the program in the currently specified program area and enters edit mode.	18. EDIT: 'Start editing from the first line of the program
			19. EDIT 30: 'Edit line number 30
			20. EDIT .: 'Edit the last line handled

DELETE	DELETE [starting line number] [-	Delete part of the program by line number. If there is no argument, SN Error occurs.	21. DELETE 50: 'Delete line number 50

	[ending line number]]		22. DELETE 20-80: 'Do line numbers 20-80 23. DELETE 20-: 'Delete line number 20 and later 24. DELETE -80: 'Delete line number 80 from the first line
* **SYSTEM**	SYSTEM [*]	Without arguments, printer (PR) ON / OFF setting, trace mode (TR) ON / OFF setting, CLEAR statement setting, text area free capacity (FREE), variable area (V) free area capacity, characters Displays the free capacity ($) of the area. Enter the test mode with the argument "*" as a hidden command (Reference (1)).	1. SYSTEM: Displays the BASIC system settings 2. SYSTEM *: Test mode
* **CONT**	CONT	Resume execution of a program that was stopped with the STOP statement or STOP key.	1. CONT
* **LIST #**	LIST #	Displays all text data written in the data bank area "F0". When LIST is LLIST, output from the screen is output to the printer.	1. LIST #
* **SAVE #**	SAVE # " File descriptor "	Outputs the memo data written in the data bank area “F0” to the file specified by the file descriptor.	25. SAVE # "0: TEST": 'F0 contents are output to floppy with file name "TEST"
* **LOAD #**	LOAD # " File descriptor "	Read the contents of the file specified by the file descriptor into the data bank area “F0”.	26. LOAD # "0: TEST": Load the contents of the file "TEST" on the floppy disk to 'F0
* **MERGE #**	MERGE # " file descriptor "	Adds the contents of the file specified by the file descriptor to the memo data in the data bank area "F0".	1. MERGE # "0: TEST": '
* **NEW #**	NEW #	All the memo data written in the data bank area “F0” is deleted.	1. NEW #

2-5 BASIC Program Commands

- {} Indicates one of them. {} Itself is not written.
- [] Can be omitted. However, [] itself is not written. If there are "..." in [], it means that it can be recursively defined in [].
- | Means "or" and is one of the identifiers on both sides of |.
- *Italicized words* are identifiers that are not reserved words, and are constants, variables, and expressions.

Table 4. Program Commands

Command Name	Format	Function	Example of Use
ANGLE	ANGLE formula	Specify the angle unit.	1. ANGLE 0: 'DEG: degree 2. ANGLE 1: 'RAD: Radian 3. ANGLE 2: 'GRA: Grado 4. ANGLE A: Change angle unit according to 'A value * 360 deg = 2 * PI rad = 400 gra
BEEP	BEEP {[0] | 1}	Sound the buzzer.	1. BEEP: 'Sound with bass 2. BEEP 0: 'Sound with bass 3. BEEP 1:
CLEAR	CLEAR [variable area size] [, work area size]	Clear all variables and allocate memory area according to the arguments. The work area refers to the entire work area of BASIC used for I / O buffers, character operation work, FOR stack, GOSUB stack, numeric data, variable table, and character variable data (machine language is also used in PB-1000). The variable area indicates the data storage area of the last character variable (including array character variables). **Therefore, the variable area size must be smaller than**	1. CLEAR: 'Clear variable 2. CLEAR 1024: 'After clearing the variable, 1024 bytes are reserved for the variable area. 3. CLEAR 1024,2048: 'After clearing the variable, 1024 bytes and 2048 bytes are secured in the variable area and work area, respectively.

		the work area size, and a certain area must be secured in addition to the variable area. The default variable area and work area sizes are 512,1536 when VX-4 (RAM: 8KB) and RP-8 are added (RAM: 16KB), and 1024, 8192 otherwise. The size of the current work area, variable area, and free space can be determined by the SYSTEM command and the built-in function **FRE** .	
DIM	DIM array name (maximum subscript [, maximum subscript ...)	Declaring array variables. However, subscript starts from 0.	1. DIM A (5): 'Declaration of numeric variable of one-dimensional array 2. DIM B $ (2,5): 'Declaration of two-dimensional array character variable
ERASE	ERASE array-name [, array-name]	Erase the specified array variable by variable name.	1. ERASE A, B: 'Erasing array variables A and B.
END	END	Terminate the program. However, even if the program does not have an END statement, the program ends when it reaches the end of the program.	
DATA	DATA data 1 [, data 2 ...]	Used to embed data read by **READ** statement in the program.	1. DATA 10,20,30
READ	READ variable 1 [, variable 2 ...]	Store the data prepared by the **DATA** statement in a variable.	1. READ A, B, C
RESTORE	RESTORE [line number]	Specify the start line of **DATA** statement to be read by READ statement.	1. RESTORE: 'Specify the start line of the data statement 2. RESTORE 100: 'Read from the data of line

			number 100 with READ statement
FOR ~ TO ~ STEP **. . .** **NEXT**	FOR variable = initial value TO final value [STEP increment value] . . . NEXT [variable] (formula)	Repeat the FOR and NEXT statements from the initial value until the final value is not exceeded while adding the increment value (1 if there is no STEP or less).	1. FOR I = 1 TO 10 SUM = SUM + A (I) NEXT I

GOTO	GOTO {
	• Branch precedence number
	• #Program area number
	}
		Jumps unconditionally to the specified branch precedence number or the first line of the program area.	1. GOTO 80: 'Jump to line number 80
			2. GOTO # 7: 'Jump to the first line of program area 7

GOSUB	GOSUB {
	• Branch precedence number
	• #Program area number
	}
		Calls a subroutine starting from the specified branch precedence number or the first line of the program area. Even if the program area changes, variable definitions and their values are inherited.	1. GOSUB 100
			2. GOUB # 5

RETURN	RETURN [{ • Branch precedence number • #Program area number }]	Return to the first line of the branch preceding number and program area number specified from the subroutine. When the return destination is omitted, it returns to the next sentence after the one that called the subroutine with a **GOSUB** statement. * To make the program easier to read, it is better not to specify the return destination.	1. RETURN 2. RETURN 20 3. RETURN # 1

IF ~ {
• **THEN**
• **GOTO**
} ELSE
	IF conditional statement {
	• THEN {
	○ Sentence [: sentence]
	○ Branch precedence number
		When the conditional statement is true, the statement below THEN is executed or jumps to the destination specified by the GOTO statement.	1. IF A> = 100 THEN 50 ELSE 100
			2. IF B = 0 THEN X = 10 ELSE Y = B
			3. IF C = 1 THEN GOSUB 500: 'GOSUB can be used in the statement

	○ #Program area number
	}
	• GOTO {
	○ Branch precedence number
	○ #Program area number
	}
	} [ELSE {
	• Statement [: execute]
	• Branch precedence number
	• #Program area number
	}]
		If the conditional expression is false and there is a statement below ELSE, the statement below ELSE is executed or jumped to the jump destination.	4. IF D <> 50 THEN # 9

INPUT	INPUT ["message sentence 1" {; |,}] variable 1 [[, "message sentence 2" {; |,}] variable 2 ...]	Input data from the keyboard to the specified variable. If a message text is given as an argument before the variable, the data can be entered after the message text is displayed. When the comma after the message text is ";", "?" Is added to the message text, and when it is "," nothing is added and the input operation starts.	1. INPUT A, B, C 2. INPUT "X ="; X 3. INPUT "A"; A, "B"; B, "C"; C
LET	LET variable = {assigned value | expression}	Assign the assignment value on the right side or the calculation result of the expression to the variable on the left side. The assignment statement can omit LET itself.	1. LET A = 10 2. A $ = "CASIO" 3. X = Y * Z / 2

ON-GOTO	ON Formula GOTO {
	• Branch precedence number
	• #Program area number
	} [, {
	• Branch precedence number
		ON Jumps to the jump destination corresponding to the value of the formula below. The branch destination is specified when the mathematical formula is 1, 2, 3, ... from the top. When the	1. ON A GOTO 100,200,, 300: Jumps to line number 300 when 'A is 3 and does not jump when 4
			2. ON X + Y GOTO 100, # 6, # 7

	• #Program area number } · · ·]	branch destination is not defined, the command immediately after this instruction is executed without jumping.	

ON-GOSUB	ON Formula GOSUB {
	• Branch precedence number
	• #Program area number
	} [, {
	• Branch precedence number
	• #Program area number
	} · · ·]
		Calls a subroutine corresponding to the value of the expression below ON. Subroutines are specified when the formula is 1, 2, 3, ... from the top. When no subroutine is defined, nothing is called and the command immediately after this command is executed.	1. ON A GOSUB 100,200,, 300: When A is 3, do not GOSUB, and when 4, call the subroutine of line number 300
			2. ON X + Y GOSUB 100, # 6, # 7

PRINT LPRINT	[PRINT | LPRINT] [{
	• TAB (tab specification)
	• Formula
	• String
	• variable
	}] [{; |,} [{
	• TAB (tab specification)
	• Formula
	• String
	• variable
	}] · · ·]
		Displays output elements such as formulas, strings, and variable values. If PRINT is set to LPRINT, the output is changed from the screen to the printer.	1. PRINT: 'Do line feed only
			2. PRINT A, B, C
			3. PRINT "X ="; X;: 'Add a semicolon ";" at the end to avoid line breaks
			4. PRINT TAB (5); "CASIO": 'Output 5 blanks and then the string "CASIO"

PRINT USING	PRINT USING "format specification"; output element	Display output elements according to format specification. USING and below are also applicable to **LPRINT** and **PRINT #** .	1. PRINT USING "& &"; A $: 'A $ displays only the length of & &. 2. PRINT USING "###. ##"; X: '###. ## displays a numeric value, and invalid digits in the integer part display a blank. # Includes a sign and a numeric value. If the specified format cannot be displayed, it ignores the format specification and displays a numeric value with a leading%.
REM	{REM | '} Annotation	Represents an annotation (comment)	1. REM program for matrix calculation

		and does nothing. Apostrophe "'" is an abbreviation for REM.	2. 'This is comment

SET	SET {
	• F {one character of 0-9}
	• E {one character of 0-9}
	• N
	}
		Specify the output format of numeric data. F specification specifies the number of digits after the decimal point, E specification specifies the number of significant digits, and N cancels the specification.	1. SET F3: '

STOP	STOP	Pause program execution. The program resumes from where it was interrupted by the manual command CONT.	
READ #	READ # Variable 1 [, Variable 2…	Reads the memo data written in the data bank area into a variable. The default data bank area is "F0", but can be changed with the **RESTORE #** statement.	1. READ # A $, X
WRITE #	WRITE # [Data 1] [, Data 2 · · ·]	Delete or rewrite data in the data bank area. A line feed is output after each data is output. The default data bank area is "F0", but can be changed with the **RESTORE #** statement. * An FC error occurs when attempting to execute as a manual command. When the WRITE # statement is executed by the program, the data bank area is cleared, but it is not cleared by the subsequent WRITE # statement, and additional writing is performed.	1. WRITE #: 'Delete 2. WRITE # "CASIO Z-1GR": 'rewrite 3. WRITE # A $, B: 'Output of character variable A and numeric variable B

RESTORE #	RESTORE # [("file area name")] ["search string"] [, {0 | 1} [, GOTO { • Branch precedence number • #Program area number }]]	Switch the file area for **READ #** and **WRITE #** . In addition, the "search character string" in the designated file area is searched, and the data read first by the **READ #** statement is changed to start from the search character string. The third argument 0 or 1 specifies the data reading start position. 0 is the same as when nothing is specified, and the data including the search character string at the head is set as the reading start position. When 1, the search character string is searched and read with **READ #** from the beginning of the line containing the character. When "search string" is not found, if there is a **GOTO** option, jump to the specified jump destination. If there is no **GOTO** option, a DA error will occur.	1. RESTORE # ("F1"): 'Specify the target file area for READ # and WRITE # to F1 2. RESTORE # "START": '"START" position is the data reading start position 3. RESTORE # ("F1") "START": ' 4. RESTORE # "ORANGE", 0: Same as' RESTORE "ORANGE", the first data read with READ # is "ORANGE". 5. RESTORE # "ORANGE", 1: 'The beginning of the line containing "ORANGE" is the position of the data to be read first.
CLOSE	CLOSE	Close the current file and stop using the I / O buffer.	
CLS	CLS	Clear display screen.	
DEFSEG	DEFSEG = segment value	Sets the base address when executing the **PEEK** function or **POKE** statement (maybe MODE110 statement).	1. DEFSEG = 0: 'BANK1 RAM (default value). & H1000 is the same as the x86 CPU segment register, and DEFSEG * 16 is the base address. 2. DEFSEG = & H1000: 'The base address is the first (& H38000) of the 30-pin I / 0 area in the I / O space of BANK3. Reading and writing of & H38000 to & H38007

			can be executed with PEEK and POKE at addresses 0 to 7. & H1000 and above are all the same.
LOCATE	LOCATE X coordinate, Y coordinate	Move the cursor to the specified position on the virtual screen.	1. LOCATE 10,0
DEFCHR $	DEFCHR $ (code) = "character form"	Sets the display pattern according to the character form of the specified code. You can specify 4 codes from & HFC (252) to & HFF (255). The character form is a 12-character hexadecimal code, and two characters from the beginning are assigned from left to right.	1. DEFCHR $ (252) = "0F0F0F0F0F0F": 'The lower half is a black pattern 2. DEFCHR $ (252) = "0F0F0F000000": 'Black pattern in the lower left half
POKE	POKE address, data	Write data to the address specified by the formula. The actual address is the base address specified in the **DEFSEG** statement plus the address of the PEEK statement argument.	1. POKE & H7000,0
TRON	TRON	Set the BASIC program to trace mode.	
TROFF	TROFF	Release the BASIC program from trace mode.	
VARLIST	VARLIST	Displays all variable names and array names that currently exist.	
INPUT #	INPUT # file number, variable name 1 [, variable 2 ...	Reads data from the sequential file with the file number declared in the OPEN statement.	1. INPUT # 1, A: '
LINE INPUT #	LINE INPUT # file number, character variable name 1	Reads one line of character string data from the sequential file with the file number declared in the **OPEN** statement.	1. LINE INPUT # 1, A $: '

ON ERROR GOTO	ON ERROR GOTO branch precedence number	Specify the branch destination when an error occurs.	
OPEN	OPEN " file descriptor " [FOR {INPUT | OUTPUT | APPEND} AS [#] file number]	Open the file. **INPUT**, **OUTPUT**, and **APPEND** specify the input, output, and additional write modes, respectively.	1. OPEN "DATA1.DAT" FOR INPUT AS # 1: '

PRINT #	PRINT # file number, [{
	• TAB (tab specification)
	• Formula
	• String
	• variable
	}] [{; |,} [{
	• TAB (tab specification)
	• Formula
	• String
	• variable
	}] · · ·]
		Outputs output elements such as mathematical expressions, character strings, and variable values to the sequential file with the file number declared in the **OPEN** statement.	1. PRINT # 1, A $

RESUME	RESUME [{NEXT | Return line number}]	Return from error handling routine. If NEXT or return destination is omitted, return to the statement where the error occurred.	1. RESUME NEXT: 'Return to the statement following the statement where the error occurred 2. RESUME 100
FORMAT	FORMAT	Format the floppy disk. There is no / 6, / 9, / M option to specify the floppy capacity like FX-890, Z-1.	
FILES	FILES [" file descriptor "]	Displays the file name, attribute, used capacity, etc. specified by the file descriptor in the floppy disk. * ,? wildcards can be used for file descriptors.	1. FILES 2. FILES "0: TEST.DAT" 3. FILES "0: *. DAT"
KILL	KILL " File descriptor "	Delete the file specified by the file descriptor in the floppy disk. * ,? wildcards can be used for file descriptors.	1. KILL "0: TEST.DAT" 2. KILL "0: *. DAT"
NAME	NAME "old file descriptor " AS "new file descriptor"	The file specified by the old file descriptor on the floppy disk is	1. NAME "0: TEST.BAS" AS "0: NEW.BAS"

		changed to the file name of the new file descriptor.	
CHAIN	CHAIN " File descriptor "	Reads and executes the program specified by the file descriptor in the current program area.	1. CHAIN "CAS0: TEST" 2. CHAIN "0: TEST.BAS"
STAT	STAT X data [, Y data] [; Frequency]	Enter statistical data.	1. STAT 1,3; 10
STAT CLEAR	STAT CLEAR	Clear (initialize) the statistical processing function.	
MODE	MODE formula	Hidden instructions not in the CASIO manual. Refer to the usage examples for arguments and grammar. If the argument is out of range, it will be "BS error".	1. MODE 10: 'Perform rounding after four arithmetic operations. 2. MODE 11: 'Do not perform rounding after the four arithmetic operations. 3. MODE110 (*Addr*): 'Call the machine language at *Addr* 's address. 4. MODE {200 | 201} (*Tr* , *Sf* , *Sc*): 'Floppy disk sector READ, WRITE command. *Tr* is 0-79 for truck, *Sf* is 0-1 for surface, *Sc* is 1-8 for sector. It is unknown whether 200 or 201 of the first argument is **READ.** 5. MODE A: The above processing is executed according to the value of 'A. However, with A = 110, 200, 201, the following argument is required, so "SN error".

2-6 File Descriptor

For the FX-870P and VX-4, three file descriptors can be specified as devices: Floppy disk, Cassette tape, and RS-232C.

For a floppy, it is "0: file name".

In the case of cassette tape, it is represented by "CAS {0 | 1} ({F | S}): file name", and the numbers are phase designation when reading from MT: 0: normal phase, 1: reverse phase, in parentheses The alphabetical characters are F: 1200bps and S: 300bps in transfer rate specification, and are described as "CAS0: (F) TEST1".

In the case of RS-232C, “COM0: communication parameter ” (for example, “COM0: 6, E, 8, 1, N, N, N, B, N”).

Communication parameters

Each of the nine settings is represented by one character, and is described by a character string with a comma inserted between each character:

The **first parameter** is the communication speed setting, which is 1,2,3, ..., 7. If this is *n* , the communication speed is set to 75 * 2 ^ *n* bps. Specifically:

1: 150 bps	4: 1200 bps	6: 4800 bps
2: 300 bps	5: 2400 bps	7: 9600 bps
3: 600 bps		

The **second parameter** is the parity setting. One of the three characters E, O, and N represents even parity, odd parity, and non-parity, respectively.

The **third parameter** is the data length setting. The data length is 7 bits or 8 bits in either of 7 and 8 characters.

The **fourth parameter** is the stop bit setting. Stop bit is 1 bit or 2 bit in either of 1 or 2 characters.

The **fifth parameter** is the CTS setting. CTS represents ON or OFF for either of the two characters C and N. CTS is an abbreviation of “Clear To Send”. DCE (Data Circuit terminating Equipment; here, the other party) informs DTE (Data Terminal Equipment; here the Pokécon) that it is ready to receive. In the 3-wire system with audio mini plugs, only RxD, TxD, and SG (signal ground) signal lines are required, so CTS, DSR, and CD must be turned off.
The sixth parameter is the DSR setting. DSR is ON or OFF for either of the two characters D and N. DSR is an abbreviation for “Data Set Ready”. DCE informs the DTE that the operation is ready.

The **seventh parameter** is the CD setting. One of the two letters C and N indicates that CD is ON or OFF, respectively. CD is an abbreviation for "Carrier Detect" and is a signal that informs that there is data to be transmitted by DCE to DTE.

The **eighth parameter** is the soft flow control setting. Soft flow control indicates ON or OFF for either of the two characters B and N. Soft flow control is control in which Xoff is transmitted to DCE and DCE transmission is interrupted until Xoff is transmitted when the buffer is likely to overflow during data reception.

The **ninth parameter** is SI / SO setting. SI / SO indicates ON or OFF with either of the two letters S and N. With SI / SO control, data length is 7 bits and half-width kana is communicated.After receiving SI (14), the 8th bit is interpreted as 1 and data is received.After

receiving SO (15), Protocol to return to normal mode, receiving 0th return bit as 0. Therefore, SI / SO control is not required when the data length is 8 bits.

For **example**: "6, E, 8,1, N, N, N, B, N" is communication speed 4800 bps, even parity, data length 8 bit, stop bit 1 bit, CTS: OFF, DSR: OFF, CD: It means OFF, soft flow control: ON, SI / SO: OFF.

2-7 BASIC Built-in Functions

Internal functions are classified as follows according to the return value.

- Numeric functions
- Hex prefix
- Character functions
- Other functions

Here, there are the following notes.

- In numeric functions, except for ROUND (, DEG (, REC (, POL (, NPR (, NCR (), parentheses () can be omitted when using numerical values or variables as mathematical expressions.
- As a rule, the accuracy is ± 1 in the 10th digit of the mantissa.
- BS error occurs when the arguments of NPR (, NCR () are $n = 0$ and $r \neq 0$.
- In FX-890P and VX-4, calculation is normally performed with 13 digits in the mantissa, and the result is rounded and the result is displayed in 10 digits for the mantissa + 2 digits for the exponent.

Table 5. Mathematik-Commands

Command Name	Function Type	Format	Function
SIN	Numeric functions	SIN (Formula)	Sine function SIN. | Formula | <1440 ° (8π rad, 1600 grad)
COS	Numeric functions	COS (formula)	Cosine function COS. | Formula | <1440 ° (8π rad, 1600 grad)
TAN	Numeric functions	TAN (formula)	Tangent function TAN. | Formula | <1440 ° (8π rad, 1600 grad). However, MA error occurs when the argument is an odd multiple of 90 ° and the function diverges at ∞.
ASN	Numeric functions	ASN (Formula)	Inverse sine SIN^{-1}, ARCSIN. | Formula | <= 1, -90 ° <= ASN <= 90 °
ACS	Numeric functions	ACS (formula)	Inverse cosine function COS^{-1}, ARCCOS. | Formula | <= 1, 0 ° <= ACS <= 180 °
ATN	Numeric functions	ATN (Formula)	Inverse tangent function TAN^{-1}, ARCTAN. | Formula | <1, -90 ° <ACS <90 °
HYP SIN	Numeric functions	HYPSIN (Formula) or HYP SIN (Formula)	Hyperbolic sine function SINH. | Formula | <= 230.2585092
HYP COS	Numeric functions	HYPCOS (formula) or HYP COS (formula)	Hyperbolic cosine function COSH. | Formula | <= 230.2585092

HYP TAN	Numeric functions	HYPTAN (formula) or HYP TAN (formula)	Hyperbolic tangent function TANH. | Formula | <1E100
HYP ASN	Numeric functions	HYPASN (Formula) or HYP ASN (Formula)	Inverse hyperbolic sine function $SINH^{-1}$. | Formula | <5E99
HYP ACS	Numeric functions	HYPACS (Formula) or HYP ACS (Formula)	Inverse hyperbolic cosine function $COSH^{-1}$. | Formula | <5E99
HYP ATN	Numeric functions	HYPATN (Formula) or HYP ATN (Formula)	Inverse hyperbolic tangent function $TANH^{-1}$. | Formula | <1
SQR	Numeric functions	SQR (formula)	Square root √. Formula> = 0
CUR	Numeric functions	CUR (formula)	Cubic root ^{3}√. | Formula | <1E99
^	Numeric functions	*x* ^ *y*	Power. ; *X* , *y* in the formula, *x* when <0, *y* must become an integer.
EXP	Numeric functions	EXP (formula)	An exponential function whose base is the natural constant *e* (2.718281828 ...). -1E100 <formula <= 230.2585092
LOG	Numeric functions	LOG (formula)	Logarithm with base 10 and common logarithm. Formula> 0
LN	Numeric functions	LOG (formula)	The base is the logarithm of *e* , the natural logarithm. Formula> 0
ABS	Numeric functions	ABS (formula)	| Formula |. Gives the absolute value of the formula.
INT	Numeric functions	INT (formula)	Integer function. Gives the largest integer that does not exceed the value of the formula.
FRAC	Numeric functions	FRAC (formula)	Gives the fractional part of the formula.
FIX	Numeric functions	FIX (formula)	Gives the integer part of the formula.
SGN	Numeric functions	SGN (Formula)	Gives the sign of the formula. When formula> 0, 1 is returned. When formula = 0, 0 is returned. When formula <0, -1 is returned.
ROUND (	Numeric functions	ROUND (formula, digit)	Gives the value of the mathematical expression rounded to the specified digit (rounded). | Digit | <100 rounds 10 ^ specified digits. For example, ROUND (1234.56, -2) = 1234.6
RAN #	Numeric functions	RAN #	Give a random number within 10 digits after the decimal point. 0 <= RAN # <= 0.999,999,999,9
π	Numeric functions	PI	Gives an approximate number of pis. The value of π takes 3.1415926536 internally.
DEG (	Numeric functions	DEG (degree [, minute [, second]])	Converts a hexadecimal number to a decimal number. DEG (a, b, c) = a + b / 60 + c / 3600 | DEG (a, b, c) | <10 ^ 100

REC (	Numeric functions	REC (*r* , *ϑ*) where *r* and *ϑ* are mathematical expressions	The two-dimensional polar coordinate representation given by the radius *r* and the argument *ϑ* is converted into Cartesian coordinates (*x* , *y*). As a function value, x coordinate *x* is returned, *x* is stored in variable X, and *y* is stored in variable Y. Where 0 <= *r* <10 ^ 100, | *ϑ* | <1440 ° (8π rad, 1600 grad)
POL (	Numeric functions	POL (*x* , *y*) where *x* and *y* are mathematical expressions	Converts a two-dimensional orthogonal coordinate representation given by x-coordinate *x* and y-coordinate *y* to polar coordinates (*r* , *ϑ*). As a function value, the radius *r* is returned, the radius *r* is stored in the variable X, and the argument *ϑ* is stored in the variable Y. Where | *x* | <10 ^ 100, | *y* | <10 ^ 100, | *x* | + | *y* |> 0 and -180 ° < *ϑ* <= 180 °
FACT	Numeric functions	FACT (formula)	Gives the factorial of the formula, *n* ! However, 0 <= Formula <= 69 and an integer.
NPR (	Numeric functions	NPR (*n* , *r*)	Returns a permutation that selects from *r* different *n* . NPR (*n* , *r*) = *n* P *r* = *n* ! / *R* !. However, 0 < *r* <= *n* <10 ^ 100, and *n* and *r* are both positive integers.
NCR (	Numeric functions	NCR (*n* , *r*)	Returns a combination that selects *r* from *n* different numbers . NPR (*n* , *r*) = *n* C *r* = *n* ! / (*R* ! (*N* - *r*)!). However, 0 < *r* <= *n* <10 ^ 100, and *n* and *r* are both positive integers.
FRE	Numeric functions	FRE (argument)	Gives the size of the memory area according to the argument. 1 <= Argument <= 5, 1: Size of unused memory in the entire program / memo data area, 2: Size of the entire work area, 3: Size of the entire character area, 4: Unused size in the work area Used memory size, 5: Size of unused memory when character area is free
DEGR	Numeric functions	DEGR (hexadecimal number)	*Ab.Cdefgh* · · · numbers represented by *ab* degrees to, *cd* minute, *Ef.Gh* · · · converting the 60 decimal likened to the second decimal. It is equal to DEG (*ab* , *cd* , *ef.gh* ...).
DMS	Numeric functions	DMS (formula)	The inverse function of DEGR, which converts decimal numbers to hexadecimal numbers. Decimal number is *converted* to a value represented by *ab.cdefgh* ... , *ab* is in degrees, *cd* is in minutes, *ef.gh* ... is in seconds.
CNT	Numeric functions	CNT	Gives the number of statistically processed data.
SUMX **SUMY** **SUMX2** **SUMY2** **SUMXY**	Numeric functions	SUMX SUMY SUMX2 SUMY2 SUMXY	Gives the sum of X data. Gives the sum of Y data. Gives the sum of squares of X data. Gives the sum of squares of Y data. Gives the product sum of X data and Y data.
MEANX **MEANY**	Numeric functions	MEANX MEANY	Give the average value of X data. Give the average value of Y data.
SDX **SDY**	Numeric functions	SDX SDY	Gives the sample standard deviation of the X data. SDX = SQR (MEANX2-MEANX ^ 2) Gives

SDXN **SDYN**		SDXN SDYN	the sample standard deviation of Y data. SDY = SQR (MEANY2-MEANY ^ 2) Gives the standard deviation of the X data. SDXN = SQR (CNT / (CNT-1)) * SDX Gives the standard deviation of the Y data. SDYN = SQR (CNT / (CNT-1)) * SDY
LRA **LRB**	Numeric functions	LRA LRB	Find the linear regression constant term. Find the linear regression coefficient.
COR	Numeric functions	COR	The correlation coefficient (γ) is obtained based on the statistically processed data.
EOX **EOY**	Numeric functions	EOX argument (formula) EOY argument (formula)	Based on the statistically processed data, an estimated value of X for Y is obtained. Based on the statistically processed data, an estimated value of Y for X is obtained.
&H	Hex prefix	& H hexadecimal string	Converts the hexadecimal string following "& H" to hexadecimal (signed 2 byte integer). & HFF = 255
DMS$	Character functions	DMS $ (Formula)	Converts a decimal number given as an expression into a character string in hexadecimal notation. | Formula | <10 ^ 5, degree minute second display.
LEN	Character functions	LEN (character expression)	Returns the length of the string stored in the character expression.
MID$	Character functions	MID $ (character expression, position [, number of characters]) where the position and number of characters are mathematical expressions	Returns a string starting at the specified position in the string of the character expression. When the number of characters is specified, the character string of the specified number of characters is returned from the start position. When the number of characters is omitted, the character string from the specified position to the end is returned.
CHR$	Character functions	CHR $ (Formula)	Returns the character code character of the formula. 0 <= Formula <256
LEFT$	Character functions	LEFT $ (character expression, number of characters)	Returns the character string for the specified number of characters from the left of the character string in the character expression.
RIGHT$	Character functions	RIGHT $ (character expression, number of characters)	Returns the character string for the specified number of characters from the right of the character string in the character expression.
STR$	Character functions	STR $ (Formula)	Returns the value of the formula converted to a string.
VAL	Character functions	VAL (character expression)	Returns a character expression that represents a number converted to a number.
HEX$	Character functions	HEX $ (formula)	Returns the numeric value converted to a 4-digit hexadecimal string. -32769 <Formula <65536
ASC	Character functions	ASC (character expression)	Returns the character code of the first character of the character expression.

VALF	Character functions	VALF (character expression)	Returns the evaluation value of a mathematical expression expressed as a character expression.
EOF	Other functions	EOF (file number)	Indicates the end of reading the file.
ERL	Other functions	ERL	Returns the line number of the line where the error occurred.
ERR	Other functions	ERR	After an error occurs, an error code corresponding to the content is returned.
PEEK	Other functions	PEEK (address)	Returns the contents of the specified address.
DSKF	Other functions	DSKF	Returns the number of remaining clusters on the floppy disk. One cluster is 1 Kbyte.
TAB	Other functions	TAB (formula)	Display to the horizontal position specified by the formula or move the print position of the printer.
INPUT$	Other functions	INPUT $ (formula [, file number])	Reads and returns a string of the number of characters specified by the formula from the keyboard or the file with the opened file number.
INKEY$	Other functions	INKEY $	Returns one character of the key being pressed when this function INKEY $ is executed. When not pressed, it stops execution like INPUT and does not wait for input, but returns null "". Refer to the key code table by INKEY (191DH) of FX-870P / VX-4 internal information for return value .

2-8 BASIC Logical Operations, etc.

Logical operators are prepared. Can also be used in CAL mode.

Table 6. Logical Operators and Others

Operator	Operation Type	Format	Function	Example of Use
NOT	logic	NOT ***A***	Returns the bit inversion of A. The argument type is a signed 16-bit integer (-32768 to 32767; & H8000 to & H7FFF).	A = NOT 123: '
AND	logic	***A*** AND ***B***	Returns the logical AND of A and B. The argument type is a signed 16-bit integer (-32768 to 32767; & H8000 to & H7FFF).	A = B AND C: '
OR	logic	***A*** OR ***B***	Returns the logical OR of A and B. The argument type is a signed 16-bit integer (-32768 to 32767; & H8000 to & H7FFF).	A = B OR & H8000: '
¥	Numeric	***A*** ¥ ***B***	Returns the value obtained by rounding off the decimal part of the result of dividing A and B into integers.	A = 16.1 ¥ 3.5: returns' 5
MOD	Numeric	***A*** MOD ***B***	The remainder when A and B are converted to integers and then divided.	A = B MOD 3: '

2-9 Arithmetic Priority

The priority of calculation in BASIC and CAL mode is as follows.

Table 7. Logical Operators

Priority	Operation Type	Symbol
1	brackets	()
2	function	SIN, COS, etc.
3	Power	^
4	Sign	+-
5	Multiplication and division	* /
6	Addition and subtraction	+-
7	Comparison operator	= <>> <> <= <<= =>> =
8	Logical operators	NOT AND OR XOR

note:

① For non-functions, if the precedence is the same, the expression is computed from left to right. Unlike normal mathematical notation, it is also applied to the power (^). For example, 3 ^ 3 ^ 2 = (3 ^ 3) ^ 2 = 729.

② For complex functions, it is computed from right to left in the expression. For example, SIN COS 60 = SIN (COS (60)).

③ Comparison operators cannot be used with BASIC manual commands.

③ The priority between logical operators is ①NOT, ②AND, ③OR, and XOR.

2-10 BASIC Error Messages

Table 8. FX-890P Error Messages

Error code	Error message	Error Contents	Workaround
1	**OM error**	1. Memory over or system overflow. 2. A value that cannot secure memory was set in the CLEAR statement.	1. Shorten the program. Consider the dimensions of the array. Consider the dimensions of the array. 2. Consider the value in the CLEAR statement. 3. If RAM is not expanded, expand it.
2	**SN error**	Incorrect command or statement format.	1. Check the spelling of the instruction. 2. Check the program input.
3	**ST error**	The character length exceeds 255 characters.	Limit the length of characters to 255 characters.
4	**TC error**	The formula is too complex.	Separate the expressions.
5	**BV error**	1. I / O buffer overflowed. 2. One line is 256 bytes or more. Or you entered more than 256 characters.	1. Reduce the baud rate of RS-232C. 2. Enter up to 255 characters per line.
6	**NR error**	1. I / O is not ready for input / output. 2. An attempt was made to access a file that was not opened.	1. Check I / O connection and power supply. 2. Set a floppy disk in the MD-120. 3. Open the file correctly.
7	**RW error**	An error occurred during I / O device operation.	Check the I / O device.
8	**BF error**	There is an error in the file name specification.	Check the file name.
9	**BN error**	There is an error in the file number specification.	Check the file number specification.
10	**NF error**	The specified file name cannot be found.	1. Check the file name again. 2. Check the file attributes.
11	**LB error**	There is no power supply for MD-110S.	1. Replace the battery with a new one. 2. Use an AC adapter.
12	**FL error**	1. An attempt was made to write to a floppy disk when there was no space to write. 2. One program file exceeds approximately 64K bytes. 3. The total size of the array exceeds 64K bytes.	1. Delete unnecessary files with the KILL statement to increase the free space. 2. Use a new formatted floppy disk. 3. Reduce the size of one file. 4. Reduce the size of the array.
13	**OV error**	The calculation result or entered numerical value exceeded the allowable range.	Consider the numbers that will be calculated.
14	**MA error**	1. Mathematical errors such as division by zero. 2. The function argument exceeds the calculation range.	Consider formulas and numerical values.

15	**DD error**	An attempt was made to double-define the same sequence.	1. Do not use the same array. 2. Once the array is cleared with the ERASE instruction, it is redefined.
16	**BS error**	The subscript or parameter exceeds the specified range.	1. Consider subscript parameters. 2. Increase the array.
17	**FC error**	1. There is an error in the way functions and statements are called. 2. An attempt was made to execute a statement that cannot be used in direct mode. Or vice versa. 3. An attempt was made to execute a statement that cannot be executed in CAL mode. 4. Tried to undefined array.	1. Review argument values and statements. 2. Check the grammar as some can only be used in program mode and direct mode. 3. Check the sentence. 4. Use after defining the array in the DIM statement.
18	**UL error**	1. There is no line number specified by GOTO, GOSUB, etc. 2. You entered a statement without entering a line number in BASIC EDIT mode.	1. Check the line number. 2. Be sure to include the line number.
19	**TM error**	1. The variable type does not match in the right side, left side, or function argument of the expression. 2. An attempt was made to read character data into a numeric variable with a READ statement. 3. An attempt was made to read character data into a numeric variable with the INPUT # statement.	Check the type of the right and left sides of the expression.
20	**RE error**	There is a RESUME statement even though control was not transferred to the error handling routine.	Consider where to use the RESUME statement.
21	**PR error**	1. An invalid command or operation was performed when PASS was set. 2. An attempt was made to write to a write-protected floppy disk.	1. Cancel PASS. 2. Release write protection and set to write mode.
22	**DA error**	A READ statement was executed when there was no data to read.	1. Check the DATA statement. 2. Check the READ statement.
23	**FO error**	1. There is no FOR statement for the NEXT statement. 2. CLEAR statement and ERASE statement are included in the FOR-NEXT loop.	1. Check the combination of FOR and NEXT statements. 2. Delete the CLEAR and ERASE statements in the loop.
24	**NX error**	There is no NEXT statement for the FOR statement.	Check the combination of NEXT and FOR statements
25	**GS error**	1. GOSUB statement and RETURN statement do not correspond correctly. 2. There is a CLEAR statement at the destination.	1. Check the correspondence between GOSUB statement and RETURN statement. 2. Delete the CLEAR statement at the jump destination.

26	**FM error**	The floppy disk is not formatted. Or the format is broken.	Always format a new floppy disk.
28	**OP error**	An attempt was made to reference a file that was not opened. Or tried to OPEN twice.	Be sure to execute the file after executing the OPEN statement. To OPEN a file that has already been opened, close it once.
29	**AM error**	An attempt was made to use an output command for an input open. Or vice versa.	Use input and output commands correctly.
30	**FR error**	The RS-232C port detected a framing error.	Check the RS-232C connection and data transfer method.
31	**PO error**	1. The RS-232C port detected a parity error or overrun error. 2. There was a defect in reading the cassette tape.	1. Check the RS-232C connection and data transfer method. 2. Reduce the transfer speed. 3. Adjust the cassette tape volume. 4. Invert the cassette tape phase setting. 5. Clean the cassette tape head.
32	**DF error**	1. An undefined command was sent to FDD. 2. An error occurred in the drive device.	1. Check the command for FDD. 2. The contents of the floppy disk are not guaranteed. If you still get this error after trying several times, contact CASIO.

2-11 Character Code Table

Table 9. Character Code Table

1.The actual shape of & H60 is a mirrored version of the characters in the table.
2.The actual shape of & H86 is 8 x 6 dots, "AA55AA55AA55" Ichimatsu pattern.
3.The shape of the characters & HE0 and & HE1 is slightly different.
4.Characters with pink background are special characters.
5.Other than special characters can be printed with FP-40 and FP-100.
6. The four characters & HFC to & HFF are user-defined characters, and the character pattern is defined by DEFCHR $.

		Upper 4 bits															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Subordinate 4 bit	0	(NULL)		SP C	0	@	P	‘	p	Å	0	SP C	−	ﾀ	ﾐ	≥	×
	1		(DEL)	!	1	A	Q	a	q	∫	1	。	ｱ	ﾁ	ﾑ	≤	円
	2	(L.TOP)	(INS)	”	2	B	R	b	r	√	2	「	ｲ	ﾂ	ﾒ	≠	年
	3			#	3	C	S	c	s	´	3	」	ｳ	ﾃ	ﾓ	↑	月
	4			$	4	D	T	d	t	∑	4	、	ｴ	ﾄ	ﾔ	←	日
	5	(L.CAN)		%	5	E	U	e	u	Ω	5	・	ｵ	ﾅ	ﾕ	↓	千
	6	(L.END)		&	6	F	V	f	v	■	6	ﾖ	ｶ	ﾆ	ﾖ	→	万
	7	(BEL)		’	7	G	W	g	w	■	7	ｧ	ｷ	ﾇ	ﾗ	π	£
	8	(BS)		(	8	H	X	h	x	α	8	ｨ	ｸ	ﾈ	ﾘ	♠	¢
	9	(TAB)		)	9	I	Y	i	y	β	9	ｩ	ｹ	ﾉ	ﾙ	♥	±
	A			*	:	J	Z	j	z	γ	+	ｪ	ｺ	ﾊ	ﾚ	♦	干
	B	(HOME)		+	;	K	[	k	{	ε	-	ｫ	ｻ	ﾋ	ﾛ	♣	0
	C	(CLS)	CURSOR (→)	,	<	L	¥	l	¦	θ	n	ｬ	ｼ	ﾌ	ﾜ	□	(US R1)
	D	(CR)	CURSOR (←)	-	=	M	]	m	}	μ	x	ｭ	ｽ	ﾍ	ﾝ	○	(US R2)
	E		CURSOR (↑)	.	>	N	^	n	~	σ	-1	ｮ	ｾ	ﾎ	゛	△	(US R3)
	F		CURSOR (↓)	/	?	O	_	o		φ	÷	ｯ	ｿ	ﾏ	゜	╲	(US R4)

Character Code Table

High-order digit →

Low-order digit →

	HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
0	0			SPC	0	@	P	`	p	Ä	0	SPC	-	タ	ミ	≥	×
1	1		DEL		1	A	Q	a	q	∫	1	°	ア	チ	ム	≤	円
2	2	LINE TOP	INS	“	2	B	R	b	r	√	2	｢	イ	ツ	メ	±	年
3	3			#	3	C	S	c	s	´	3	｣	ウ	テ	モ	↑	月
4	4	SHIFT RELEASE			4	D	T	d	t	∑	4	｀	エ	ト	ヤ	←	日
5	5	LINE CANCEL			5	E	U	e	u	Ω	5	．	オ	ナ	ユ	↓	千
6	6	LINE END		&	6	F	V	f	v	■	6	ｦ	カ	ニ	ヨ	→	万
7	7	BEL		’	7	G	W	g	w	■	7	ｧ	キ	ヌ	ラ	π	£
8	8	BS		(	8	H	X	h	x	α	8	ｨ	ク	ネ	リ	♠	¢
9	9	CAPS L-U		)	9	I	Y	i	y	β	9	ｩ	ケ	ノ	ル	♥	±
10	A	LF		*	:	J	Z	j	z	γ	+	ｪ	コ	ハ	レ	♦	∓
11	B	HOME		+	;	K	[	k	{	ε	-	ｫ	サ	ヒ	ロ	♣	°
12	C	CLS	➡	,	<	L	¥	l	|	θ	n	ｬ	シ	フ	ワ	□	
13	D	CR	⬅	-	=	M	]	m	}	μ	x	ｭ	ス	ヘ	ン	○	
14	E	SHIFT SET	⬆	.	>		^	n	~	σ	-1	ｮ	セ	ホ	¨	Δ	
15	F	CAPS U-L	⬇	/	?		_	o		φ	÷	ｯ	ソ	マ	｡	\	

III. Internal Information

Table of Contents

This information is a summary of “FX-870P analysis details” (Kota-chan) published in the July 1991 issue of PJ .
Information related to machine language in the “FX-870P Analysis Details” is currently available at http://pb-prog.sakura.ne.jp/fx-870p.html .

3-1 Machine Language Related

Memory Map

FX-870P and VX-4 have 4 memory banks (64KB × 4). The overall memory map is shown in FIG. The features are as follows.

1. Compared to the PB-1000 (see A-1.) With the same CPU as the FX-870P, it is an orderly layout with BANK0 to 3 assigned to ROM, RAM, ROM, and I / O, respectively . There are advantages such as easy to program.
2. All system programs (BASIC, C, CASL) are in the BANK0 ROM.
3. In the BANK1 RAM, 4KB from 0000H to 0FFFH is not used by the system at all, and the VX-4 has no memory.
4. The BANK2 ROM stores overseas characters and fonts, various messages, and training board programs.
5. BANK3 is used for I / O. Addresses 0 to 7 of the 30-pin connector are assigned to 8000H to 8007H. By setting DEFSEG = H1000 , the PEEK and POKE argument addresses can be input / output from 0 to 7.

Of these, the first unused 4KB of BANK1 is suitable for storing machine language programs.

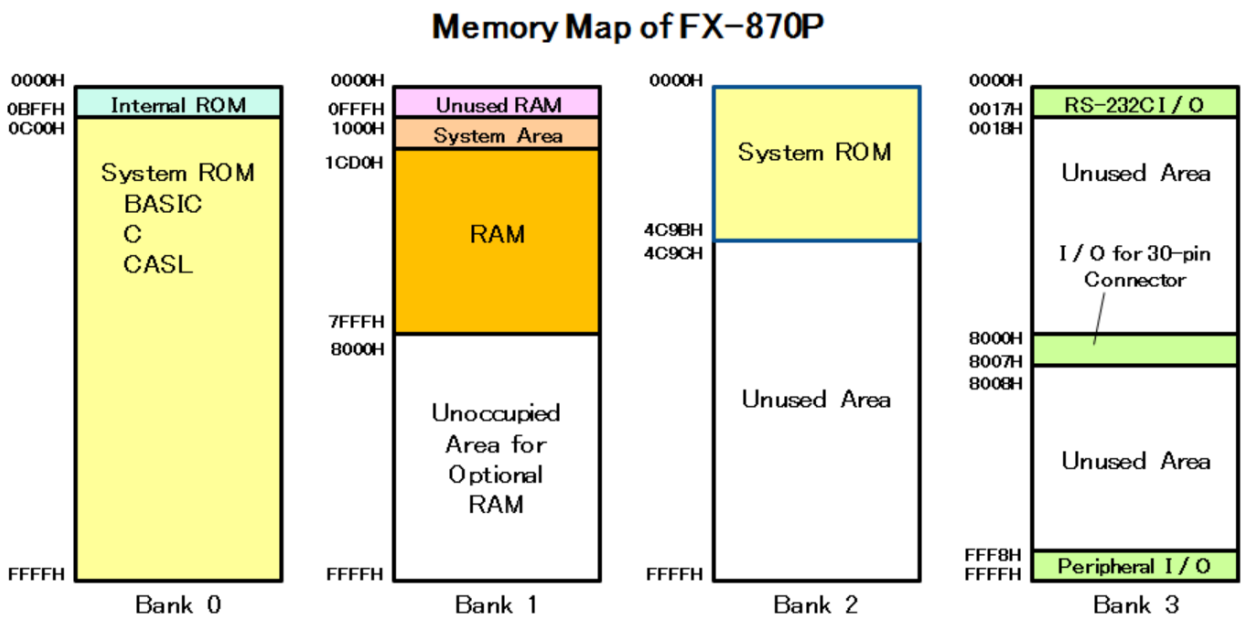


Table 1. BANK 3 ROM Details

Start address (Hexadecimal)	ROM Contents
0000H	Standard character font
0540H	Character font for overseas
0A80H	BASIC error message table
0EA8H	unused
13F7H	Data area for F.COM, CASL, FX, system message
2739H	unused
27D4H	Data area for ROM check program
2AD9H	unused
2E1EH	BASIC program for communication with 3 pins, etc.
38C4H	C language command table
3BCBH	unused
4248H	C error message table
47CFH	unknown
4C9CH	unused

System Area (BASIC)

BANK1 0000H to 0FFFH is not used. 1000H to 1CD0H are used as system areas as shown in Table 2.

- Label names that are basically the same as PB-1000 have the same name as the "PB-1000 Technical Handbook". Other than that, Kota-chan was named.
- For bit specification, the left side of / is 1 and the right side is 0. In the case of true / false, 1 is true and 0 is false.
- Where "Unknown" is written, the part that could not be confirmed

Table 2. List of System Work Areas

Data classification	LABEL	ADDRESS (Hexadecimal)	BYTE number	Explanation
data	**INTOP**	1000	256	Intermediate code buffer
	LCDST	1100	1	7bit: NONE 6bit: NONE 5bit: Inverted display (ON / OFF) 4bit: Cursor bar ON / OFF 3bit: Cursor movement range specification 2bit: Virtual screen / Real screen 1bit: Virtual display enable 0bit: KEY input / PRINT
	EDCSR **SCTOP** **TOEDB** **BOEDB** **MOEDB**	1101 1102 1103 1104 1105	1 1 1 1 1	Cursor position Real screen top (upper 3 bits, lower 5 bits 0) Logical row top (upper 3 bits, lower 5 bits 0) Log. row bottom (upper 3 bits, lower 5 bits 1) Logical line top (when INPUT)
	TOARE **BOARE**	1106 1107	1 1	Cursor movement range top Cursor movement range bottom
	EDCNT **DSPMD** **SCROL**	1108 1109 110A	1 1 1	Position of last character entered +1 00H = Normal display / 01H = PF display 80H = 4 line scroll / 60H = 3 line scroll / 40H = 2 line scroll / 20H = 1 line scroll
	ELVAD	110B 110D	2 6	Contrast data (ROM address) unknown
Key data	**KEYMD**	1113	1	6bit: Kana 5bit: NONCAPS
	KYSTA	1114	1	7bit: AC 6bit: OFF 5bit: APO prohibited 4bit: Contrast 3bit: REPEAT enable 2bit: REPEAT ON / OFF

				1bit: 0 0bit: 0
	CHATA	1115	1	For time counting of chattering
	KEYCM **KEYIN**	1116 1117	1 2	KO KI
	KYREP	1119 111A	1 1	Key repeat count time unknown
	KECNT	111B	22	Key buffer 1 byte: 00H pointer reference 2 bytes: buffer pointer 1 byte: 10H buffer length 2 bytes: buffer start address 16 bytes: buffer
		1131	1	unknown
BASIC data 1	**ANGFL** **RNDFL**	1132 1133 1134	1 1 1	Angle mode (0: DEG, 1: RAD, 2: GRA) 0: Round after computation (MODE10), 1: No rounding after computation (MODE11) ... (Note 1) unknown
Screen data	**CSRDT** **EDTOP** **LEDTP**	1135 113B 123C	6 257 768	Data buffer for blinking cursor Input buffer Display dot buffer
	CGRAM	153C	24	Display dot pattern for character code FCH to FFH
I / O data	**RS1**	1554	1	7,6,5bit: (1 1 1) ... 75 baud (unconfirmed) (1 1 0) ... 150 baud (1 0 1) ... 300 baud (1 0 0) ... 600 baud (0 1 1) ... 1,200 baud (0 1 0) ... 2,400 baud (0 0 1) ... 4,800 baud (0 0 0) ... 9,600 baud (use confirmation) 4bit: Stop bit 1/2 3bit: Data length (bit) 7/8 2bit: Parity ON / OFF 1bit: Parity Odd / Even 0bit: MT / RS-232C
	RS2	1555	1	1bit: For input SO 0bit: For output XOFF
	RS3	1556	1	7bit: NONE 6bit: For input XOFF 5bit: SO for output 4bit: CD control specification 3bit: DSR control designation

				2bit: CTS control designation 1bit: XON / XOFF specification 0bit: SI / SO control designation
	RS4	1557	1	4bit: Framing 3bit: parity 2bit: Overrun 1bit: not Ready 0bit: Buffer
	INTCK	1558	1	01H・・・Data reception
	RXCNT	1559	258	RS-232C, MT reception buffer 1 byte: Number of receive buffers 1byte: Input pointer 256byte: Receive buffer
		165B	1	unknown
	ACJMP	165C	2	Jump destination address at BREAK
BASIC data 2	**WORK1**	165E	28	WORK buffer
		167A	4	unused(?)
	VAR1 **VAR2** **VAR3** **VAR4**	167E 167F 1680 1681	1 1 1 2	Variable work Variable work Variable work Variable work
	PASS	1683	8	Password storage area (entered as XOR255)
	CASPN **CPN**	168B 168C	1 1	CASL program number C program number
		168D 168E	1 41	unknown unknown
	FCOMD **FCOM1** **FCOM2**	1687 1688 1689	1 1 1	F.COM device, (000000AB) B. AB = 00・・・RS-232C AB = 01・・・DISK AB = 10・・・MT F.COM P / F F.COM number
		16BA	5	unknown
	OPTCD **SEGAD**	16BF 16C0	1 2	Option code Segment value
		16C2	1	unknown
	SETDA	16C3	1	With SET instruction data (00AB ####) B, E ... A = 1 / F ... B = 1 / #### = Number of BCD digits
	MODE1	16C4	1	Impossible to confirm
	MODE2	16C5	1	In FX-870P / VX-4, it always seems to be 0.

	MODE3	16C6	1	01H: BASIC running (RUN) 02H: BASIC stopped (STOP) 00H: Other
	NOWFL **NOWLN** **EXEDE**	16C7 16C9 16CB 16CD	2 2 2 2	Same as below The address of the file currently in use Currently executing line number The address of the instruction currently being executed
		16CF	12	unknown
	DATPA **CONTA** **ERRFL** **EJPDE** **ERRLN** **ERRDE** **ERRN** **EJPFG** **TRAFG** **INPER** **STAT** **OUTDV** **IOSTS** **PRSW** **PTABC** **RSFG** **RND** **ANSAD** **FDDBF**	16DB 16DD 16DF 16E1 16E3 16E5 16E7 16E8 16E9 16EA 16F1 1739 173A 173B 173C 173D 173E 1740 1749 174A 1753 1770 1790 1793	2 2 2 2 2 2 1 1 1 2 72 1 1 1 1 1 2 9 1 9 29 35 3 258	DATA statement pointer Pointer to resume execution at CONT ON ERROR Valid file DIR address ON ERROR Jump destination pointer Error line number Error statement statement address Error number 00H: Normal processing / 01H: ON ERROR processing 00H: TROFF / 01H: TRON INPUT Error return address Data for STAT Output device (00: display, 02: printer, 04: FCB) IBIT ON reception open / OBIT ON transmission open PRT ON / OFF (1/0) Number of printer output characters unknown RS-232C default value (DATA of RS1, RS3) Random number data unknown ANS data unknown FILE work (?) unknown FDD buffer
Main data	**IOBF** **SSTOP** **SBOT** **FORSK** **GOSSK** **TONDT** **DTTB** **TOSDT** **PTSDT** **P0STT** **P1STT** **P2STT** **P3STT**	1895 1897 1899 189B 189D 189F 18A1 18A3 18A5 18A7 18A9 18AB 18AD	2 2 2 2 2 2 2 2 2 2 2 2 2	Start address of I / O buffer First address of character calculation work Stack free area start address FOR stack pointer GOSUB stack pointer Numeric conversion data Variable table Character variable data Character data free area P0 first address P1 start address P2 start address P3 start address

	P4STT	18AF	2	P4 start address
	P5STT	18B1	2	P5 start address
	P6STT	18B3	2	P6 start address
	P7STT	18B5	2	P7 start address
	P8STT	18B7	2	P8 start address
	P9STT	18B9	2	P9 first address
	F0STT	18BB	2	F0 start address
	F1STT	18BD	2	F1 start address
	F2STT	18BF	2	F2 start address
	F3STT	18C1	2	F3 start address
	F4STT	18C3	2	F4 start address
	F5STT	18C5	2	F5 start address
	F6STT	18C7	2	F6 start address
	F7STT	18C9	2	F7 start address
	F8STT	18CB	2	F8 start address
	F9STT	18CD	2	F9 start address
	MEMEN	18CF	2	File / Free area start address
	DIREN	18D1	2	RAM end address
	CALC	18D3	258	Calc buffer
	IOBUF	19D5	258	I / O buffer for SAVE / LOAD
stack	**SSPBT**	1AD7	256	
	SSPTP	1BD7	0	System stack area
	USPBT	1BD7	249	
	USPTP	1CD0	0	User stick area

* (Note 1) Although it was written as MODED in “FX-870P Analysis Details”, it was found to be data that determines the validity / invalidity of rounding after four arithmetic operations. The name of the equivalent system area data of the described FX-890P / Z-1 ROUNDFLG is now referred to as RNDFL in accordance with the nomenclature of FX-870P (PB-1000).

ROM Routine

 Table 3 shows the available BANK1 ROM routines that have been confirmed so far. The names of the same routines as in the "PB-1000 Technical Handbook" remain the same. How to call a ROM routine from a machine language program is explained in 1-5.

Table 3. FX-870P ROM Routine List

Label Name	Address	Function
NEXTC	0049H (73)	The search is started from the address specified by IZ, and if a code other than space (20H) is found, that code is placed in $ 0. [input] IZ: Search start address [output] IZ: Address where the code at $ 0 exists $ 0: first non-space code
ENDSC	003CH (60)	When NEXTC is executed and the value of $ 0 is 0, 1, 2, the flag register carry is turned ON (1) [input] IZ: Search start address [output] IZ: Address where the code at $ 0 exists $ 0: first non-space code FLG: Carry flag = 1 @ $ 0 = 0,1,2
OKNMI	002BH (43)	When the value of $ 0 is a number (ASCII code 30H to 39H), the flag register carry is turned ON (1). [input] $ 0: code to check [output] $ 0: code FLG: Carry flag = 1 @ $ 0 = 30H-39H
OKAMI	00ABH (171)	When the value of $ 0 is an alphabetic capital letter (A to Z), the flag register carry is turned ON (1). [input] $ 0: code to check [output] $ 0: code FLG: Carry flag = 1 @ $ 0 = "A"-"Z"
FC07	00E9H (233)	The search starts from the address specified by IZ, and if a code other than space (20H) is found, $ 1 (7 is stored) and $ 2 are compared against the 2 bytes of the code at the next address . As a result, if they match, the zero flag is turned ON (1). [input] IZ: Search start address $ 2: Second code [output] $ 1: 07H $ 2: Second code FLG: Zero flag 1 @ match / 0 @ mismatch IZ: Address of the first code found + 2 @ Z = 1 / unchanged @ Z = 0 Register $ 0 whose contents are destroyed ☆ Routines of the same series FC06 00EBH (235) $ 1 = 06H FC05 00EDH (237) $ 1 = 05H FC04 00EFH (239) $ 1 = 04H

		The rest is exactly the same as FC07. This routine is used to determine BASIC instructions.
SCF2F	00BBH (187)	After executing NEXTC , if the value of $ 0 matches $ 1 (= 2FH), the zero flag is turned ON (1). [input] IZ: Search start address [output] $ 0: first non-space code $ 1: 2FH FLG: Zero Flag 1 @ ($ 0) = ($ 1) / 0 @ ($ 0) <> ($ 1) IZ: Address of the first code found + 1 @ Z = 1 / unchanged @ Z = 0 ☆ Routines of the same series SCF3A 00BDH (189) $ 1 = 3AH SCF22 00BFH (191) $ 1 = 22H SCF40 00C1H (193) $ 1 = 40H SCF2C 00C3H (195) $ 1 = 2CH SCF28 00C5H (197) $ 1 = 28H SCF29 00C7H (199) $ 1 = 29H SCF2D 00C9H (201) $ 1 = 2DH SCF3B 00CBH (203) $ 1 = 3BH SCF23 00CDH (205) $ 1 = 23H SCF2E 00CFH (207) $ 1 = 2EH SCFXX 00D1H (209) $ 1 = value entered by myself immediately before The rest is exactly the same as SCF27.
SCE3B	00D7H (215)	After executing NEXTC , if the value of $ 0 matches $ 1 (= 3BH), the zero flag is turned ON (1). If it doesn't match, it becomes SNerr. [input] IZ: Search start address [output] FLG: Zero Flag 1 @ ($ 0) = ($ 1) / 0 @ ($ 0) <> ($ 1) When Z = 1 $ 0: first non-space code $ 1: 3BH (";") IZ: First code address +1 When Z = 0 SNerr ☆ Routines of the same series SCE24 00D9H (217) $ 1 = 24H SCE2C 00DBH (219) $ 1 = 2CH SCE2D 00DDH (221) $ 1 = 2DH SCE29 00DFH (223) $ 1 = 29H SCE28 00E1H (225) $ 1 = 28H SCF3D 00E3H (227) $ 1 = 3DH SCEXX 00E5H (229) $ 1 = value entered by myself just before The others are exactly the same as SCE3B.
TCAPS	00B6H (182)	Convert lowercase alphabetic codes in $ 0 to uppercase alphabetic codes. No conversion is performed for non-alphabetic characters. [input] $ 0: lowercase alphabetic code [output] $ 0: Alphabet capital letter code
CHEXI	009DH (157)	If the code in $ 0 is characters 0 to 9, A to F, a to f (30H-3H, 41H-46H, 61H-66H), $ 0 is converted to a numerical value (00H-0FH) as a

		hexadecimal character . [input] $ 0: Hexadecimal character code [output] $ 0: Hexadecimal conversion value (00H-0FH)
CLEME	014CH (332)	Clears the number of bytes specified by $ 2 and $ 3 to 0 from the specified saler address by $ 0 and $ 1. If $ 2 and $ 3 are 0, do not execute. [input] $ 0, $ 1: Start address to clear $ 2, $ 3: number of bytes to clear [output] IZ: Cleared address + 1 $ 5 to $ 13: All 0 Registers whose contents are destroyed $ 0 to $ 2, $ 14
CLEDB	9338H (37688)	Clear the contents of EDTOP (113BH-123BH) and LEDTP (123CH-153BH) of BANK1 to 0 and set each pointer to CLS. [output] IX: Contents of EDCSR (1101H) Contents of IZ: MOEDB (1105H) Registers whose contents are destroyed $ 0 to $ 14
DOTDS	930FH (37647)	Displays full screen according to the contents of DSPMD (1109H). Transfer the contents of 3 or 4 lines from LEDTP (123CH-153BH) + SCTOP (1102H) x 6 to the LCD. [input] Depending on the contents of DSPMD (1109H), it is determined whether it is 3 or 4 lines. [output] None Registers whose contents are destroyed $ 0 to $ 15, IX
BRSTR	297AH (10618)	Put the contents of $ 2 and $ 3 into ACJMP (165CH, 165DH). [input] $ 2, $ 3: data [output] None Register IX whose contents are destroyed
CRTKY	23C8H (9160)	Contrast key execution KEY sample flow. **The BREAK** key jumps to the address specified by ACJMP (165CH, 165DH). [input] None [output] $ 0: Key code (see Table 4) is entered. Registers whose contents are destroyed $ 1 to $ 11, IX, IZ
KYCHK	506EH (20590)	Check the **OFF** , **BREAK** , and **STOP** keys. [input] None [output] FLG: Zero flag = 1 @ **STOP** key Registers whose contents are destroyed $ 0 to $ 4
BKCK	29C5H (10693)	Check **OFF** key and sample **BREAK** key. [input] None [output] None Registers whose contents are destroyed $ 0 to $ 4
OUTCR	2AE8H (10984)	Outputs 0DH and 0AH (CR, LF) to the device. [input] The device depends on the contents of OUTDV (1739H). [output] None Registers whose contents are destroyed $ 0 to $ 13, $ 16, IX

PROUT	89A9H (35241)	Output $ 16 contents to the printer. If it is not connected to the printer, it will be NRerror. [input] $ 16: Data output to the printer [output] None Registers whose contents are destroyed $ 0 to $ 6, IX
DTBIN	1EE6H (7910)	The ASCII code existing at the address specified by IZ is converted to a numerical value as a decimal number. • If the conversion result exceeds 65536, an OV error will occur. • Returns 0 if there are no numeric characters (30H-39H). ・ If a code other than numeric characters (30H-39H) exists, it will end immediately. At this time, skip the space. [input] IZ: Start address of the string to be converted to a number [output] IZ: Address where data other than "0"-"9" (30H-39H) exists $ 17, $ 18: Conversion result value Registers whose contents are destroyed $ 0 to $ 3, $ 16
BINMZ	0EFDH (3837)	Real type number x in $ 10 to $ 18 is -32769 <x<65536 [input] $ 10 to $ 18: Real number [output] $ 15, $ 16: integer type number Registers whose contents are destroyed $ 10 to $ 14, $ 17 to $ 18, IX</x<65536
BIN01	0EC6H (3782)	If the real type number x in $ 10 to $ 18 is 0 <= x <256, it is converted to an integer type number. If it is out of range, a BS error occurs. [input] $ 10 to $ 18: Real number [output] $ 15, $ 16: integer type number Registers whose contents are destroyed $ 10 to $ 14, $ 17 to $ 18, IX
BIN11	0ECEH (3790)	If the real type number x in $ 10 to $ 18 is 1 <= x <256, it is converted to an integer type number. If it is out of range, a BS error occurs. [input] $ 10 to $ 18: Real number [output] $ 15, $ 16: integer type number Registers whose contents are destroyed $ 10 to $ 14, $ 17 to $ 18, IX
BIN02	0EE2H (3810)	If the real type number x in $ 10 to $ 18 is 0 <= x <65536, it is converted to an integer type number. If it is out of range, a BS error occurs. [input] $ 10 to $ 18: Real number [output] $ 15, $ 16: integer type number Registers whose contents are destroyed $ 10 to $ 14, $ 17 to $ 18, IX
BIN12	0EE8H (3816)	If the real type number x in $ 10 to $ 18 is 1 <= x <65536, it is converted to an integer type number. If it is out of range, a BS error occurs. [input] $ 10 to $ 18: Real number [output] $ 15, $ 16: integer type number Registers whose contents are destroyed $ 10 to $ 14, $ 17 to $ 18, IX
SIKI	1088H (4232)	Execute an expression (which may be a character expression) and obtain the result. • When the result is a numeric value, it is stored as a real number value in $ 10 to $ 18. • When the result is a character string, it is stored in the free area of RAM, the start address of the character string is stored in $ 15 and $ 16, and the character length is stored in $ 17. [input] IZ: RAM start address where the expression is stored. Reserved words (functions, etc.) in expressions must be converted to internal code.

		[output] IZ: End of expression + 1 address ・ When the result is numeric $ 10 to $ 18: Real number FLG: Turn carry (OFF). -When the result is a string $ 15, $ 16: string start address $ 17: string length FLG: Turns carry on (1).
EXPRW	112FH (4399)	Execute the mathematical formula and obtain the result. [input] IZ: RAM start address where the expression is stored. Reserved words (functions, etc.) in expressions must be converted to internal code. [output] IZ: End of expression + 1 address $ 10 to $ 18: Real number
NISIN	0AFAH (2810)	The value of $ 17 is the BCD number. Convert to binary. [input] $ 17: BCD number [output] $ 17: Binary conversion value Register $ 19 whose contents are destroyed
SIKI2	11D2H (4562)	Execute a character expression and obtain the result. [input] IZ: RAM start address where the expression is stored. Reserved words (functions, etc.) in expressions must be converted to internal code. [output] IZ: End of expression + 1 address $ 15, $ 16: string start address $ 17: string length
INKEY	191DH (6429)	INKEY $ subroutine. [input] None [output] $ 15, $ 16: Address where keyed data (see Table 5) is stored $ 17: 0 @ No key input / 1 @ Key input Registers whose contents are destroyed $ 0 to $ 5, $ 18, IX
?? Err	following	BASIC error occurred. After execution, waits for input in BASIC or CAL mode. [input] None [output] None The error name and its address are as follows. LBERR・・・・2B5EH (11102) (Note 1, 2) OMERR・・・2B6DH (11117) SNERR・・・2B70H (11120) STERR・・・2B74H (11124) TCERR・・・2B78H (11128) BVERR・・・2B7CH (11132) NRERR・・・2B80H (11136) RWERR・・・・2B84H (11140) BFERR・・・2B88H (11144) BNERR・・・2B8CH (11148) NFERR・・・2B90H (11152) FLERR・・・・2B94H (11156) OVERR・・・2B98H (11160) MAERR・・・2B9CH (11164)

		DDERR・・・2BA0H (11168) BSERR・・・2BA4H (11172) FCERR・・・・2BA8H (11176) ULERR・・・2BACH (11180) TMERR・・・2BB0H (11184) REERR・・・2BB4H (11188) PRERR・・・2BB8H (11192) DAERR ... 2BBCH (11196) FOERR・・・2BC0H (11200) NXERR 2 ... 4BC4H (11204) GSERR・・・2BC8H (11208) FMERR・・・2BCFH (11215) FDERR・・・2BD3H (11219) OPERR・・・2BD7H (11223) AMERR・・・2BDBH (11227) FRERR・・・2BDFH (11231) POERR・・・・2BE3H (11235) DFERR・・・2BE7H (11235)
BEEP	33B3H (13235)	BASIC BEEP sound is generated. [input] None [output] None Registers whose contents are destroyed $ 0 to $ 3
ENLST	508BH (20619)	The BASIC program stored in internal code is converted into ASCII code for one line from the address specified by IZ and stored in INTOP (1000H-10FFH). [input] IZ: Address where the line of the BASIC program to convert starts [output] IZ: Start address of next line or program end (0) Registers whose contents are destroyed $ 0 to $ 16, IX
RSOPN	84ECH (34028)	Open RS-232C hardware. ・ Set baud rate ・ Turn on DTR and RTS. [input] $ 00: Open mode = 01H @ Transmission / 02H @ Reception / 03H @ Transmission / reception $ 11: Value entered in RS1 (1554H) $ 13: Value entered in RS3 (1556H) If you do not set RS1 to RS4 of the work area before calling this routine, it will not operate normally. [output] None Registers whose contents are destroyed $ 0 to $ 6, IX
RSCLO	8563H (34147)	Performs RS-232C hardware close. [input] None [output] None Registers whose contents are destroyed $ 0 to $ 3, IX
RSGET	8590H (34192)	Extract one character from the RS-232C receive buffer. When the buffer is empty, wait until data is received. • If XON / XOFF is specified and XOFF is selected, one character is first

		extracted from the buffer. When the remaining characters are 32 characters or less, XON is transmitted. • When an error is detected, jump to each error. [input] None [output] $ 0: Receive data Registers whose contents are destroyed $ 1 to $ 4, IX
PRTRS	85FBH (34299)	Send $ 16 data via RS-232C. • If XON / XOFF is specified and XOFF is set, wait until it becomes XON. • If SI / SO is specified, control it. [input] $ 16: Transmission data [output] None Registers whose contents are destroyed $ 0 to $ 4, IX
NTX	865CH (34396)	Send the contents of $ 0 via RS-232C. • Sends the contents of $ 0 regardless of the XON / XOFF and SI / SO specifications. [input] $ 0: Transmission data Upper 2 bits of UA register = 11 [output] None
DOTMK	977FH (38783)	Create a dot pattern for the character in EDTOP (113BH-123BH) specified by $ 10, $ 11 in LEDTP (123CH-153BH). [input] $ 10: Start cursor address $ 11: End cursor address [output] None Registers whose contents are destroyed $ 0 to $ 11, IX, IZ

*
(Note 1) Ayaka Toji, PJ February 1991, p.106, `` ROM analysis of FX-870P ".
(Note 2) Errors not listed in the error message list in CASIO “VX-4 Operation Text”, p.93. Short for "Low Battery"?

Table 4. Key Code Table by CRTKY (23C8H)

* **E on** A0H is the π button on the Numeric Keypad
BRK , **STOP** , **OFF** , **ALL RESET** , **CASL** , **FX** , **C** , **MODE** ,
CONTRAST ↑ ↓ Keys are executed. **CAPS** , **Kana** changes State.

		Upper 4 Bits															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Under Place 4 Bit	**0**		F.TOP	SPC	0	@	P	'	p	PRINT	$^3\sqrt{}$	E		タ	ミ	ENG	P0
	1	F.END	DEL	!	1	A	Q	a	q	SYSTE M	√	X^2	ア	チ	ム	TAB	P1
	2	L.TOP	INS	"	2	B	R	b	r	CLEAR	hyp	X^3	イ	ツ	メ	MR	P2
	3			#	3	C	S	c	s	CONT	SET		ウ	テ	モ	Min	P3
	4			$	4	D	T	d	t	RENUM	FACT		エ	ト	ヤ	M +	P4
	5	L.CAN		%	5	E	U	e	u	RUN	RAN #		オ	ナ	ユ	M-	P5
	6	L.END		&	6	F	V	f	v	EDIT	π	ヲ	カ	ニ	ヨ	IN	P6
	7			'	7	G	W	g	w	log	nPr	ァ	キ	ヌ	ラ	OUT	P7
	8	BS		(	8	H	X	h	x	ln	nCr	ィ	ク	ネ	リ	CALC	P8
	9			)	9	I	Y	i	y	e^x	HEX $	ゥ	ケ	ノ	ル	ANS	P9
	A			*	:	J	Z	j	z	sin	DEGR	ェ	コ	ハ	レ		
	B	HOME		+	;	K	[	k	{	cos	DMS	ォ	サ	ヒ	ロ		
	C	CLS	→	,	<	L	¥	l	|	tan	POL (	ャ	シ	フ	ワ		
	D	EXE	←	-	=	M	]	m	}	$\sin^{-1}$	REC (	ュ	ス	ヘ	ン		
	E		↑	.	>	N	^	n	~	$\cos^{-1}$	& H	ョ	セ	ホ	゛	MEM O	
	F		↓	/	?	O	_	o		$\tan^{-1}$	10^x	ッ	ソ	マ	゜	LINE	

Table 5. Key Code Table by INKEY (191DH)

* When **BRK** is executed, processing is transferred to the address indicated by ACJMP .

		Upper 4 Bits															
		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
Under Place 4 Bit	**0**			SPC	0		P	‘	p							ENG	
	1				1	A	Q	a	q								
	2		INS		2	B	R	b	r							MR	
	3		OFF		3	C	S	c	s								
	4				4	D	T	d	t							M+	
	5				5	E	U	e	u								
	6				6	F	V	f	v							IN	
	7				7	G	W	g	w							OUT	
	8	BS		(	8	H	X	h	x							CAL C	
	9			)	9	I	Y	i	y							ANS	
	A			*		J	Z	j	z								ALL RESET
	B			+		K		k									MODE
	C	CLS	→	,		L		l									
	D	EXE	←	-	=	M		m									
	E		↑	.		N	^	n								MEMO	
	F		↓	/		O		o								LINE	

1-4. Key matrix Table 6 shows the key matrix of FX-870P. To obtain a key, first assign the specified output value (7 if "6") to the IA register, and if the key is pressed, the corresponding bit in the KY register will be 1 (If it is "6", the 0th bit becomes 1. In other words, KY = 0001H).
Listing 1 shows a sample program that can read **2** , **4** , **6** , **8** and **SPC** simultaneously. If you call this program, $ 0 returns the result as follows.

7 6 5 4 3 2 1 0 (bit)
0 0 0 SPC 8 2 4 6

The bit where the key was pressed becomes 1.

Table 6. FX-870P Key Matrix Table

* **E** is the π button on the numeric keypad

		IA Register Key Output Specification Value								
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
KY Les The T of Out Power Bi Tsu G Place Place	**0**		Fx	ln	hyp	(	9	6	3	**E** *
	1		CASL	log	MR	M +	8	5	2	.
	2		SHIFT	7	ENG	4	ANS	1	SPC	0 (zero)
	3		→	INS	O	P	K	L	,	=
	4		↓	←	U	I	H	J	N	M
	5		CALC	↑	T	Y	F	G	V	B
	6		IN	OUT	E	R	S	D	X	C
	7	BRK	OFF	MEMO	Q	W	RESET	A	CAPS	Z
	14		X^2	MODE	cos	tan	CLS	/	-	EXE
	15		DEGR	√	sin	)	^	BS	*	+

Listing 1. Simultaneous Key Input Subroutine

ADRS	Code	Label	Mnemonic		Comment
xx00	02 60 1F	KEY:	LD	$ 0, $ 31	; Clear result input register ($ 31 = 0)
xx03	57 00 08		PST	IA, & H08	; SPACE check
xx06	42 01 04		LD	$ 1, & H04	
xx09	77 2D xx		CAL	SCAN	
xx0C	57 00 06		PST	IA, & H06	; 8 check
xx0F	42 01 02		LD	$ 1, & H02	
xx12	77 2D xx		CAL	SCAN	
xx15	57 00 08		PST	IA, & H08	; 2 check
xx18	42 01 02		LD	$ 1, & H02	
xx1B	77 2D xx		CAL	SCAN	
xx1E	57 00 05		PST	IA, & H05	; 4 check
xx21	42 01 04		LD	$ 1, & H04	
xx24	77 2D xx		CAL	SCAN	
xx27	57 00 07		PST	IA, & H07	; 6 check
xx2A	42 01 01		LD	$ 1, & H01	
xx2D	18 60	SCAN:	BIU	$ 0	; Bit up $ 0
xx2F	9F 22		GRE	KY, $ 2	; Matrix key scan
xx31	0C 62 01		AN	$ 2, $ 1	; Clear key bits to check
xx34	F0		RTN	Z	; Return if no key to check is pressed
xx35	0E 60 1E		OR	$ 0, $ 30	; Set the least significant bit ($ 30 = 1)
xx38	F7		RTN		

Note: Although the subroutine SCAN is as follows in the original, it is NG because the result is strange.

ADRS	Code	Label	Mnemonic		Comment
xx2D	18 60	SCAN:	BIU	$ 0	; Matrix can
xx2F	9F 22		GRE	KY, $ 2	; Dummy input
xx31	9F 24		GRE	KY, $ 4	; This input
xx33	81 62 04		SBCW	$ 2, $ 4	; Key check
xx36	B4 8A		JR	NZ, SCAN	; Return if not pressed. → When you **return, the result is strange because $ 0 is bit-up extra!**
xx38	0C 62 01		AN	$ 2, $ 1	; Clear key bits to check
xx3B	F0		RTN	Z	; Return if no key to check is pressed
xx3C	0E 60 1E		OR	$ 0, $ 30	; Set the least significant bit ($ 30 = 1)
xx3F	F7		RTN		

1-5. Notes on creating Machine Language Programs

The FX-870P / VX-4 uses an 8-bit CPU called Hitachi's HD61700. This CPU has the following registers (Figure 2). For details, refer to “ 2-2. Register Configuration ” in “ HD61700 Cross Assembler ”.

- Internal register
  - $ 0 to $ 31 Main register
  - IX, IY, IZ Index register
  - SSP, USP Stack pointer
  - PC Program counter
  - SX, SY, SZ Specific index register
- Flag register (F)
  - Z Zero flag
  - C Carry flag
  - LZ Lower digit flag
  - UZ Upper digit flag
  - SW Power switch state flag
  - APO Auto Power Off State Flag
- Status register
  - IE Interrupt enable register
  - IA Interrupt selection & KEY output register
  - IB Interrupt control and memory bank range specification register
  - UA Upper address specification register
  - PE Port status specification register
  - PD Port data register
  - TM Timer data register
  - KY Key input register

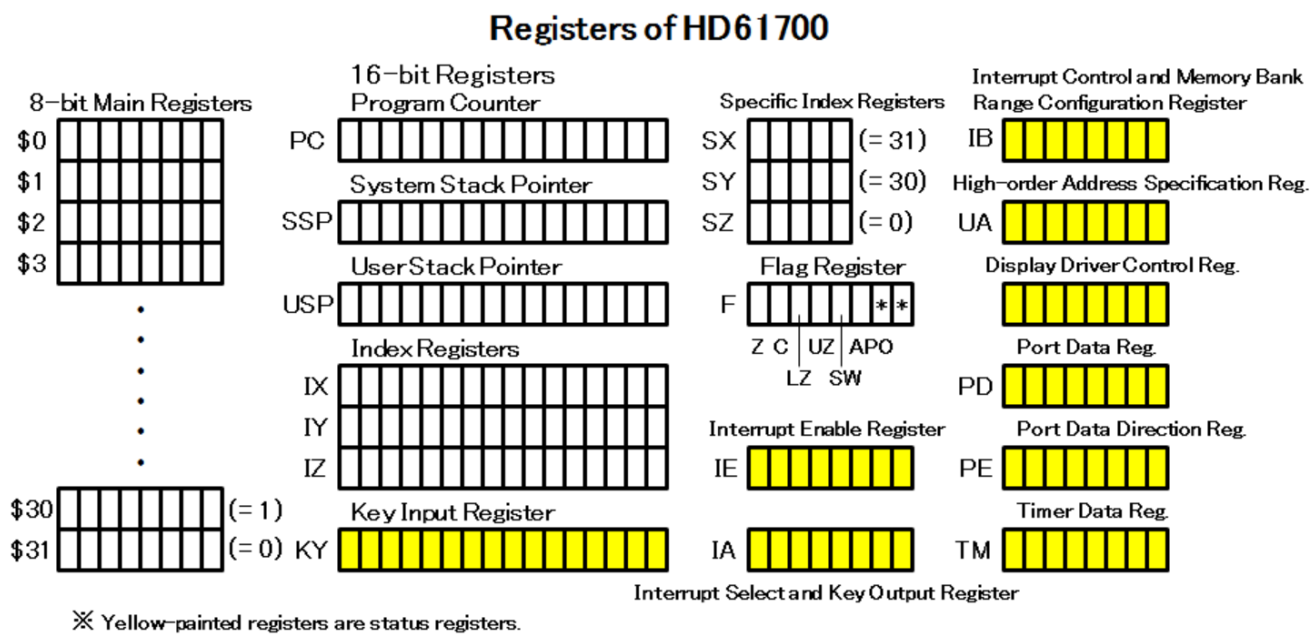


Figure 2. HD61700 register configuration

Here, SX, SY, SZ, IB, and TM were unknown in “FX-870P Analysis Details” . The user can freely use $ 0 to $ 29, index registers IX, IY, IZ and flag register F, and there are restrictions on the use of other registers. In particular, Casio's pocket computer is fixed at $ 30 and $ 31, SX, SY, and SZ are fixed at 31, 30, and 0, respectively, and can operate at high speed when $ 31, $ 30, and $ 0 are specified as the

second operand, respectively. The ROM is coded so that **Therefore, be careful not to change the contents of $ 30 (= 1), $ 31 (= 0), SX (= 31), SY (= 30), SZ (= 0).** (note)

FX-870P and VX-4 can call their own machine language program with BASIC hidden instruction **MODE110** (address) , but it is not officially supported. Therefore, unlike PB-1000 and FX-890P / Z-1, FX-870P and VX-4 BASIC cannot secure the machine language area with the CLEAR instruction, so secure the machine language area as follows. There is a need to.

- Use less frequently used areas such as CALC (calc buffer), IOBUF (SAVE / LOAD I / O buffer), and CGRAM (user-defined character area) in the system area. **However, a large free area cannot be secured, and there is always a risk of machine language data being destroyed.**
- The first 4KB of RAM area 0000 to 0FFFH is unused on the system side, so it can be used for machine language with a certain size.
  However, the unmodified VX-4, which is 32KB and not equivalent to FX-870P, cannot be used even if the additional memory RP-33 is 40KB.
- Ao's extended CLEAR instruction can secure the machine language area for the number of bytes specified from 1CD0H (from 6CD0H for VX-3 extended CLEAR). However, because the extended CLEAR machine language routine is placed in the CALC (calc buffer) in the system area , **storing the formula with more than 32 characters with the IN key limits the extended CLEAR.**
- If CLEAR-ZERO is used, the above extended CLEAR is relocated to addresses 0 to 123, the same operation as the above extended CLEAR is possible, and the user program can be resident unless the contents of addresses 0 to 123 are destroyed. Become. However, as mentioned above **, CLEAR-ZERO cannot be used with an unmodified VX-4 that is not 32KB.**

In this way, there are merits and demerits in securing the machine language area of FX-870 and VX-4. Even if a **machine language is secured from 1CD0H** with **extended CLEAR, if a C program is executed in C language mode, the information in the machine language area will be destroyed.** Therefore, when returning from C language and executing machine language, it is necessary to reload machine language again. For the time being, CASL confirmed that the data in the machine language area was not destroyed after executing a simple program, but it is unknown whether it was completely destroyed.

When returning (ending) from a user-written machine language program to BASIC, processing must be transferred from BANK1 with the machine language program to BANK0 with the BASIC ROM. Therefore, bank switching is required at the end of the program, so be sure to add the following code at the end of the machine language program.

Listing 2. Exit code for machine language program

ADRS	Code	Label	Mnemonic	Comment
xxxx	56 60 54		PST UA, & H54	; Switch to bank 0
xxxx	F7		RTN	; RETURN

Similarly, bank calls are required for FX-870P ROM calls from homebrew machine language programs. Listings 3 and 3-2 show machine language samples that make ROM calls. This ROM call is a well-known method for HD61700 ROM calls. First, enter the ROM routine address to be called into $ 17 and $ 18, call your own BSCLL, switch from here to BANK0 and jump. This program uses $ 15 to $ 18, but if you want to use these registers in a BIOS call, you need to change the registers accordingly.

Listing 3. Machine language sample for ROM calls

ADRS	Code	Label	Mnemonic		Comment
0000	D1 11 0F 93		LDW	$ 17, & H930F	; DOTDS (full screen display) address
0004	77 0B 00		CAL	RMCLL	; Execute ROM call
0007	56 60 54		PST	UA, & H54	; Specify to switch PC BANK to 0
000A	F7		RTN		; Back to BASIC
000B	D1 0F 23 53	RMCLL:	LDW	$ 15, & H5323	; ROM call routine
000F	A6 10		PHSW	$ 16	; & H5323 is pushed into system stack
0011	56 60 54		PST	UA, & H54	; Specify to switch PC BANK to 0
0014	DE 11		JP	$ 17	; BIOS call

Listing 3-2. ROM call routine

ADRS	Code	Label	Mnemonic		Comment
000B	D1 0F 23 53	RMCLL:	LDW	$ 15, & H5323	; ROM call routine
000F	A6 10		PHSW	$ 16	; & H5323 is pushed into system stack
0011	56 60 54		PST	UA, & H54	; Specify to switch PC BANK to 0
0014	DE 11		JP	$ 17	; BIOS call

* "JP $ 17" (opcode DEH) has been described as "JP ($ C5)" in "FX-870P Analysis Details" , but Piotr Piatek specified indirect memory address using the main register ($ C5) By finding the jump instruction (opcode DFH), the unnaturalness of the notation can no longer be ignored, and now it has been changed to "JP $ C5".

Ao's HD61 cross assembler supports "JP $ C5" notation from Ver0.34, so it will malfunction when assembling the old notation source.
Therefore, if there is a "JP ($ C5)" mnemonic, the source may be modified, so be careful.

Also, Ao taught me the equivalent of the ROM call routine prepared in the AI-1000 ROM that was introduced in Ref. (5) , so it is shown in Listing 4 (Ref. (16)). This method is characterized by the fact that the registers to be destroyed are fixed at $ 28 and $ 29, but the number of registers used is smaller than in list 3, and the execution time is longer than in list 3.

Listing 4. Using the ROM call routine provided in FX-870P / VX-4 ROM

ADRS	CODE	LABEL	MNEMONIC		comment
0000	D1 1C 0F 93		LDW	$ 28, & H930F	; DOTDS (full screen display) address
0004	77 0B 00		CAL	RMCLL	; Execute ROM call
0007	56 60 54		PST	UA, & H54	; Specify to switch PC BANK to 0
000A	F7		RTN		; Back to BASIC
000B		RMCLL:			; ROM call routine
000B	56 60 54		PST	UA, & H54	; Specify to switch PC BANK to 0
000E	37 21 53		JP	& H5321	;

(note)
Settings for $ 30 and $ 31 in CASIO Pokécons PB-1000, FX-860P, FX-870P, VX-4, etc. , SZ = 0 is assumed to be used as a fixed value, but the values of $ 30 and $ 31 are 1 and 0 as follows.

- The HD61700 does not have increment and decrement instructions, but these are operations that are frequently used by computers, so the benefits of high speed are significant. At this time, if 1 is put in the main register specified by the specific index register rather than adding constant 1, high-speed operation is possible. In fact,
    - AD $ 2, $ SX — Indirect specification of $ 30 (= 1) by specific register. One byte code can be shorter than main register specification.
    - AD $ 2, $ 1 — Here, $ 1 = 1
    - AD $ 2, 1 — Increment by immediate value 1
- Of these operations, only the top is 9 clocks and the rest is 12 clocks, which can be 25% faster.
- HD61700 index registers IX and IZ cannot be used alone, except for the exception of block transfer instructions, and can only be used in the form {IX | IY} ± A (specific index register specification, main register, 8-bit direct value) If you want to perform IX + 0, you can use the register specified by the specific index register to increase the speed by 3 clocks as above. Therefore, it is useful to assign 0 to the main register specified by a specific index register.
- For the above reasons, assigning both 0 and 1 to the main register specified by the specific index register is effective for speeding up, but by setting $ 30 = 1 and $ 31 = 0, the register pair ($ 31, $ 30) The increment / decrement speed can be increased even with 16-bit arithmetic. Also, it is important to assign 0 to the main register.

    LD $ 2, $ SY — Indirect specification of $ 31 (= 0) by specific register. One byte code can be shorter than main register specification.
    - LD $ 2, 0 — Immediate value substitution of 0
    - XR $ 2, $ 2 — Own exclusive OR.
- Of these, only 9 clocks can be transferred by specifying the top specific index register, and the remaining 12 clocks, which can be accelerated by 3 clocks. However, speeding up the word transfer of 0

    LDW $ 0, $ SY — Indirect specification of $ 31 (= 0) by specific register. One byte code can be shorter than main register specification.
- Can only be loaded into ($ 1, $ 0) pairs, generally

    XRW $ 2, $ 2 — Exclusive OR of yourself in the word.

- Seems to be the fastest exclusive OR (likely because I'm not familiar with HD61700 yet).

The HD61700 cross assembler has the optimization option turned on by default, and even if a specific register is not specified by the above-mentioned Casio Pokekon register setting premise, the specific register is automatically specified. Therefore, it is only necessary to remember that $ 30 = 1, $ 30 = 0 and $ 30, $ 31, $ 0 can be accelerated by specifying a specific index register.

3-2 BASIC Related

In “FX-870P Analysis Details” , only the BASIC hidden instructions and the program storage format were explained. Later, Jun Amano's “BB variable storage format of PB-1000 / C” explained the variable storage method in PB-1000. This time, we will investigate the storage method of variables based on this, and also explain what was corrected in the above explanation.

Hidden BASIC Instructions

Two hidden instructions were found.

① **MODE command** grammar is
MODE Argument 1 (argument 2) has different functions depending on the value of argument 1.
Mode10, 11: PJ Although unknown in the FX-870P analysis details of the July 1991 issue, rounding is performed after four arithmetic operations in **MODE10**, and rounding is not performed in **MODE11**.
Mode110: Call the machine language program in BANK1. Argument 2 is an address.
Mode200,201: FD sector READ, WRITE command. Argument 2 is (track, surface, sector), track is 0-79, surface is 0-1, sector is 1-8. It is unknown which is READ.

(2) **CALCJMP instruction This** is the same as pressing the **CALC** key with an instruction without an argument, and executes the formula entered with the **IN** key. However, it can be executed only in CAL mode, and FCerror in BASIC mode.

BASIC Program and (Text) File Storage Format

In the (text) file area file that can store P0 to F9 programs and C and CASL source files, the start addresses where the respective data are stored are stored in P0STT to F9STT of the system area. The end code of the program (BASIC) is 00H and the end code of the file is 1AH, both of which consume at least 1 byte and consume 20 bytes in total. In the VX-4 manual, the user area is the total of 21 bytes subtracted from the file area, and it seems that the last 1 byte of memory is not consumed. The end-of-file code 1AH is well known as the end-of-file (EOF) code used by many operating systems.

In addition, the system automatically performs memory block transfer and changes in P1STT to MEMEN so that unnecessary data does not occur between files. However, P0STT is not changed unless the user makes a CLEAR statement, and the system side does not change it arbitrarily.

BASIC programs are stored in P0 to P9 in the program area, and the BASIC program method is exactly the same as PB-100. The BASIC sample program in Listing 5 is stored as shown in Table 7. Each line consists of the line length (1 byte), line number (2 bytes), space (1 byte), BASIC code (variable length), and line end code (1 byte). The line length is the total number of bytes from the line number to the line end code. If this is 0, it indicates the end of the program. The line number is 2 bytes of little endian. Space is a space between the line number and the BASIC code, and is fixed with & H20. A BASIC code is a character string in which a reserved word is converted to a 2-byte internal code with big endian. There are reserved words that have processing destinations and no processing destinations such as functions, and Tables 8 and 9 show the internal codes. The line end code is fixed at 0.

Listing 5. BASIC Sample Program

```
100 REM Sample
110 'Program
120 CLS
130 PRINT "Hello"
140 END
```

Table 7. Memory Contents of Listing 5

LEN 1byte	LNUM 2bytes	SPC 1byte	Program Statement Variable Length									EOL 1byte
0D 13	64 00 100	20	04 A9 REM		20	53 S	61 a	6D m	70 p	6C l	65 e	00
0D 13	6E 00 110	20	02	20	50 P	72 r	6F o	67 g	72 r	61 a	6D m	00
06 6	78 00 120	20	04 71 CLS									00
0D 13	82 00 130	20	04 A3 PRINT		twenty two "	48 H	65 e	6C l	6C l	6F o	twenty two "	00
06 6	8C 00 140	20	04 87 END									00
00 0												

Table 8. Internal Code with Processing Destination Address

CODE	BASIC Command	Processing Destination
0449H	GOTO	368AH
044AH	GOSUB	3620H
044BH	RETURN	3663H
044CH	RESUME	3ACBH
044DH	RESTORE	42EBH
044EH	WRITE #	5517H
0450H	CONT	35ADH
0452H	SYSTEM	51BAH
0453H	PASS	525CH
0455H	DELETE	3CDDH
0457H	LIST	3D26H
0458H	LLIST	3D21H
0459H	LOAD	4753H
045AH	MERGE	474BH
045CH	RENUM	43DAH
045DH	TRON	3617H
045FH	TROFF	3614H
0460H	VERIFY	474FH
0463H	POKE	3A23H
0469H	CHAIN	4762H
046AH	CLEAR	53A8H
046BH	NEW	4594H
046CH	SAVE	4736H
046DH	RUN	352CH
046EH	ANGLE	3929H
046FH	EDIT	58B8H
0470H	BEEP	43C7H
0471H	CLS	2ADFH
0472H	CLOSE	46B0H
0476H	DEF	397DH
0478H	DEFSEG	3A3AH
047CH	DIM	3A4AH
0480H	DATA	0B9BH
0481H	FOR	36F9H
0482H	NEXT	383BH
0485H	ERASE	3A81H
0486H	ERROR	2BA8H
0487H	END	3520H
048BH	FORMAT	7F0FH
048DH	IF	38BBH
048EH	KILL	7F1EH
048FH	LET	2EA2H
0490H	LINE	3E26H
0491H	LOCATE	39FAH
0496H	NAME	7F35H
0497H	OPEN	45DFH
0499H	OUT	2BA8
049AH	ON	3B71H
049FH	CALCJMP	542CH
04A3H	PRINT	3EF1H
04A4H	LPRINT	3EECH
04A5H	PUT	2BA8H
04A8H	READ	42A0H
04A9H	REM	0B9BH
04ACH	SET	532AH
04ADH	STAT	4322H
04AEH	STOP	3500H
04B0H	MODE	52A2H
04B2H	VAR	3BEBH
04B5H	FILES	7F87H

Table 9. Internal Code without Processing Destination

CODE	BASIC Function
054FH	ERL
0550H	ERR
0551H	CNT
0552H	SUMX
0553H	SUMY
0554H	SUMX2
0555H	SUMY2
0556H	SUMXY
0557H	MEANX
0558H	MEANY
0559H	SDX
055AH	SDY
055BH	SDXN
055CH	SDYN
055DH	LRA
055EH	LRB
055FH	COR
0560H	PI
0561H	DSKF
0563H	CUR
0567H	FACT
0569H	EOX
056AH	EOY
056BH	SIN
056CH	COS
056DH	TAN
056EH	ASN
056FH	ACS
0570H	ATN
0571H	HYPSIN
0572H	HYPCOS
0573H	HYPTAN
0574H	HYPASN
0575H	HYPACS
0576H	HYPATN
0577H	LN
0578H	LOG
0579H	EXP
057AH	SQR
057BH	ABS
057CH	SGN
057DH	INT
057EH	FIX
057FH	FRAC
0581H	DEGR
0582H	DMS
0586H	PEEK
058AH	EOF
058DH	FRE
0590H	ROUND
0592H	VALF
0593H	RAN#
0594H	ASC
0595H	LEN
0596H	VAL
059BH	HYP
059CH	DEG
05A7H	REC
05A8H	POL
05AAH	NPR
05ABH	NCR
05ACH	HYP
0697H	DMS$

069BH	INPUT
069CH	MID$
069DH	RIGHT$
069EH	LEFT$
06A0H	CHR$
06A1H	STR$
06A3H	HEX$
06A8H	INKEY$
0747H	THEN
0748H	ELSE
07B6H	TAB
07BBH	ALL
07BCH	AS
07BDH	APPEND
07C0H	STEP
07C1H	TO
07C2H	USING
07C3H	NOT
07C4H	AND
07C5H	OR
07C6H	XOR
07C7H	MOD

F0 to F9 in the file area are general-purpose files that can be used as input and output destinations for C and CASL source files and BASIC. The data storage format is exactly the same as a general OS such as MS-DOS. For example, the file in Listing 6 is stored in memory as shown in Table 10. The line feed code is ODH, 0AH, and the end-of-file code is 1AH, which is exactly the same as MS-DOS. The list of

programs B-1. CHKPFAV4.BAS for checking programs and file areas is shown, so you can use this to check the contents of this section yourself.

Listing 6. Sample file
HELLO, WORLD!

Table 10. Memory storage format in Listing 6

Data														
48 H	45 E	4C L	4C L	4F O	2C ,	57 W	4F O	52 R	4C L	44 D	21 !	0D CR	0A LF	1A EOF

Storage Format of Variable Data

When variables are used in program execution, CAL mode, etc., numerical variables and character variables that have not been registered in the variable table, that is, for the first time, are automatically registered and instantiated by the BASIC system. Also, array variables cannot be instantiated automatically by the BASIC system , and the user must intentionally declare and instantiate them with a DIM statement (probably to prevent unnecessary memory consumption). Figure 3 shows the situation of materialization and storage in the BASIC work area as described above.

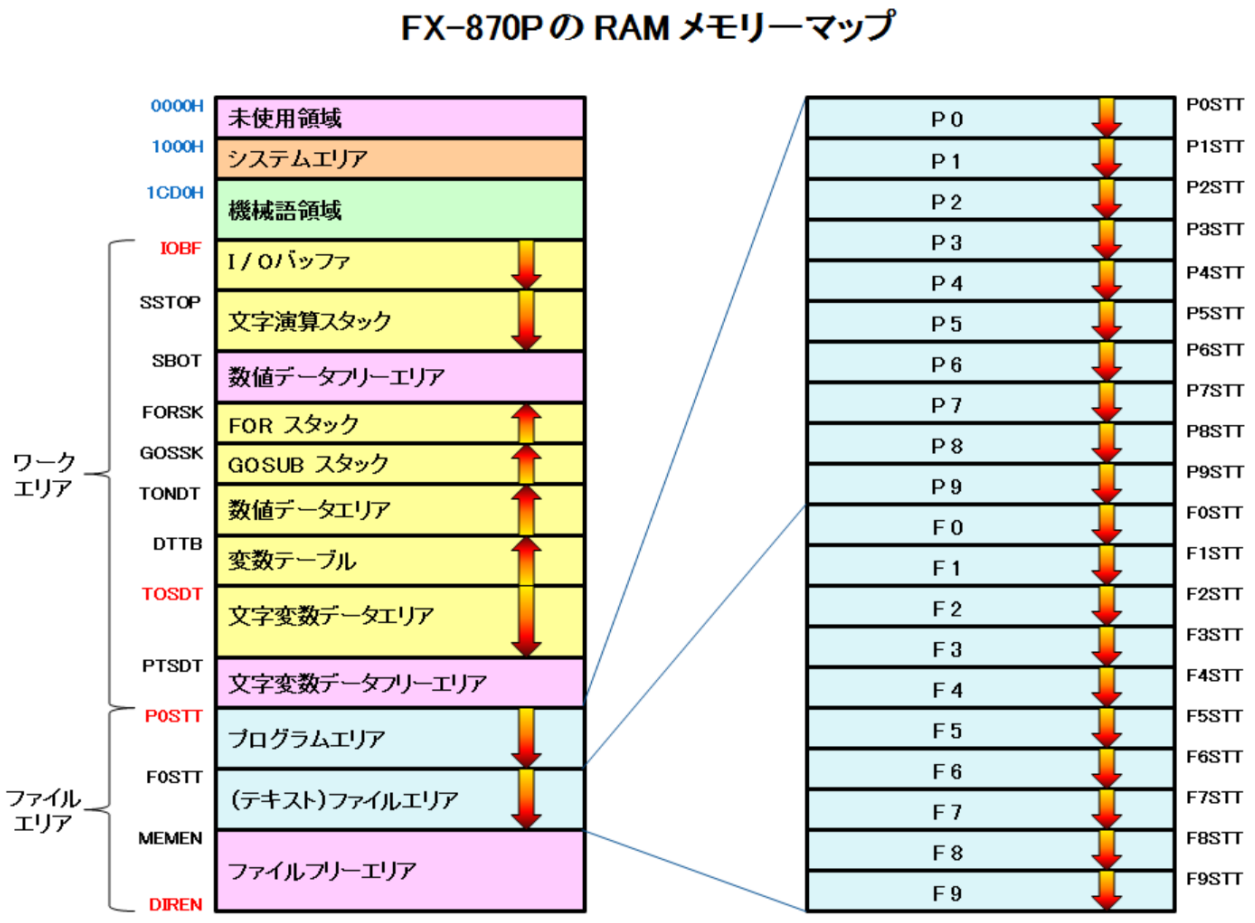


Figure 3. FX-870P / VX-4 RAM Memory Map (BASIC)

In Figure 3, the three numbers at the top of the RAM written in blue are fixed values. IOBF, TOSDT, P0STT, and DIREN written in red are values that can be set by the user, and BASIC cannot be changed by themselves. 1CD0H to (IOBF-1) is a machine language area, and IOBF cannot normally be changed, and 1CD0H and the machine language area is 0 bytes. However, the machine language area can be secured by changing with extended CLEAR . P0STT and TOSDTT can be set with the BASIC CLEAR statement, (P0STT-IOBF) is the work area size, (P0STT-TOSDT) is the variable area size, and is actually the area where character variables and array character variables are stored. . DIREN is the final RAM address of FX-870P / VX-4 and is not normally changed. Usually 1 byte, but if you make a few bytes free by changing DIREN, you can use it to store high scores of the game. **The machine language area is destroyed when a C program is executed in C language mode, but the data in the area after DIREN seems to be immune to destruction.**

At this time, the BASIC system uses the I / O buffer from the IOBF and the memory for the character operation stack, and uses the memory from the TOSDT as the variable table, numeric variable / array numeric variable data area, GOSUB stack, and FOR stack in the reverse address direction. . Finally, it is used as a data area for character variables and array character variables in the address forward direction from TOSDT.
Jun Amano has already explained the basics of variable storage format (Ref. (13)).). This time, in order to complete the information of the variable storage format , the analysis result using the program B-2. OUTWRKV4.BAS that outputs the variable storage status of the work area to a file is described.

Tables 11 to 13 show the results of analyzing the storage format of the materialized variables by this program.
The variable table is searched in the forward direction from the address stored in the DTTB. Data

addresses are (DTTB) to (TOSDT) -1. The data format of the variable table is variable attribute (1 byte), number of characters of variable name (1 byte), variable name (variable length), pointer to actually store data (2 bytes). (Number of characters of variable currently being searched) + 4 should be added to (address of variable attribute currently being searched). In addition, the variables are searched in the reverse order of the materialized variables, and the last materialized variable is first hit in the search. There are four types of variable attributes: character variables, numeric variables, array character variables, and array numeric variables, which are 20H, 28H, A0H, and A8H, respectively. Therefore, four variable types can exist simultaneously with the same variable names as A $, A, A $ (), and A ().

The numeric data area is an area with addresses (TONDT) to (DTTB) -1, and stores data for numeric variables and array variables. Basically numeric data is packed and little endian encoded BCD floating point formatHowever, in the case of an array variable, the pointer of the variable table points to the declaration information of the array numeric variable. The first byte is the dimension of the array variable, and the maximum value of each subscript is arranged for each array dimension by 2 bytes. Multidimensional array variables of two or more dimensions must be managed with a one-dimensional

$$\sum_{j=1}^{N} I_j \cdot \prod_{k=0}^{j-1} M_k = I_N \cdot (M_{N-1} + 1) \cdot (M_{N-2} + 1) \cdot \cdot \cdot (M_1 + 1)$$

$$+ I_{N-1} \cdot (M_{N-2} + 1) \cdot \cdot \cdot (M_1 + 1)$$

$$\cdots\cdots\cdots\cdots$$

$$+ I_1$$

, ここで $M_0 = 0$.

subscript inside the BASIC system, but they are unified so that the rightmost subscript is inside the loop. That is, if DIM A (M_N , M_{N-1} , ..., M_1) is declared , one of the array numeric variables written as A (I_n , I_{n-1} , ..., I_1) You can think of the subscripts of the elements as being unified in the expression inside . In addition, for array numeric variables, there is basically no memory size change after securing the data storage area as declared in the DIM part in the numeric data area (initial value 0), so BASIC system management is a character array variable. It is easier compared to

The character variable data area is an address area from (TOSDT) to (PTSDT) -1, and stores data for character variables and array character variables. In the case of a character variable, the first byte pointed to by the variable table pointer is the number of characters in the data stored in the variable, and character string data of that number of characters is stored subsequently. In the case of an array character variable, the declaration information of the array numeric variable is contained in the same manner as the array numeric variable. However, the one-dimensionalization inside a multidimensional array is the same as a numeric array variable, but each numeric data is 8 bytes, but the character variable is variable, so the internal one-dimensional subscript is searched from 0. The target index must be reached, and access is less efficient than array numeric variables. In addition, substitution and deletion of character data (substitution of "") does not leave unnecessary data in the character variable data area, and the BASIC system automatically manages memory. In other words, when the data of a character variable or array character variable is changed and the size of the character variable data area needs to be changed, it is materialized after that variable (in the case of an array variable, the one-dimensional subscript is larger Subscript) data is shifted by the necessary amount, and the variable table pointer is also shifted by the shift amount. Therefore, **the load of the BASIC system due to the substitution of the character variable is smaller for the character variable (character array variable) that is materialized last.**

Also, instead of clearing the work area contents with CLEAR, only the pointers are changed. Variable initialization is performed when a variable is registered in the variable table.

The above analysis results are summarized as follows.
Table 14 shows the memory usage of variables.

Table 14. Variable Memory Usage

Variable type	Variable Table Usage (byte)	Data Storage Destination	Data storage size (byte)
Numeric variable	(Number of characters in variable name) + 4	Numerical data area	8
Array numeric variables			1+ (number of dimensions) x 2+ (number of array elements) x 8
Character variable		Character variable data area	(Number of characters in the assigned string) + 1
Array numeric variables			1 + (number of dimensions) x 2+ (number of array elements) + (number of characters in the string assigned to all array elements)

In addition, the following precautions are effective for speeding up BASIC.

- The registration order of the variable table and the search order of the variable table are reversed, and the search time is shorter for the variables registered in the variable table later. Therefore, it is effective for speed-up to start using frequently used variables as much as possible.
- When it is necessary to change the size of the character variable data area by changing the data of a character variable or array character variable, the data is materialized after that variable (in the case of an array variable, it is a large subscript with a one-dimensional internal subscript). Must be shifted as much as necessary, and the variable table pointer must also be shifted by the shift, which places a heavy load on the BASIC system. Therefore, **using frequently used character variables and character array variables as soon as possible is especially effective for speeding up BASIC programs.**

Table 11. Variable Table (DTTB) analysis Results

Address (Hexadecimal)	Variable Table Data						
	Attribute 1byte	Word Count 1byte	Variable Name Variable Length			Pointer 2 Bytes	
3A8A	20	01	53			EF	3A
	Ch	01	S			3AEF	
3A8F	28	03	53	54	30	ED	39
	Nu	03	S	T	0	39ED	
3A96	28	02	4E	58		F5	39
	Nu	02	N	X		39F5	
3A9C	28	02	53	54		FD	39
	Nu	02	S	T		39FD	
3AA2	28	02	41	44		05	3A
	Nu	02	A	D		3A05	
3AA8	20	01	46			ED	3A
	Ch	01	F			3AED	
3AAD	28	01	4A			0D	3A
	Nu	01	J			3A0D	
3AB2	28	01	49			15	3A
	Nu	01	I			3A15	
3AB7	A0	03	51	57	45	D6	3A
	AC	03	Q	W	E	3AD6	
3ABE	A8	03	50	4F	49	1D	3A
	AN	03	P	O	I	3A1D	
3AC5	20	02	42	43		D0	3A
	Ch	02	B	C		3AD0	
3ACB	28	01	41			82	3A
	Nu	01	A			3A82	

Table 12. Numerical Data Area (TONDT) Analysis Results

Address (Hexadecimal)	Variable Table Data	Remarks

39ED	00	00	00	00	16	48	41	10	ST0 value
	14816								
39F5	00	00	00	00	86	49	41	10	NX value
	14986								
39FD	00	00	00	00	29	48	41	10	ST value
	14829								
3A05	00	00	00	00	86	49	41	10	AD value
	14986								
3A0D	00	00	00	00	00	00	03	10	J value
	Three								
3A15	00	00	00	00	73	48	41	Ten	I value
	14873								
3A1D	02	02	00	03	00				DIM statement for array numeric variable POI () Information declared in DIM POI (2,3). The first byte is the dimension. Defines the maximum subscript value by 2 bytes.
	2	2	Three						
3A22	00	00	00	00	00	00	00	00	POI (0,0) value
	0								
3A2A	00	00	00	00	00	00	01	66	POI (0,1) value
	-1E60								
3A32	00	00	00	00	00	00	02	66	POI (0,2) value
	-2E60								
3A3A	00	00	00	00	00	00	00	00	POI (0,3) value
	0								
3A42	00	00	00	00	00	00	04	66	POI (1,0) value
	-4E60								
3A4A	00	00	00	00	00	00	05	66	POI (1,1) value
	-5E60								
3A52	00	00	00	00	00	00	06	66	POI (1,2) value
	-6E60								
3A5A	00	00	00	00	00	00	00	00	POI (1,3) value
	0								
3A62	00	00	00	00	00	00	08	66	POI (2,0) value

	-8E60								
3A6A	00	00	00	00	00	00	09	66	POI (2,1) value
	-9E60								
3A72	00	00	00	00	00	00	11	66	POI (2,2) value
	-1E61								
3A7A	00	00	00	00	00	00	00	00	POI (2,3) value
	0								
3A82	20	01	89	67	45	twenty three	01	Ten	A value
	1.234567890120								

Table 13. Character Variable Data (TOSDT) Analysis Results

Address (Hexadecimal)	TOSDT Data									Remarks
3AD0	05	43	41	53	49	4F				BC $ value. The first byte is the number of characters.
	5	"CASIO"								
3AD6	01	05	00							Information declared in DIM statement DIM QWE (5) of array character variable QWE $ (). The first byte is the dimension. Defines the maximum subscript value by 2 bytes.
	1	Five								
3AD9	00									QWE $ (0) value
	"" (no data; null)									
3ADA	06	50	4F	43	4B	45	54			The value of QWE $ (1).
	6	"POCKET"								
3AE1	00									QWE $ (2) value
	"" (no data; null)									
3AE2	08	43	4F	4D	50	55	54	45	52	The value of QWE $ (3).
	8	"COMPUTER"								
3AEB	00									QWE $ (4) value
	"" (no data; null)									
3AEC	00									QWE $ (5) value
	"" (no data; null)									
3AED	01	30								F $ value
	1	"0"								
3AEF	00									The value of S $ (4)
	"" (no data; null)									

PS
I would like to thank Jun Amano because I could not understand the variable storage format so far without the website of Jun Amano.

3-3 Appendix

A-1. PB-1000 Memory Map

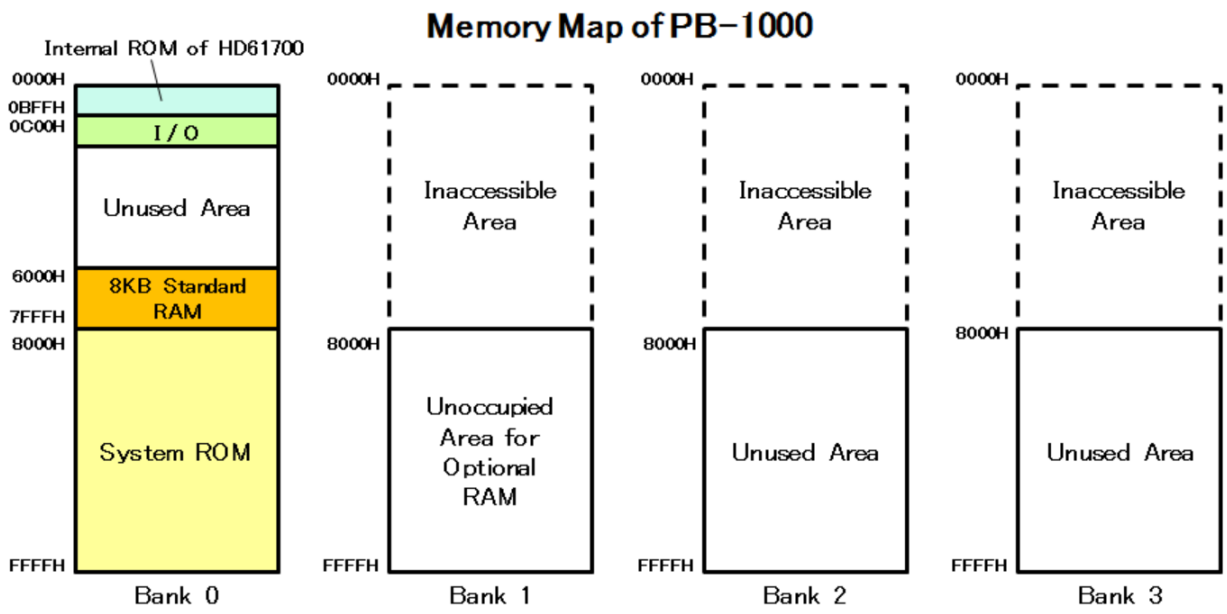


Figure A1. PB-1000 Memory Map

A memory map of Casio PB-1000 is shown in Figure A1 (Reference (11)). By switching the bank of the address space from 8000H to FFFFH, the BANK 0 system ROM and the BANK 1 RAM (extension RAM) are accessed. In addition, 0000H to 7FFFH are designed so that only BANK 1 can be accessed even if another bank is specified, and addresses 0000H to 7FFFH of BANK 1 to 3 cannot be accessed.

A-2. BCD floating point format and internal format

Casio's pocket computers, except for some logical operations, perform numerical calculations using BCD floating-point data, and all numerical variables and array numerical variables are stored as BCD floating-point data. PB-1000's BCD floating-point format and internal storage format are described by Polish Piotor Piatek (Ref. (14)). However, there are places where explanation is insufficient and there are places where it is difficult to understand.
First, to conclude, the data storage format for numeric data is

1. **Casio's BCD** floating-point data format,
2. **Little** endian encoding,
3. **Packed Little endian** (Packed little endian encoding)

It is easier to understand if you understand in the order

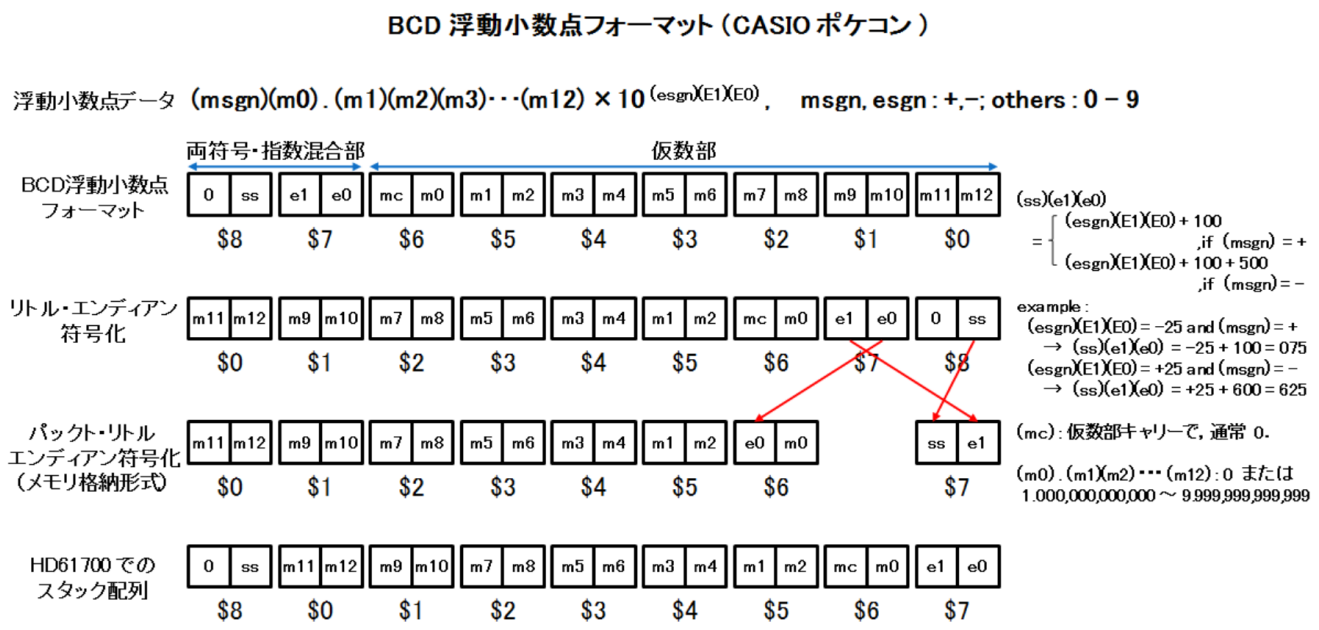


Figure A2. BCD floating-point data format (Casio)

In FX-870P / VX-4, the basic format of numeric data is a normalized decimal exponent with a signed mantissa part of 13 digits and a signed exponent part of 2 digits.

That, (msgn) (m0). (M1) (m @ 2) · · · (m11) (m12) × 10 $^{(Esgn)\,(E1)\,(E0)}$
It is expressed.Here, (msgn) and (esgn) are the sign of the mantissa part and the exponent part (Exponetial part), respectively, and are + or-. Others are numbers from 0 to 9, the exponent part is 2 digits, and the mantissa part is normalized, so it is 1.000,000,000,000 to 9.999,999,999,999. By following this rule, you can make a number such as
+1.123456789012 × 10 $^{-25}$

In addition, 0 is expressed by setting all 0s to 0.
This number is expressed in BCD (Binary-Coded Decimal) with 18 digits of 9 bytes. At this time, MSD (Most Siginificant Digit) is 0, and the next three digits (ss) (e1) and (e0) are a combination of the sign and exponent part of the mantissa part (both sign and exponent mixture part), The mantissa part carry (mc) is usually 0, and the remaining 13 digits are the mantissa part (m0). (M1) (m2) ... (m11) (m12). Since the mantissa part is the original numerical value, there is no problem, but the sign / exponent mixed part is given as follows.

The sign / exponent mixed part (ss) (e1) (e0) is first considered to be a decimal three-digit number and is offset by +100 to the exponent part. The sign of the mantissa part is minus (-) Only in some cases, you can think of +500.
For example, if the exponent part (esgn) (E1) (E0) = -25 and the mantissa sign (msgn) = +, then
(ss) (e1) (e0) = -25 + 100 = 075,
exponent part (esgn) If (E1) (E0) = +25 and the mantissa code (msgn) =-,
(ss) (e1) (e0) = + 25 + 100 + 500 = 625

The reason why the value is +500 is presumed to be that sign calculation of the mantissa part can be performed simultaneously with exponent addition and subtraction in multiplication and division. For example, when multiplying-, 500 and 500 are added together to become 1000, and the significand sign is + at the same time.

The above description can encode numbers in BCDC floating point format. For example, -1.123456789012 × 10^{-29} is
05 71 01 12 34 56 78 90 12 in the BCD floating point format

If you can understand the BCD floating-point format, it becomes a CPU problem. **FX-870P / VX-4 CPU HD61700 is a little endian system, so loading to the main register is performed in ascending order from the least significant byte.** For example, 05 71 01 12 34 56 78 90 12 is loaded as
12 90 78 56 34 21 01 71 05 from $ 0 to $ 8.

The load state to this register is the first explanation of Piotr Piatek's BCD format, and the original BCD floating-point format is not specified, so the packed little-endian encoding state, which is the memory storage format, is difficult to understand. ing.
Since MSD and (mc) are 0 in the original BCD floating point format, moving (e0) to (mc) and shifting (ss) (e1) up one digit to reduce 1 byte • Little-endian encoding, which is stored in memory using this method.

For example, little-endian encoded data 12 90 78 56 34 21 01 71 05 (numeric value: -1.123456789012 × 10^{-29}) is stored as
12 90 78 56 34 21 11 57 in the memory

In Figure B2, it seems that digit movement is complicated and difficult to understand in packed little endin coding, but it can be seen that it is natural digit movement when considered in the original BCD floating point format. In fact, this operation is performed when $ 0, $ 1, ..., $ 8 contains floating point data.

```
DIUW  $ 7        ; Digit Up of ($ 8, $ 7) pair
OR    $ 6, $ 7   ; $ 6 <-$ 6 or $ 7, where the upper digit of $ 7 and the lower one are equal to
                 (e0) and zero respectively.
LD    $ 7, $ 8   ; $ 7 <-$ 8
```

Can be compressed. Conversely, compressed numeric data loaded from $ 0 to $ 7 from memory is

```
LD    $ 8, $ 7   ; $ 8 <-$ 7
LD    $ 7, $ 6   ; $ 7 <-$ 6
DIDW  $ 8        ; Digit Down of ($ 8, $ 7)
AN    $ 6, & H0F ; clear the upper digit of $ 6
```

It is reasonable to realize the original state.

The successor FX-890P / Z-1 CPU is also an x86-based 80186, little-endian CPU, and the storage format in memory is the same.

The FX-3870P / VX-4 provides a program that displays the internal storage format of numeric data in hexadecimal format in B-3 , so you can check the memory storage format yourself.

At the bottom of Fig. A2, Piotr Piatek explains the arrangement on the stack of numerical data on PB-1000 explained by HP. He does not give a reason just by fact, but **this stacking arrangement is for reasons specific to HD61700.** When saving $ 0 to $ 8 with BCD floating-point data to the user stack,

it can be accelerated by using multi-byte PUSH, but only up to 8 bytes are supported. Therefore, it is necessary to push 1 byte separately. Therefore,

```
PHUM  $ 7, 8  ; PushH User-stack Multibyte for ($ 7, ..., $ 0)
PHU   $ 8     ; PusH User-stack for $ 8
```

Is saved in the user stack as shown in the figure. To pop the saved data,

```
PPU   $ 8     ; PoP User-stack for $ 8
PPUM  $ 0, 8  ; PoP User-stack Multibyte for ($ 7, ..., $ 0)
```

What should I do? Here, the register number is different between DIUW and DIDW, PHUM and PPUM. This is also a specification unique to HD61700. When restoring packed little-endin encoded data, the first two instructions are used.

```
LDW  $ 7, $ 6  ; ($ 8, $ 7) <-($ 7, $ 6)
```

However, if $ 7 ← $ 6 and $ 8 ← $ 7 are executed, $ 6 is copied up to $ 8 of the most significant byte, and the target operation is not achieved.
Finally, when pushing to the user stack,

```
PHU   $ 8     ; PusH User-stack for $ 8
PHUM  $ 7, 8  ; PushH User-stack Multibyte for ($ 7, ..., $ 0)
```

If you push $ 8 first, it will be packed in the normal order of $ 0, $ 1,…, $ 7, $ 8 from the low address side of the stack. Whether the FX-870P and VX-4 are still using the PB-1000 is currently unknown, so it is unclear.

PS: I would like to thank Piotr Piatek for not being able to understand the BCD floating-point format so far.

3-4 BASIC Programs

This time, in order to independently investigate the internal information of FX-870P / VX-4, several programs were created and investigated. Below is a list of the main programs, a brief explanation of the programs and how to use them. Such a program is unnecessary in nature, but it can be used as a reference, such as output to a file.

B-1. CHKPFAV4.BAS: Check program area and file area

Listing B-1. CHKPFAV4.BAS

```
100 'CHKPFAV4.BAS
110 'check program and file area
120 '
130 'for FX-870P / VX-4
140 '
150 'program by 123
160 'since 30th, Oct., 2010.
170 '
190 '
200 INPUT "1:disp addrs, 2:disp one of P0-F9";MD
210 IF MD=2 THEN GOSUB 500 ELSE GOSUB 300
220 END
290 '*DISPADR:'disp addrs
300 POI=&H18A7:'addr of P0
310 PRINT "Addresses of P0-F9"
320 FOR I=0 TO 9
330 AD=POI:GOSUB 1000
340 PRINT "P";RIGHT$(STR$(I),1);":";HEX$(AD);" ";
350 POI=POI+2
360 NEXT
370 PRINT
380 FOR I=0 TO 9
390 AD=POI:GOSUB 1000
400 PRINT "F";RIGHT$(STR$(I),1);":";HEX$(AD);" ";
410 POI=POI+2
420 NEXT
430 'PRINT
440 AD=POI:GOSUB 1000
450 PRINT "MEMEN:";HEX$(AD)
```

Listing B-2. OUTWRKV4.BAS

```
000 ' OUTWRKV4.BAS
110 ' output data of work area of FX-870P/VX-4
120 ' to File F0-9
130 ' for FX-870P/VX-4
140 '
150 ' programmed by 123
160 ' since 30th, Oct., 2010.
170 '
180 ' S$ must be emobodied at lat for string data stability!!
190 '
200 ' Data input
```

```
210 A=1.23456789012
220 BC$="CASIO"
230 DIM POI(2,3)
240 DIM QWE$(5)
250 FOR I=0 TO 2
260 FOR J=0 TO 2
270 POI(I,J)=(I*4+J)*(-1E60)
280 NEXT
290 NEXT
300 QWE$(1)="POCKET"
310 QWE$(3)="COMPUTER"
320 F$="0":AD=0:ST=0:NX=0:ST0=0:S$=""
490 ' Output work area to F0...9
500 INPUT "Output FileNumber";F$
510 RESTORE#("F"+F$)
520 WRITE#:'clear file
530 WRITE#"WORK AREA DATA"
540 ' TONDT(&H189F):numerical data
550 AD=&H189F:GOSUB 1000:ST=AD
560 AD=&H18A1:GOSUB 1000:NX=AD
570 WRITE#"TONDT:numerical data"
580 GOSUB 1100
590 ' DTTB(&H18A1):variable table
600 ST=NX
610 AD=&H18A3:GOSUB 1000:NX=AD
620 WRITE#"DTTB:variable table"
630 GOSUB 1100
640 ' TOSDT(&H18A3):string data
650 ST=NX
660 AD=&H18A5:GOSUB 1000:NX=AD
670 WRITE#"TOSDT:string data"
680 GOSUB 1100
690 ' PTSDT(&H18A5):free area of string
700 ST=NX
710 AD=&H18A7:GOSUB 1000:NX=AD
720 WRITE#"TOSDT:free area of string"
730 GOSUB 1100
740 END
990 '*GETAD:'get address
1000 AD=PEEK(AD)+PEEK(AD+1)*256
1010 RETURN
1090 '*OUTHEX
1100 ST0=ST AND &HFFF0
1110 S$=""
1120 FOR I=ST0 TO NX-1
1130 IF (I AND &HF)=0 THEN S$=HEX$(I)+":"
1140 IF I>=ST THEN S$=S$+" "+RIGHT$(HEX$(PEEK(I)),2) ELSE S$=S$+" "
1150 IF (I AND &HF)=15 OR I=NX-1 THEN WRITE# S$:S$=""
1160 NEXT
1170 WRITE#
1180 RETURN
1190 ' end of program
```

- Since WRITE # cannot be output without line breaks, as in PRINT A $; in the PRINT statement, the file is output after combining it into a character variable S $.
- For numeric variables and array numeric variables, changing the value only affects the data contents, but for character variables and character array variables, if the contents change, the pointer values and character variables stored in the variable table It changes to the state of the data area. In order to minimize the impact, S $ whose contents change frequently in the program is used last in the program and registered in the variable table. In this way, other character variables and array variables are free from the influence of dynamic fluctuation of character variable data caused by program operations.

  There are two ways to operate the program.

- Execute CLEAR (the CLEAR command may be placed at the top of the program), clear the variables, and then simply execute RUN.
  The output results are useful for understanding how variables are stored. However, the content of the character data area of S $ (the last 1 byte of the TOSDT area of the output data) is not 0 but contradicts, but is actually 0 (S $ = "").
-
  Executes RUN500
  after assigning character variables and deleting the contents (substituting "") . Thereby, the dynamic change of the character variable data area can be confirmed.

Listing B-3. CHKAV4.BAS: Numerical data of numerical variable A is displayed in binary (for BCD floating point format investigation)

Listing B-4. CHKAV4.BAS

```
100 'CHKAV4.BAS
110 'check A, numerical variable
120 'to inspcet the inner represenation
130 'for FX-870P, VX-4
140 ' programmed by 123
150 ' since 28th,Oct.,2010
200 AD=&H18A1
210 DTTB=PEEK(AD)+PEEK(AD+1)*256
220 AD=&H18A3
230 TSDT=PEEK(AD)+PEEK(AD+1)*256: 'TOSDT
240 '
250 FOR AD=DTTB TO TSDT-1
260 IF PEEK(AD)=&H28 AND PEEK(AD+1)=1 AND PEEK(AD+2)=&H41 THEN 310
270 NEXT
280 PRINT "Failed to find var A!"
290 END
300 '
310 AD=PEEK(AD+3)+PEEK(AD+4)*256
320 PRINT "A= ";A
330 FOR II=0 TO 7
340 PRINT RIGHT$(HEX$(PEEK(AD+II)),2);" ";
350 NEXT
360 PRINT
370 END
380 ' end of program
```

Examine the variable table of DTTB to TOSDT in the system area and output the internal format of the value of numeric variable A in hexadecimal. This allows you to check the storage format of numeric variables in memory.
In the FX-890P / Z-1 successor to FX-870P / VX-4, A is a fixed variable ("Z-1 / FX-890P Utilization Research"), so without examining the variable table, The program is simple because it only outputs the contents of a fixed address. For reference, the equivalent program for FX-890P / Z-1 is shown in List B-3. The reason why I used II instead of I in the FOR to NEXT loop is because I was not able to use I because the original program targeted not only A but also variables A to Z. It is.

Listing B-5. CHKAZ1.BAS (for FX-890P / Z-1)

```
000 'CHKAZ1.BAS
110 'check A, numerical variable
120 'to inspcet the inner represenation
130 'for FX-890P, Z-1
140 'program by 123
150 'since 28th, Oct., 2010
200 AD = & H196F
210 PRINT "A ="; A
220 FOR II = 0 TO 7
230 PRINT RIGHT $ (HEX $ (PEEK (AD + II)), 2); "";
240 NEXT
250 PRINT
260 END
270 'end of program
```

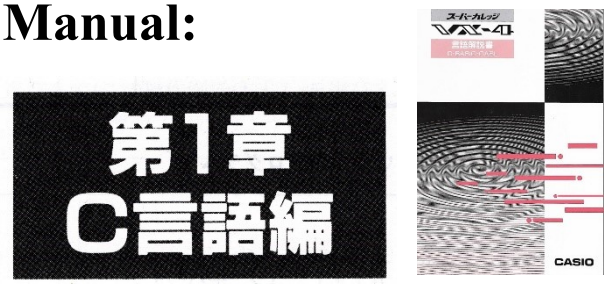
IV. C - Referenz

While the BASIC Manual part was shown very well, the C-Manual part is not executed on the Japanese website. On the Internet and in books enough references to look up the C language (see operating instructions "Introduction to C programming Casio PC-2000C").

The commands from the original manual are listed here using screenshots. Despite the Japanese characters integrated as a result, the existing command set can be recognized and the examples also show how they are used. For further interest you can use it to experiment and compare with other C manuals.

4-1 Sides from the Original Manual:

Starts C with ON / Shift / C →



C言語モード

まず、電源をONにして、C言語モードに入ります。

操作		表示	
①	ON SHIFT C	①	(C) F 0 1 2 3 4 5 6 7 8 9 3355B F2>Run/Load/Source/Cal

メニュー	キー	機　　能
Source	Sキー	プログラムの入力と修正が行なえます。
Load	Lキー	プログラムがロードされます。
Run	Rキー	プログラムが実行されます。
Cal	Cキー	マニュアル計算モードに戻ります。

the C-Commands list

キーワードとライブラリー関数名

abort	default	goto	sinh
abs	do	gotoxy	sizeof
acos	double	if	sprintf
acosh	else	inport	sqrt
angle	enum	int	sscanf
asin	exit	log	static
asinh	exp	log10	strcat
atan	extern	long	strchr
atanh	fflush	main	strcmp
auto	fgetc	malloc	strcpy
beep	fgets	outport	strlen
break	float	pow	struct
breakpt	for	printf	switch
calloc	fprintf	putc	tan
case	fputc	putchar	tanh
char	fputs	puts	typedef
clearerr	free	register	union
clrscr	fscanf	return	unsigned
const	getc	scanf	void
continue	getch	short	volatile
cos	getchar	signed	while
cosh	gets	sin	

型宣言子	本機のC言語
char	8ビット
short	16ビット
int	16ビット
long	32ビット
float	32ビット
double	64ビット

float	0，±1e−63 ～ ±9.99999e+63
doudle	0，±1e−99 ～ ±9.9999999999e+99

	キーワード	意味と用法
	auto	局所変数の記憶クラス指定
	break	for, do, while, switch文からの脱出
×	case	switch文の名札。使用不可
	char	文字型データ(8ビット長)の宣言子
×	const	定数の宣言。使用不可
	continue	for, do, whileにおいて、次の繰り返しへジャンプ
×	default	switch文で該当しないときの飛び先。使用不可
	do	処理の繰り返し実行。do {~} while(式);
	double	倍精度浮動小数点型(64ビット長)の宣言子
	else	if文とともに使用。if(式) {~} else {~}
×	enum	列挙型の宣言子。使用不可
	extern	外部変数や外部定義の記憶クラス指定
	float	単精度浮動小数点型(32ビット長)の宣言子
	for	繰り返し実行。for(式1;式2;式3) {~}
	goto	指定したラベルへのジャンプ
	if	もし式が真ならば実行。if(式) {~}
	int	整数型(16ビット長)の宣言子
	long	倍長整数型(32ビット長)の宣言子
	register	レジスタ変数の記憶クラス指定。autoと同じ
	return	関数の値を返し、呼び出しへ戻る
×	sigened	符号つき型の宣言。使用不可
	sizeof	データ型の長さ。sizeof(型)は使用不可
	short	短整数型の宣言子。intと同じ
	static	静的変数の記憶クラス指定
×	struct	構造体の宣言子。使用不可
×	switch	条件による分岐。使用不可
×	typedef	新しいデータ型の指定。使用不可
×	union	共用体の宣言子。使用不可
	unsigned	符号なし整数型の宣言子
	void	値を返さない関数の型宣言子
×	volatile	プログラムの外側から変更できる型の宣言。使用不可
	while	繰り返しの実行。while(式) {~}

	型	例
10進定数	int long	0～32767 32768～2147483647
8 進定数	int unsigned int long	00～077777 0100000～0177777 0200000～017777777777
16進定数	int unsigned int long	0x0000～0x7FFF 0x8000～0xFFFF 0x10000～0x7FFFFFFF

演算子の種類と優先順位および結合規則

優先順位	演算子		結合規則
高い	.	( ) 〔 〕	→
	単項	! ^ ++ −− (型) *注1) & sizeof	←
	乗除	*注2) / %	→
	加減	+ −	→
	シフト	≪ ≫	→
	比較	> < >= <=	→
	等価比較	== !=	→
	ビットAND	&	→
	ビットXOR	^	→
	ビットOR	¦	→
	論理AND	&&	→
	論理OR	¦¦	→
	代入	= += −= *= /= その他	←
低い	順序	,	→

→ 左から右に演算
← 右から左に演算

注1) 間接指定記号の *
注2) 乗算記号の *

キャラクター	ASCII コード(10進)	ASCII コード(16進)
A	65	41
Z	90	5A
0	48	30
9	57	39

分岐や繰り返しなどの制御構造

条件分岐	
if （式） 　文； }	①式が真ならば、 ②文を実行します。
if （式）｛ 　文； } else｛ 　文； }	①式が真ならば、 ②文1を実行し、 ③それ以外ならば(偽ならば)、 ④文２を実行します。
if （式1）｛ 　文1； } else if （式２）｛ 　文２； } else if （式３）｛ 　文３； …… } else if（式n） 　文n； } else ｛ 　文n＋1； }	①式１が真ならば、 ②文１を実行し、 ③それ以外で（偽で）、式２が真ならば、 ④文２を実行し、 ⑤それ以外で(偽で)、式３が真ならば、 ⑥文３を実行し、 …… ⑦それ以外で(偽で)、式nが真ならば、 ⑧文nを実行し、 ⑨それ以外なら（式１～式nが偽なら） ⑩文n＋１を実行します。
繰り返し制御	
while（式）｛ 　文； }	①式が真のあいだ ②文を実行します。
for（式1；式2；式3）｛ 　文； }	①式１を実行して初期設定を行ないます。 ②式２が真なら、文を実行します。 ③式３を実行して条件を更新をします。 ④式２が真なら、文を実行します。 ⑤以下、式２が真のあいだ、文の実行、式３による更新を繰り返します。
do ｛ 　文； } while(式)；	①文を実行します。 ②式が真なら、元に戻ります。 ③以下、式が真のあいだ、文の実行を繰り返します。

int x, *px; x=*px;	左の例ではpxが指している内容がxに代入されます。

int x, y, *px;
px=&x;
y=*px;
	左の例ではxのアドレスがpxに代入された後、pxが示すアドレスの内容をyに代入します。
	すなわち、y=x;と同じことになります。

main()
{
char *p;
p="Casio";
printf("%c %s ¥n", *p,p);
}
	文字列においてポインタを用いると、文字列を単一文字に分解することができます。実行させると結果は次のようになります。
	C Casio
	ポインタpはCasioのCを指します。

main()
{
char *p;
p="Casio";
printf("%c %c ¥n", *p, *(p+2));
}
	CasioのCとsを取り出すには左のようにします。

main()
{
int i, a[5], *pa;
pa=a;
for(i=0;i<=4;i++)
*(pa+i)=i;
for(i=0;i<=4;i++)
printf("%d" a[i]);
printf("¥n");
}
	配列の各要素はポインタで示すこともできます。左の例では配列aの各要素をpaで示しています。
	配列名aは、配列の最初の要素a[0]を指すポインタを表わします。したがって、配列の各要素を指すポインタは次のようになります。
	pa …… a[0]
	pa+1 …… a[1]
	pa+2 …… a[2]
	pa+3 …… a[3]
	pa+4 …… a[4]

main()
{
char *pc, c[80];
pc=c;
strcpy(pc, "abcdefgh");
printf("%s¥n" ,pc);
}
	ポインタは変数のアドレスを持っているだけです。したがって、場合によっては格納領域の確保は別に行なう必要があります。
	左の例では、ポインタpcと80文字の格納領域を確保するために配列cを宣言し、ポインタpcを配列のポインタcと共通にしています。

無限ループ	
for(;;) 　文；	①文を繰り返し実行します。 （for文の初期設定、条件判断、条件更新がない）
while(1) 　文；	①文を繰り返し実行します。 （while文の条件がいつも真（1））
do 文； while(1)；	①文を繰り返し実行します。 （do～while文の条件がいつも真（1））
break文、continue文	
while（1）｛ 　…… 　if（式）break； 　…… ｝	①while文による無限ループが実行されます。 ②式が真ならbreak文が実行され、無限ループから抜け出します。
while（式1）｛ 　…… 　if（式2）continue； 　…… ｝	①式1が真のあいだ、｛　｝の中が実行されます。 ②式2が真なら、continue文が実行され、while文に戻り、式1が実行されます。
無条件ジャンプ	
goto ラベル； …… ラベル；	①ラベルの位置に無条件にジャンプします。

4-2 The C-Code in Original Manual

RUN	（書式） RUN RUN＞"PRN：" RUN＞"prn："
（機能） プログラムをロードして実行させます。 "PRN："または"prn：" を指定すると、実行結果をプリンタに出力します。 指定を省略すると、画面に出力します。	
EDIT	（書式） EDIT
（機能） プログラムの編集を行なうエディタに入ります。 プログラムの実行でエラーが発生したときにEDITを入力すると、エディタに入りエラー発生行を表示します。 ブレークでプログラムの実行が中断しているときにEDITを入力すると、エディタに入りブレークした行を表示します。	
TRON	（書式） TRON
（機能） トレースしながらプログラムを実行するトレース機能を指定します。トレース機能を指定してプログラムを実行すると、プログラムを１行実行するごとに、次に実行する行を表示します。 トレース中のキー操作 EXE キー：実行を続行します。 C キー：実行を続行します。 BRK キー：実行を中断します。 トレース機能は、起動時はOFFになっています。	
TROFF	（書式） TROFF
（機能） トレース機能を解除します。	

getchar

（書式） int getchar( );　**（戻り値）** 読み込んだ文字のコード。int型。

（機能）
キーボードから1文字読み込みます。
getcharは、getc(stdin)と同じです。
入力は、EXEキーを押すと行なわれます。
SHIFT EXEと押すと、EOF(−1)が入力されます。

（例）
```
int c;
  ⋮
c=getchar ( ) ;
```

getc

（書式） int getc(stdin) ;
extern FILE *stdin;
（戻り値） 読み込んだ文字のコード。int型。

（機能）
読み込んだ1文字のコードを返します。
入力は、EXEキーを押すと行なわれます。
動作は、getcharと同じです。
stdin(キーボード)以外からの入力を指定することはできません。

（例）
```
extern FILE *stdin;
int c;
  ⋮
c=getc(stdin);
```

fgetc

（書式） int fgetc(stdin);
extern FILE *stdin;
（戻り値） 読み込んだ文字のコード。int型。

（機能）
キーボードから1文字読み込みます。
入力は、EXEキーを押すと行なわれます。
動作はgetcharと同じです。
stdin(キーボード)以外からの入力は指定できません。

（例）
```
extern FILE *stdin;
int c;
  ⋮
c=fgetc (stdin) ;
```

putchar

（書式） int putchar(c);
int c;
（戻り値） 出力した文字のコード。int型。

（機能）
・stdout(表示画面に)1文字出力します。

（例）
```
main ( )
{
  char buffer [64];
  int i, c;
  gets(buffer);
  for(i=0; buffer[i]!='¥0'; i++){
      c=putchar (buffer[i]);
      if(c==EOF)break;
  }
}
```

putc

（書式） int putc (c, stdout);
int putc (c, stdprn);
int c;
extern FILE ＊stdout, ＊stdprn;

（戻り値） 出力した文字のコード。int型。

（機能）
stdout（表示画面）またはstdprn（プリンタ）に１文字出力します。

fputc

（書式） int fputc (c, stdout);
int fputc (c, stdprn);
int c;
extern FILE ＊stdout, ＊stdprn;

（戻り値） 出力した文字のコード。int型。

（機能）
stdout（表示画面）またはstdprn（プリンタ）に１文字出力します。
動作は、putcと同じです。

（例）

```
extern FILE ＊stdout;
char buffer[30];
int c;
    ⋮
c=fputc (buffer[0], stdout);
```

gets

（書式） char ＊gets(string);
char ＊string;

（戻り値） 格納されたデータのポインタ。char型。

（機能）
stdin（キーボード）から１行読み込み、それをstringに格納します。
文字列は、改行文字まで読み込まれます。
改行文字は、string中では「¥0」(NULL)文字に置き換えられます。

（例）

```
char string[30], ＊result;
    ⋮
result=gets (string);
```

fgets

（書式） char ＊fgets(string, count, stdin);
char ＊string;
int count;
extern FILE ＊stdin;

（戻り値） 格納されたデータのポインタ。
char型。

（機能）

stdin（キーボード）から文字列を読み、それをstringに格納します。

文字は、改行文字または読み込んだ文字数がcount-1になるまで読まれます。文字列の最後に「¥0」(NULL)文字が付加されます。

改行文字が読まれた場合は、string中で「¥0」(NULL)文字に置き換えられます。

（例）

```
entern FILE ＊stdin;
char string[30], ＊result ;
        ⋮
result＝fgets(string, 30, stdin);
```

puts

（書式） int puts(string);
char ＊string;

（戻り値） 改行文字コード。
int型。

（機能）

stdout（表示画面）にstringを出力します。

文字列の終了を表わす「¥0」(NULL)文字は、改行文字に置き換えて書き込みます。

（例）

```
int result;
        ⋮
result＝puts("string");
```

fputs

（書式） int fputs(string, stdout);
int fputs(string, stdprn);
char ＊string;
extern FILE ＊stdout, ＊stdprn;

（戻り値） 書き込んだ最後の文字。
int型。

（機能）

stdout（表示画面）またはstdprn（プリンタ）にstringを出力します。

文字列の終了を表わす「¥0」(NULL)文字を書き込みません。

（例）

```
extern FILE ＊stdprn;
int result;
        ⋮
result＝fputs("string", stdprn);
```

printf
fprintf
sprintf

(書式)

```
printf (format [, argument……]);
int fprintf (stdout, format [, argument……]);
int fprintf (stdprn, format [, argument……]);
int sprintf (buffer, format [, argument……]);
char *format;
char *buffer;
extern FILE *stdout, *stdprn;
```

(戻り値) 出力した文字数。int型。(sprintf での最後の「¥0」(NULL)文字は数えません)
エラーなら、EOF。

(機能)

argumentをformatにしたがって変換し、printfはstd out(表示画面)に、fprintfはstd-out(表示画面)またはstdprn(プリンタ)に、sprintfはbufferに、それぞれ出力します。

sprintfの場合だけ、最後に「¥0」(NULL)文字を出力します。

formatは、0個以上の文字列で、普通の文字、エスケープシーケンス、変換仕様からなります。普通の文字とエスケープシーケンスは、現われる順にそのまま出力されます。

argumentが変換仕様よりも多いときは、余分なargumentは無視され、少ないときは結果が不定となります。

(例1)

```
int count;
count=234;
sprintf ("%d%06d%X%x%o¥n",
  count, count, count, count, count);
```

出力結果　234 000234EA ea 352

(例2)

```
int count;
count=234;
printf ("¦%d¦%6d¦%-6d ¥n",
  count, count, count);
```

出力結果　¦234¦　　234¦234　　¦

scanf
fscanf
sscanf

(書式)

```
int scanf (format [, argument……]);
int fscanf (stdin, format [, argument……]);
int sscanf (buffer, format [, argument……]);
char *format;
char *buffer;
extern FILE *stdin;
```

(戻り値) 代入されたargumentの個数(0もあり得ます)。int型。

(機能)

入力したデータをformatにしたがって変換し、argumentに代入します。

scanfとfscanfはstdin(キーボード)から、sscanfはbufferから入力します。

argumentは、formatで指定された型に対応する型の変数を指すポインタです。

scanfとfscanfは、EXEキーを押すと入力されます。sscanfは、「¥0」(NULL)文字がbufferの終わりと見なされます。

(例)

```
int i;
float f;
doubled;
acanf ("%d%f%1f", & i, & f, & d);
```

1 2 3 EXE − 1 · 2 3 E
1 0 EXE 2 0 3 と入力すると、iに123、fに−1.23e10、dに203.0が代入されます。

fflush	**（書式）** int fflush(stdin); extern FILE ＊stdin;	**（戻り値）** 正常ならば、0。int型。
（機能） バッファの内容をクリアーします。	**（例）** extern FILE ＊stdin; int c; ⋮ c＝getc (stdin); fflush (stdin);	
getch	**（書式）** int getch( );	**（戻り値）** 読み込んだ1文字のコード。int型。
（機能） stdin(キーボード)から直接1文字読み込み、そのキャラクターコードを返します。読み込んだ値は表示されません。 入力バッファが空の場合は、カーソルが点滅して待機します。	**（例）** int x; ⋮ x＝getch( );	
inport	**（書式）** int inport(n); int n;	**（戻り値）** 読み込んだ値。int型。
（機能） nで指定されたポートから1バイト読み込みます。 nは、0から7の範囲です。		
outport	**（書式）** void outport(n, i); int n, i;	**（戻り値）** 何も返しません。
（機能） nで指定されたポートにiを出力します。 nは0から7の範囲、iは0～255の範囲です。		
clearerr	**（書式）** void clearerr(stdin); extern FILE ＊stdin;	**（戻り値）** 何も返しません。
（機能） 入力バッファ内のEOFをクリアーします。		

breakpt

（書式） void breakpt ( );　　**（戻り値）** 何も返しません。

（機能）
プログラムの実行を停止し、ブレークモードに入ります。

（例）

```
  ⋮
breakpt ( );
  ⋮
```

コラム●ブレークモード●

breakpt関数が実行されたとき、またはトレース中にBRKキーが押されると、ブレークモードに入り「Break?」というメッセージが表示されます。

ブレークモードでは次のようなキー操作を行ないます。

キー	機　　能
A	実行を終了
C	実行を再開
EXE	実行を再開
T	トレースしながら実行を再開
N	トレースしないで実行を再開
D	変数名を入力すると変数の型とブレークしたときの変数の値を表示 （もう一度BRKキーを押すと変数表示から抜け出す）

exit

（書式） void exit ( );　　**（戻り値）** 何も返しません。

（機能）
プログラムを正常終了させます。正常終了させる前に出力バッファの内容をクリアーします。

abort

（書式） void abort ( );　　**（戻り値）** 何も返しません。

（機能）
プログラムを異常終了させます。このとき、「Abort」というメッセージをstdout（表示画面）に出力します。

malloc

（書式） char ＊malloc (size);
unsigned size;

（戻り値） 確保されたメモリー領域のポインタ。char型。確保できなかった場合は「¥0」(NULL)を返します。

（機能）

sizeで指定した大きさのメモリー領域を確保します(単位はバイトです。)

確保されたメモリー領域は、プログラムの実行終了とともに解放されます。

（例）

```
main ( )
{
char ＊c ;
   ⋮
if ((c＝malloc(256))＝＝NULL){
  printf (“データリョウイキガトレマセン¥n”);
  exit ( );
  }
   ⋮
}
```

calloc

（書式） char ＊calloc (n, size);
unsigned n;
unsigned size;

（戻り値） 確保したメモリー領域のポインタ。char型。確保できなかった場合は「¥0」(NULL)を返します。

（機能）

sizeで指定した大きさ（バイト）のn個の要素の配列をメモリー領域に確保し、0で初期化します。

確保されたメモリー領域は、プログラムの実行終了とともに解放されます。

（例）

```
main ( )
{
int ＊iarry, i;
   ⋮
it ((iarry＝(int＊)calloc (1000,2))＝＝
  NULL){
  printf("ハイレツガトレマセン¥n");
  exit ( );
  }
for (i＝0; i<1000;i＋＋) iarry [i]＝0;
   ⋮
}
```

free	(書式) int free (ptr); char ＊ptr;	(戻り値) 解放されると0。int型。ptrが無効だと(calloc、mallocによって確保されたメモリー領域のポインタでないと)、−1を返します。
(機能) mallocやcallocで確保されたメモリー領域を解放します。 ptrで、calloc、mallocによって確保されたメモリー領域のポインタを指定します。		(例) char ＊arry; ⋮ arry＝malloc(256); ⋮ free (arry);

文字列関数

strlen	(書式) int strlen (string); char ＊string;	(戻り値) 文字列stringの「¥0」(NULL)を含まない長さ。int型。
(機能) 文字列stringの「¥0」(NULL)文字の直前までのバイト数を返します。		(例) int length; ⋮ length＝strlen ("adc"); /＊length＝3＊/
strcpy	(書式) char ＊strcpy (dest, source); char ＊dest, ＊source;	(戻り値) destのポインタ。 char型。
(機能) 文字列sourceの先頭から「¥0」(NULL)文字まで(「¥0」(NULL)を含む)の範囲を、文字列destの後ろにコピーします。 コピーするとき、オーバーフローのチェックは行ないません。		(例) char ＊result, string [64]; ⋮ result＝strcpy (string, "abc"); /＊string＝"abc"＊/

strcat

(書式)

```
char strcat (dest, source);
char *dest, *source;
```

(戻り値) destのポインタ。
char型。

(機能)

文字列sourceの先頭から「¥0」(NULL)文字の直前までの範囲を文字列destの最初の「¥0」(NULL)文字の後ろに付加し、最後に「¥0」(NULL)文字を付加します。

付加のとき、オーバーフローチェックは行ないません。

strcmp

(書式)

```
int strcmp (string1, string2);
char *string1, *string2;
```

(戻り値) 次の整数値を返します。
int型。
string1<string2→0より小さい整数
string1=string2→0
string1>string2→0より大きい整数

(機能)

文字列string1と文字列string2の先頭から一文字ずつ、「¥0」(NULL)文字が現われるまで比較(ASCIIコード順)します。

「¥0」(NULL)も比較の対象となります。

(例)

```
int result;
char string1[5];
char string2[5];
    :
strcpy (string1, "abcde");
strcpy (string2, "abcda");
result=strcmp (stringl, string2 );
                    /*result=1*/
```

strchr

(書式)

```
char *strchr (string, chr);
char *string;
int chr;
```

(戻り値)

検索したchr(指定文字)のポインタ。char型。
見つからなかったときは「¥0」(NULL)を返します。

(機能)

文字列stringで最初に現われる指定文字chrを検索します。
「¥0」(NULL)文字も検索の対象となります。

(例)

```
main ()
{
    char instr[64], *ss;
    printf ("Input string=");
    gets (instr);
    ss=instr;
    while ((ss=strchr (ss,'*'))!=NULL)
                        /*アスタリスク ノ
                          ケンサク*/
      *ss='_';   /*アンダーバー ニ
                    オキカエ*/
    puts (instr);
}
```

abs

（書式） int abs (n);
int n;

（戻り値） nの絶対値。int型。

（機能）

整数の絶対値を返します。

（例）

```
main ( )
{
  int i, ans;
  i=−1;
      ⋮
    ans=abs (i);
      ⋮
}
```

sin cos tan

（書式） double sin (x);
double cos (x);
double tan (x);
double x;

（戻り値） double型。

（機能）

角度xに対する三角関数の値を返します。

角度xが演算範囲を超えている場合は、エラーになります。

角度xの演算範囲は、次の通りです。

|x| ＜1440 (DEG)
|x| ＜8π (RAD)
|x| ＜1600 (GRA)

（例）

```
main ( )
{
  double y;
  angle (0);
  for (;;) {
    printf ("Angle?");
    scanf ("%lf",&y);
    printf ("%11.10g%11.10g%11.10g
  ¥n", sin(y), cos(y), tan(y));
  }
}
```

asin / acos / atan

（書式） double asin (x);
double acos (x);
double atan (x);
double x;

（戻り値） double型。

（機能）

xに対する逆三角関数の値（角度）を返します。

xが演算範囲を超えている場合には、エラーになります。

xの演算範囲は次の通りです。

$-1 \leqq x \leqq 1$(asin, acosの場合)

$|x| < 100^{100}$(atanの場合)

返される関数の値の範囲は次の通りです。(RADの場合)。

$[-\pi/2, \pi/2]$(asin)

$[0, \pi]$(acos)

$[-\pi/2, \pi/2]$(atan)

（例）

```
double y;
  ⋮
angle (1);
y=asin(1.0);
y=acos(1.0);
y=atan(1.0);
```

sinh / cosh / tanh

（書式） double sinh (x);
double cosh (x);
double tanh (x);
double x;

（戻り値） double型。

（機能）

角度xに対する双曲線関数の値を返します。

$\sinh x = (e^x - e^{-x})/2$

$\cosh x = (e^x + e^{-x})/2$

$\tanh x = (e^x - e^{-x})/(e^x + e^{-x})$

xが演算範囲を超えている場合は、エラーになります。

xの演算範囲は、次の通りです。

$|x| \leqq 230.2585092$(sinh, coshhの場合)

$|x| < 100^{100}$, $-1 \leqq |\tan(x)| < 1$

(tanhの場合)

（例）

```
double y;
  ⋮
angle (0);
y=sinh(1.0);
y=cosh(1.0);
y=tanh(1.0);
```

asinh / acosh / atanh

（書式） double asinh(x);
double acosh(x);
double atanh (x);
double x;

（戻り値） double型。

（機能）

xに対する逆双曲線関数の値を返します。

$\sinh^{-1} x = \log_e(x+\sqrt{x^2+1})$

$\cosh^{-1} x = \log_e(x+\sqrt{x^2-1})$

$\tanh^{-1} x = \log_e(1+x/1-x)/2$

xが演算範囲を超えている場合は、エラーになります。

xの演算範囲は次の通りです。

$|x| < 5E+99$ (asinh)

$1 \leqq x < 5E+99$ (acosh)

$-1 < x < 1$ (atanh)

（例）

```
double y;
    :
angle(0);
y=asinh(1.0);
y=acosh(2.0);
y=atanh(0.5);
```

pow

（書式） double pow(x,y);
double x, y;

（戻り値） double型。

（機能）

xのy乗(X^Y)の値を返します。

yが0の場合は、1を返します。

xが0でyが負の場合と、xが負でyが整数でない場合は、エラーになります。

結果がオーバーフローの場合も、エラーになります。

（例）

```
double x, y, z;
x=2.0;  y=3.0;
    :
z=pow(x, y);
```

sqrt

（書式） double sqrt(x);
double x;

（戻り値） double型。

（機能）

xの平方根($\sqrt{x}$)を返します。

xが負の場合は、エラーになります。

（例）

```
double y;
    :
y=sqrt(2.0);
```

exp	**（書式）** double exp(x); double x;	**（戻り値）** double型。
（機能） xの指数関数(e^x)の値を返します。 x＞230.2585092の場合、エラーになります。		**（例）** double y; ⋮ y=exp(1.0);
log **log10**	**（書式）** double log(x); double log10(x); double x;	**（戻り値）** double型。

（機能）
logは、xの自然対数($\log_e x$)の値を返します。
log10は、xの常用対数($\log_{10} x$)の値を返します。
xが0の場合、負の場合はエラーになります。
		（例）
		double y;
		⋮
		y=log(1000.0);
		y=log10(1000.0);

angle	**（書式）** void angle(n); unsigned n;	**（戻り値）** 何も返しません。

（機能）
三角関数、逆三角関数の角度モードを指定します。nの指定により次のようになります。
0:DEG （度）
1:RAD （ラジアン）
2:GRA （グラッド）
		（例）
		double y;
		⋮
		angle(0);
		y=sin(90.0);
		angle(1);
		y=sin(1.570796327);
		angle(2);
		y=sin(100.0);

beep	**（書式）** void beep(n); unsigned n;	**（戻り値）** 何も返しません。
（機能） nの指定により、低い音か高い音を鳴らします。 0:低い音を鳴らします。 1:高い音を鳴らします		**（例）** beep(0);
clrscr	**（書式）** void clrscr(); **（機能）** 表示画面の表示を消去し、カーソルをホームポジションに戻します。	**（戻り値）** 何も返しません。 **（例）** clrscr();
gotoxy	**（書式）** void gotoxy(x, y); unsigned x; unsigned y;	**（戻り値）** 何も返しません。
（機能） 仮想スクリーン(32桁×8行)上のカーソル位置を、xとyで指定します。 座標は、仮想スクリーン上の左上隅を原点(0, 0)とし、次の範囲で指定できます。 0≦x≦31 0≦y≦7		**（例）** gotoxy(10, 2);

V. F:COM

F.COM Begin

F.COMメニュー

操作 SHIFT F.COM

```
BASICプログラムファイルエリア→ P ■ 1 2 3 4 5 6 7 8 9    [RS232C] ←入出力できるデバイス
アスキー形式のファイルエリア→ F * 1 * * 4 5 6 7 8 9      2948B ←メモリーの残り容量
対象ファイル→ P0>Save / Load / Merge / Copy
               Edit / New  / Print / Device
```

コマンドメニュー

over RS232

```
BASICプログラムファイルエリア→ P ■ 1 2 3 4 5 6 7 8 9    [RS232C] ←対象デバイス
アスキー形式のファイルエリア→ F * 1 * * 4 5 6 7 8 9      2948B
対象ファイル→ P0>RS2323C / MT / Disk / Switch ←デバイス
```

ここで次のキーを押して、デバイスを切り替えます。

R キー　RS-232C

M キー　カセットテープレコーダー

D キー　フロッピーディスクドライブ

また、S キーを押すと通信条件の設定が行なえます。

操作		表示	
①	SHIFT F.COM	①	P 0 1 2 3 4 5 6 7 8 9 [RS232C] F 0 1 ■ 3 4 5 6 7 8 9 3343B F2>Save / Load / Merge / Copy Edit / New / Print / Device
②	⇧ ⇩ ⇦ ⇨ ←(キーでコピーしたいファイルの記憶されているファイルエリアを反転表示させます。)	②	
操作		表示	
③	C	③	P 0 1 2 3 4 5 6 7 8 9 [RS232C] F 0 1 ■ 3 4 5 6 7 8 9 3343B >Copy F2 to [??]
操作		表示	
④	⇨ ⇨ ⇨ ⇨ ⇨ ⇨ ⇨ ↑(コピーする先のファイルエリアを反転表示させます。)	④	P 0 1 2 3 4 5 6 7 8 9 [RS232C] F 0 1 * 3 4 5 6 7 8 ■ 3343B >Copy F2 to [F9]
操作		表示	
⑤	EXE	⑤	P 0 1 2 3 4 5 6 7 8 9 [RS232C] F 0 1 ■ 3 4 5 6 7 8 * 3331B F2>Save / Load / Merge / Copy Edit / New / Print / Device

```
BPS [ 300] Parity [E] Data    [8]
Stop    [1] CTS   [OFF] DSR   [OFF]
CD    [OFF] Busy [ ON] SI/SO[OFF]
End   [ ON] MTphase[0] MTspeed[F]
```

```
P ■ 1 2 3 4 5 6 7 8 9    [RS232C]
F 0 * * 3 4 5 6 7 8 9      3334B
P0>SAVE"COM0:2,E,8,1,N,N,N,B,N_
```

RS-232C関係

BPS(ボーレート指定)

次のボーレートが指定できます。

150	300	600	1200	2400	4800

Parity(パリティビットの状態指定)

N	E	O
パリティなし	偶数パリティ	奇数パリティ

Data(キャラクタのビット長の指定)

7	8
JIS7ビット	JIS8ビット

Stop(ストップビット長の指定)

1	2
ストップビット＝1ビット	ストップビット＝2ビット

CTS(CTS信号の状態で制御するかどうかの指定)

ON	OFF
制御する	無視する

DSR(DSR信号の状態で制御するかどうかの指定)

ON	OFF
制御する	無視する

CD(CD信号の状態で制御するかどうかの指定)

ON	OFF
制御する	無視する

Busy(バッファビジーの制御があるかないかの指定)

ON	OFF
制御する	制御しない

SI/SO(SI/SO制御をするかしないかの指定)

ON	OFF
制御する	制御しない

End(エンドコード1Aの設定)

ON	OFF
設定する	設定しない

カセットテープレコーダー関係

MTphase(MTからの読み込むときの位相の指定)

0	1
正相	逆相

MTspeed(転送速度の指定)

S	F
300BPS	1200BPS

Save to (F)

```
P ■ 1 2 3 4 5 6 7 8 9          [MT]
F 0 * * 3 4 5 6 7 8 9        3334B
P0>SAVE"CAS0:(F)TEST1_
```

Merge Files

```
P * 1 2 3 4 5 6 7 8 9          [Disk]
F 0 * ■ 3 4 5 6 7 8 9        2283B
F2>MERGE"0:PR01_
```

VI. STAT

STAT Begin

操作		表示	
①	[Fx]	①	〈 Fx menu 〉 1:STAT(x) 2:STAT(x,y) 3:Trainig Board

操作		表示	
①	[1]	①	〈 Statistics [x] 〉 Input / Delete / Clear / List Print / T-score / Frequency

操作		表示	
①	[2]	①	〈 Statistics [x,y] 〉 Input / Delete / Clear / List Print / eoX / eoY / Frequency

Select Modi

キー操作	表　示	機　能
[I]	Input	データの入力
[D]	Delete	データの削除
[C]	Clear	全データの消去、統計量の初期化
[L]	List	各統計量の表示（結果の表示）
[X]	eoX	「y」に対する「x」の推定値計算
[Y]	eoY	「x」に対する「y」の推定値計算
[P]	Print	統計量のプリンタ出力
[F]	Frequency	度数入力切り替え

操作		表示	
①	[Fx][1][C]	①	〈 Statistics [x] 〉
②	[Y]または[EXE]		Clear data (Y/N) ?

Input Data

操作		表示	
①	[I]	①	Input data (x) [EXE]:menu CNT= 0 Freq :off x?_

操作		表示	
①	[1][0][EXE]	①	Input data (x) [EXE]:menu CNT= 0 Freq : on x?10 f?1

List Data

操作		表示	
①	Fx 1 L	①	CNT :n = 10 SUMX :Σx = 55 SUMX2 :Σx² = 385 MEANX :Σx/n = 5.5

Delete Data

操作		表示	
①	Fx 2 F	①	(Statistics [x,y:f]) Input / Delete / Clear / List Print / eoX / eoY / Frequency

操作		表示	
①	D	①	Delete data (x,y) [EXE]:menu CNT=1 Freq :on X?_ :Y? f?

Extimation of x

操作		表示	
①	Fx 2 X	①	(Statistics [x,y]) COR= 0.8572508573 (y=a+bx) Estimation of x [EXE]:menu y?_

Training Board

```
Training Board
```

VII. HD61700 Cross Assembler

Table of Contents

List of Pseudo Instructions

Pseudo Instructions				
ORG (Origin),	START・・・ (Start),	EQU・・・ (Equivalent),	DB・・・(Define Byte),	DW・・・(Define Word)
DS・・・(Define Size),	LEVEL ... (Level),	#IF・・・#ELSE --- #ENDIF	#INCLUDE	#INCBIN・・・(INClude BINary)
#NOLIST, #LIST, #EJECT	#KC, #AI, #EU			

List of Registers

General-Purpose 8-bit Register				
$ 0, $ 1, ・・・, $ 31・・・(Main Registers)				
16-bit Register				
PC・・・ (Program Counter)	SSP (Syatem Stack Pointer)	USP (User Stack Pointer)	IX, IY, IZ (Index Registers)	
Specific Index Register and flag Register				
SX, SY, SZ (Specific Index Registers)	F・・・・(Flag Registers)			
Status Register				
IE・・・ (Interrupt Enable Register)	IA (Interrupt Select and Key Output Register)	UA (High-Order Address Specification Register)	*No mnemonic* (Display Driver Control Register)	PE・・・(Port Data Direction Register)
PD・・・(Port Data Register)	TM・・・(Timer Data Register)	IB (Interrupt Control and Memory Bank Range Configuration Register)	KY・・・(Key Input Register)	

List of Mnemonics

Transfer Instruction (8 bits)				
LD・・・(Load),	LDI・・・(Load Increment),	LDD・・・(Load Decrement),	LDC (Load Check),	ST・・・(Store)
STI・・・(Store Increment),	STD・・・(Store Decrement),	PPS・・・(Pop by System stack pointer),	PPU・・・(Pop by User stack pointer),	PHS・・・(Push by System stack pointer)
PHU・・・(Push by User stack pointer),	GFL・・・(Get Flag),	PFL (Put Flag),	GPO・・・(Get Port),	GST・・・(Get Status)
PST・・・・(Put Status),	STL (Store data to LCD),	LDL (Load data from LCD),	PPO・・・(Put LCD control Port),	PSR (Put Specific index Register)
GSR・・・(Get Specific index Register)				
Transfer Instruction (16 bits)				
LDW・・・(Load Word),	LDIW・・・(Load Increment Word),	LDDW・・・(Load Decrement Word),	LDCW・・・(Load Check Word),	STW・・・(Store Word)
STIW・・・(Store Increment Word),	STDW・・・(Store Decrement Word),	PPSW・・・(Pop by System stack pointer Word),	PPUW・・・(Pop by User stack pointer Word),	PHSW・・・(Push by System stack pointer Word)
PHUW・・・(Push by User stack pointer Word),	GRE・・・(Get Register),	PRE (Put Register),	STLW・・・(Store Word data to LCD),	LDLW・・・(Load Word data from LCD)
PPOW・・・(Put LCD control Port Word),	GFLW・・・(Get Flag Word),	GPOW・・・(Get Port Word),	PSRW (Put Specific index Register Word),	GSRW・・・(Get Specific index Register Word)
Arithmetic Instructions (8 bits)				
INV ... (Invert),	CMP (Complement),	AD・・・(Add),	SB (Subtract),	ADB・・・(Add BCD)
SBB (Subtract BCD),	ADC ... (Add Check),	SBC (Subtract Check),	AN・・・(And),	ANC (And Check)
NA・・・(Nand),	NAC・・・(Nand Check),	OR・・・(Or),	ORC・・・(Or Check),	XR・・・(Exclusive Or)
XRC・・・(Exclusive Or Check)				
Arithmetic Instructions (16 bits)				
INVW・・・(Invert Word),	CMPW・・・(Complement Word),	ADW・・・(Add Word),	SBW・・・(Subtract Word),	ADBW・・・((Add BCD Word)

SBBW・・・(Subtract BCD Word),	ADCW・・・(Add Check Word),	SBCW・・・(Subtract Check Word),	ANW・・・(And Word),	ANCW・・・・(And Check Word)
NAW・・・(Nand Word),	NACW・・・(Nand Check Word),	ORW・・・(Or Word),	ORCW・・・(Or Check Word),	XRW・・・(Exclusive Or Word)
XRCW・・・(Exclusive Or Check Word)				
Rotate shift Instruction (8 bits)				
ROU・・・・(Rotate Up),	ROD・・・(Rotate Down),	BIU・・・(Bit Up),	BID・・・(Bit Down),	DIU (Digit Up)
DID・・・(Digit Down),	BYU・・・(Byte Up),	BYD・・・(Byte Down)		
Rotate shift Instruction (16 bits)				
ROUW・・・(Rotate Up Word),	RODW・・・(Rotate Down Word),	BIUW・・・(Bit Up Word),	BIDW・・・(Bit Down Word),	DIUW・・・(Digit Up Word)
DIDW・・・(Digit Down Word),	BYUW・・・(Byte Up Word),	BYDW・・・・(Byte Down Word)		
Jump / Call Instructions				
JP・・・(Jump),	JR・・・(Relative Jump),	CAL・・・(Call),	RTN・・・・(Return)	
Block Transfer / Search Instructions				
BUP ... (Block Up),	BDN・・・(Block Down),	SUP (Search Up),	SDN (Search Down),	BUPS・・・(Block Up & Search)
BDNS・・・(Block Down & Search)				
Special Instructions				
NOP・・・(No Operation),	CLT・・・(Clear Time),	FST・・・(Fast mode),	SLW・・・(Slow mode),	OFF・・・(Off)
TRP・・・(Trap),	CANI・・・・(Cancel Interrupt),	RTNI・・・(Return from Interrupt)		
Multibyte Transfer Instruction (2 to 8 bytes) not Disclosed				
LDM ... (Load Multi byte),	LDIM・・・(Load Increment Multi byte),	LDDM・・・(Load Decrement Multi byte),	LDCM・・・(Load Check Multi byte),	STM・・・(Store Multi byte memory)
STIM (Store Increment Multi byte),	STDM・・・(Store Decrement Multi byte),	PPSM・・・(Pop by Syatem stack pointer Multi byte),	PPUM・・・(Pop by User stack	PHSM・・・(Push System stack

			pointer Multi byte),	pointer Multi byte)
PHUM・・・(Push User stack pointer Multi byte),	STLM (Store LCD data port Multi byte),	LDLM (Load LCD data port Multi byte),	PPOM・・・(Put LCD control port Multi byte),	PSRM (Put Specific index Register Multi byte)
Multibyte Arithmetic Instruction (2 to 8 bytes) not Disclosed				
INVM・・・(Invert Multi byte),	CMPM (Complement Multi byte),	ADBM・・・(Add BCD Multi byte),	ADBCM・・・((Add BCD Check Multi byte),	SBBM・・・(Subtract BCD Multi byte)
SBBCM・・・(Subtract BCD Check Multi byte),	ANM・・・(And Multi byte),	ANCM・・・(And Check Multi byte),	NAM・・・(Nand Multi byte),	NACM・・・(Nand Check multi byte)
ORM (Or Multi byte),	ORCM・・・(Or Check Multi byte),	XRM・・・(Exclusive Or Multi byte)	XRCM・・・(Exclusive Or Check Multi byte)	
Multi-byte Shift Instruction (2 to 8 bytes) not Disclosed				
DIUM・・・(Digit Up Multi byte)	DIDM ... (Digit Down Multi byte),	BYUM・・・(Byte Up Multi byte),	BYDM・・・・(Byte Down Multi byte)	

7-1 HD61700 Cross Assembler

HD61700 Cross assembler HD61 was developed by Ao. It is almost the same as the assembler built in PB-1000 (upward compatibility), but the differences are as follows.

1. Label length is up to 16 characters and can be registered as long as memory allows. The code area can be secured up to 64KB.
2. Not only address labels (for JR, JP, CAL instructions) but also numeric labels can be used with transfer instructions.
3. Supports almost all orders of HD61700, including unreleased CASIO. The mnemonic can use both "AI-assembler format" and "KC format". (Mixing is also possible) From Rev 0.41, it also supports mnemonics in EU format (Europe notation), and by #EU (or / eu) specification. Switchable from AI / KC format to EU (Europe) format.
4. Second operation extension ($ 0, $ 30, $ 31, LD & JR) etc. can be specified by default. (OFF when the / n option is specified)
5. The output format supports BASIC DATA statement format and PBF format (PBF format specifies / p option).
6. Output a formatted list file (.lst).
7. Supports pseudo-instructions (DW, LEVEL, #if, #else, #endif, #include, etc.) not supported by PB-1000.

Assembling Method

HD61 is available in Windows and DOS versions, but execute the following command at each command prompt.

HD61 [source file name] (option [/ n] [/ p] [/ q] [/ w] [/ tab] [/ r] [/ o filename] [(/ set) *symbol* = *value*] [/ eu])

When executed, the specified file is assembled according to the option settings as shown in the example below.

Assemble example

```
> hd61 hd61700.s [Enter]
HD61700 ASSEMBLER Rev 0.41
Input: hd61700.s
PASS 1 END
PASS 2 END
ASSEMBLY COMPLETE, NO ERRORS FOUND
```

If normal, displays [ASSEMBLY COMPLETE, NO ERRORS FOUND] and exits. At this time, a .bas file and an .lst file are generated. If any error occurs during assembly, display an error line and exit. After Rev.0.09, when the source file name is 8 characters or more, a warning is displayed (the assembly works normally). This means that the file name output to BAS (or PBF) will be a long file name in consideration of use with models that support 8.3 file names such as PB-1000 / C and AI-1000. Warning. (The function to automatically shorten the file name is not implemented)

Assembler Options

Although it can be omitted, the following options can be specified during assembly.

List of Assembly Options

Option	Function
/ p	Output in PBF format. (Default is output in BASIC DATA statement format)
/ q	Output in QL (quick loader) format.
/ n	Turn off optimization by specifying the second operation (default is ON)
/ w	Assemble for 16-bit addressing. (Optimization is fixed at LEVEL 0) Outputs the assembly code corresponding to the 16-bit address for the internal ROM.
/ tab	Output the list file with TAB = 8.
/ r	Output relocate information file (* .roc). Outputs information file for creating relocate format file used in FBF / VX-MENU. Used when creating RR format and * .o / *. O2 format files.
/ o [filename]	Specify the file name to be output to the PBF / BAS format file header. Default is not specified (automatic generation). > / TD>
(/ set) [symbol label name] = [value / label name]	Define arbitrary symbol labels. / set can be omitted.
/EU	Set to assemble EU format (Europe format) mnemonics. Even if pseudo instruction #EU is specified in the source, the same operation is performed.

For the / p option, refer to 1-3-2.PBF format in 1-3. Executing the created program.

The / n option disables code optimization of transfer instructions for $ 0, $ 30, and $ 31, and outputs code compatible with the PB-1000 built-in assembler.

Output code Example with / n Option

Option Setting	Mnemonic	Output Code	Remarks
No / n option (default)	LD $ 2, $ 30	02 42	When 2nd operation specification is ON = 2 byte instruction is output
with / n option	LD $ 2, $ 30	02 62 30	When the second operation specification is OFF = 3-byte instruction is output

This is used when assembling a source that determines the address of the data area for the PB-1000, or when assembling a program that changes the SIR using the PSR instruction. For details on the instructions to be optimized and the output code, refer to 4. lst file output by assembling the HD61700.s file attached to mnemonic or HD61

The / set option can be used to define arbitrary symbol labels at startup since Rev 0.23. This is done using the # if ~ # else ~ # endif pseudo-instructions,

- When switching the assembly code for each model,
- When switching the assembly start address according to memory capacity

The symbol label value can be changed without modifying the source file. By using a batch file, output results for each model can be obtained automatically. **If the same label name is EQU declared in the source, the value defined in / set takes precedence, so the definition in the source functions as the default value.**
Format example) Specify the model name and start address from the command line.

HD61 SAMPLE.S / SET MODEL = PB1000 / SET BASE = 0x7000

Since Rev 0.28, you can omit the / set option and define any symbol with the description [symbol label name] = [value / label name]. The following format example is exactly the same as the above format example (no omission of / set) in terms of operation specifications.
Format example) Specify the model name and start address from the command line.

HD61 SAMPLE.S MODEL = PB1000 BASE = 0x7000

Execution of Output Format and Machine Language

The HD61 outputs one of the BAS, PBF, and QL format files as an option specified during assembly. For each type of file, the machine language can be executed by placing the machine language in the memory on the pocket computer according to the following procedure.
In the following sections, loading and execution of each type of file into memory will be explained, focusing on FX-870P / VX-4.

BAS Format

(1) Assemble with HD61. Create a bas file. For the BAS format, see the appendix.
(2) Paste the contents of Trans.b attached to HD61 into the output bas file as a machine language loader program.
For FX-870P and VX-4, leave line number 80 as a comment.
(3) Load the created program file into the pocket computer with F.COM.
(4) If it is loaded to the unused area of the system, nothing is required. Otherwise, in the case of FX-870P and VX-4 , the machine language area is secured by extended CLEAR.
(5) When the loaded program is executed, the machine language code is placed in the memory.
(6) A machine language routine is called with MODE110 (execution address) .

In PB-1000 / C and AI-1000, it is not necessary to comment on line number 80 of Trans.b. In that case, the machine language program is automatically saved by (5).

PBF Format

For the format of the PBF format, see the appendix.
For FX-870P, VX-4 (VX-3 has the same procedure):

(1) Assemble with / p option on HD61. Create a pbf file.
(2) A machine language area is secured on the pocket computer using the same method as the BAS format.
(3) Run TransVX.bas attached to HD61 on the pocket computer. When executed, it stands by in the RS232C reception state.

(4) Transfer the PBF file created in (1) to the pocket computer via RS232C.
(5) The binary code is automatically converted and the machine language code is placed in the memory. When processing is complete, "Completed!" Is displayed.
(6) A machine language routine is called with MODE110 (execution address).

For PB-1000 / C and AI-1000:

(1) Assemble with / p option on HD61. Create a pbf file.
(2) A machine language execution area is secured on the pocket computer.
(3) JUN AMANO's PbfTOBin.bas is executed and the file name is "COM0: 7". (At 9600bps)
(4) Transfer the PBF file created in (1) to the pocket computer via RS232C.
(5) When execution is completed, an EXE (or BIN) file is automatically generated.

QL Format

 Quick loader data format devised by Mr. Ao.
The usage is as follows.

(1) Assemble with / q option on HD61. Create a ql file. For the QL format, see the appendix.
(2) Paste the output ql file to the quick loader described in “QL format” at the end of the book. Add or modify code as appropriate.
(3) Load the created program file into the pocket computer with F.COM.
(4) If it is loaded to the unused area of the system, nothing is required. Otherwise, in the case of FX-870P and VX-4 , the machine language area is secured by extended CLEAR.
(5) When the loaded program is executed, the machine language code is placed in the memory.
(6) A machine language routine is called with MODE110 (execution address) .

Error Message

The error messages displayed during assembly are as follows.

Error Message List	
Error Message	**Error Contents**
Invalid Source File Name.	The source file cannot be opened.
Line Length is Too Long.	The number of characters in one line has been exceeded.
Operand Length is Too Long.	The number of operand characters has been exceeded.
LABEL Length is Too Long.	The number of label characters has exceeded.
ORG Not Entry.	There is no ORG instruction definition.
Operand Not Entry.	No operand description.
EQU without Label.	EQU has no label entry.
Illegal Operand.	Operand description error.
START Already Defined.	There are two or more START statements.
Illegal [,]	The comma description is strange.
Illegal ['] or [(] or [)]	Double coating / parentheses error.
LABEL Already Defined.	There are two or more label descriptions.
LABEL Type Mismatch.	Characters that cannot be used for labels.
Undefined LABEL.	No label registration.
Operation Type Mismatch.	No applicable instruction / Missing description method.
Operand Range Over.	Operand value is out of range.
Jump Address Over.	Relative jump is out of range.
Output Buffer Over Flow.	Output buffer over.
Assemble Address Over Flow.	Assemble address limit exceeded.
Execute Address Illegal.	The execution address is smaller than the first ORG declaration.
Could not calculate.	An operation error (division by 0, etc.) has occurred.
Illegal [#if]-[#endif]	Nesting of # if ~ # else ~ # endif is abnormal.
Invalid Include File Name.	The include file cannot be opened.
Could Not Nest Include.	include nesting error.
Illegal Register Number.	Abnormal main register number.

7-2 MPU Architecture

Features

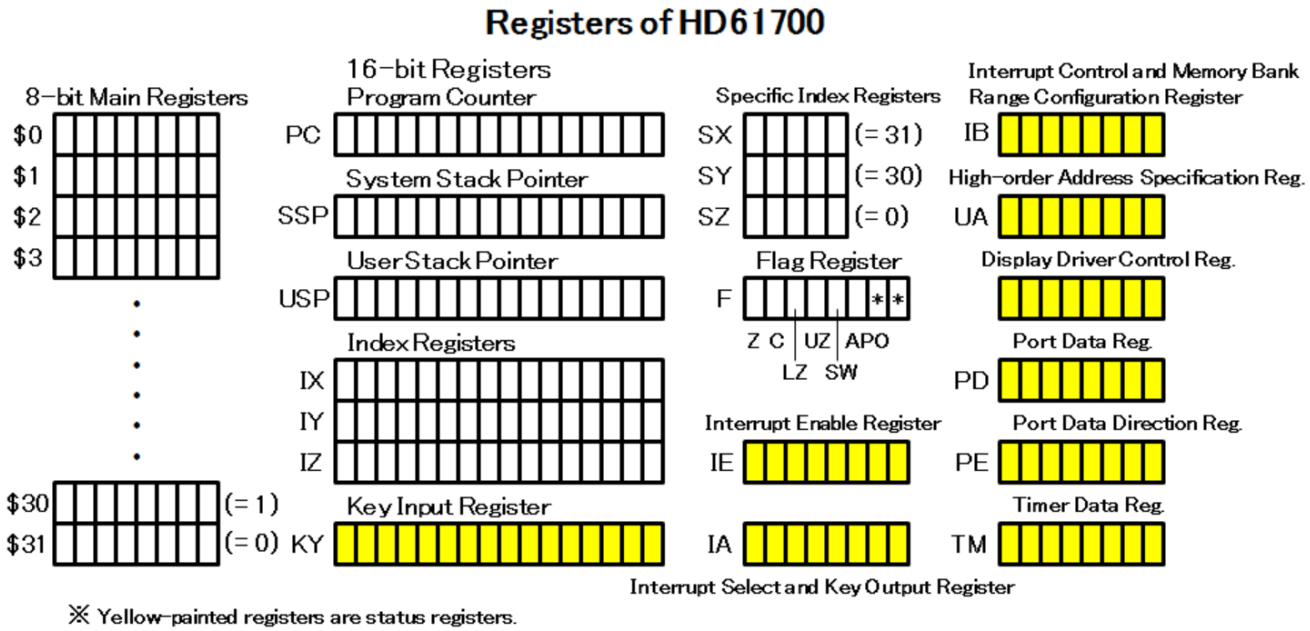
- Decimal calculation possible
- 64KB of 256KB address space (UA / IB register control)
- High-speed processing (LCD display, calculation routine, etc.) with 16bit ROM (3072 words; 64KB?
- Low power consumption (800µA)
- Built-in 32x8RAM as main register. Access in 16bit units is also possible.
  With extension of second / third operation, 8-64bit unit access is also possible.
- Built-in clock function (TM register)
- Key input terminal 12x11 + 1 (access by IA / KY register)
- Interrupt function (3 external terminals, KEY / pulse, ON terminal, 1 minute timer, TRP processing)
- 8-bit I / O port (I / O designation is controlled by PE register)
- LCD display control function (MPU built-in instruction PPO / STL / LDL)

Actually coding, the personal impression is as follows.

- Specific index registers and JR options have been introduced so that no single bit is wasted.
- The JR option is useful for speeding up loops in the algorithm.
- Instructions are arranged so that there is no space in the instruction set, but byte up / down instructions (BYU, BYD, BYUW, BYDW, BYUM, BYDM), NAND instructions (NA, NAC, NAW, NACW, NAM, NACM) Rather than carry addition / subtraction and arithmetic shift instructions, carry was more desirable. BYU and BYD are instructions that simply put 0 in an 8-bit register, and they seem to be completely meaningless just for the purpose of the beauty of the instruction system.
- Since the flag register F is unused 2 bits, I wanted a sign flag (although it was impossible on the instruction set).
- I wanted a change of carry in 4 bit shift instructions (DIU, DID, etc.).
- There were no undefined values in the instruction set, and future extensibility was not considered.

Register Configuration

Has 32 8-bit registers, 6 16-bit registers, and multiple status registers.



1) Main register (8bit)

This is a RAM module built into the HD61700, specified by addresses 0 to 31.
In mnemonics, it is expressed with a "$" mark at the beginning. For example, $ 0, $ 1, ... $ 31. Access and computation up to 64 bits in little endian format. In addition, by using a specific index register SIR (5bit), indirect access in the form of $ SIR is also possible.
* Little endian is a method of arranging & H12345678 in memory and arranging & H78, & H56, & H34, & H12 from the lower address. A typical CPU is x86.

2) Six 16-bit registers

- PC: Program counter (16bit)
- SSP: System stack pointer (16bit)
  For system operations such as CAL, RTN, interrupt processing.
  In addition, direct rewriting with the PRE instruction is possible with PUSH, POP and user programs.
- USP: User stack pointer (16bit)
  Unrestricted stack pointer that can be used freely by user programs.
  Operate with PRE, PHU, PPU.
- IX, IZ, IY: Index register (16 bits: Display format IR) A 16-bit data pointer used for various transfer instructions.
  The IY register can only be used as an end point pointer for block transfer / search instructions.

3) Specific index register and flag register

- SX, SY, SZ: Specific index register (5bit: Display format SIR)
  By defining a specific main register in the SIR in advance using the PSR instruction, the target main register can be transferred / calculated faster (code shortening), and indirect specifications such as $ SX, $ (SX) can be specified. Possible.
  However, in the CASIO HD61700 system, it is assumed that SX = 31 ($ 31 specified), SY = 30 ($ 30 specified), and SZ = 0 ($ 0 specified) are defined at the initial stage and used as they are. When the user changes, the following cautions are required.
  (1) Disable interrupts while changing SIR.
  (2) When returning to the ROM internal processing, when calling the ROM internal processing, return the SIR to the o+riginal setting.
  (3) Coding the optimization switch with OFF (LEVEL 0) specified.
  * In the EU (Europe format), these registers are called short registers (SR) and are labeled # 0, # 1, and # 2, respectively.

- F: Flag register (8bit: Display format F)
  The internal bit configuration is as follows.

MSB							LSB
Z	C	LZ	UZ	SW	APO	*	*

- Explanation of each flag
  1. Z: Zero flag When all bits of the operation result are 0, it is reset to 0, otherwise it is set to 1.
     When Z = 1, it is called NZ: Non-zero.
  2. C: Carry flag This bit is set to 1 when a carry or borrow occurs in the operation result, and 0 otherwise.
     NC: Non-carry.
  3. LZ: Lower digit zero flag If the lower 4 bits of the operation result are 0, it is reset to 0 and 1 otherwise.
     NLZ / LNZ: Non-lower digit zero flag
  4. UZ: Upper digit zero flag If the upper 4 bits of the operation result are all 0, it is reset to 0, otherwise it is set to 1.
     Negative forms such as other operation flags are not prepared as branch conditions.

5. SW: Power switch state flag Notifies the ON / OFF state of the power switch. ON: 1, OFF: 0
6. APO: Auto power off state flag 1 when the OFF command is executed with the power switch turned on. 0 when the power is turned off.

4) Status register

- IE: Interrupt enable register (read / write)
  Specify interrupt mask and conditions (edge / level, etc.) in 8 bits.

  Bit 7 ----- Enable interrupt from / INT1 pin (enabled by 1)
  Bit 6 ----- KEY, pulse interrupt enabled (enabled by 1)
  Bit 5 ----- Enable interrupt from / INT2 pin (enabled by 1)
  Bit 4 ----- 1-minute timer interrupt enabled (1 enables)
  Bit 3 ----- Enable interrupt from / ON pin (enabled by 1)
  Bit 2 ----- Enable interrupts from the Power On switch (enabled by 1)
  Bit 1 ----- / INT1 pin interrupt edge specification (0: falling, 1 rising)
  Bit 0 ----- / INT2 pin interrupt level specification (0: Low level, 1: High level)

- This register is completely cleared by a reset (RESET signal) operation.
  Bits 0, 1, 5, 6, and 7 are cleared by the power OFF / OFF command, but bits 2 to 4 are maintained even when the power is OFF.
  The interrupt priority is as follows in descending order. When a higher priority interrupt occurs, the interrupt is interrupted.

  Priority 1 (IE <7>): Interrupt from INT1 pin
  Priority 2 (IE <6>): KEY / pulse interrupt
  Priority 3 (IE <5>): Interrupt from INT2 pin
  Priority 4 (IE <4>): 1 minute timer interrupt
  Priority 5 (IE <3>): Interrupt from / ON pin
  Priority 6 (IE <2>): Interrupt from Power On switch

- IA: Interrupt Select and Key Output Register (read / write); Interrupt Select and Key Output Register

  Bit 7 ------- KEY interrupt (1), pulse interrupt (0)
  Bit 6 ------- Pulse interrupt signal (0: 256Hz, 1: 32Hz)
  Bit 5 to Bit 4- PIN specification for KEY input (0: No specified PIN, 1: ONE PIN specified, 2: TWO PIN specified, 3: ALL PIN specified)
  Bit 3 to Bit 0- KEY output specification (0 to 12: ONE KEY output, 13: ALL KEY output, 14, 15: undefined)

- * When controlling the key input with the assembler, set 13 (ALL KEY output request) to this register and then use GRE KY, $ C5 to bit OR all keys to $ C5 / $ C5 + 1. The KEY scan code is read.
  When specifying one key at a time, you can get a response according to each key matrix by executing GRE KY, $ C5 after setting 0 to 12.

- UA: Upper address specification register (read / write); High-Order Address Specification Register
  This register determines which bank each (pointer) register will access. The meaning of each bit is as follows.

  Bit 7, 6 ---- IZ register upper address specification (0 to 3)
  Bit 5, 4 ---- IX register / main register upper address specification (0 to 3)
  Bit 3, 2 ---- SSP, USP upper address designation (0 to 3)
  Bit 1, 0 ---- PC upper address specification (0 to 3) *

It is cleared to 0 at RESET and cleared even when the power is turned off except for SSP / USP. The contents of SSP and USP are saved even when the power is turned off.

* Only for the PC upper address specification bits (Bit 0 to Bit 1), there is a delay of one instruction cycle for the result to be reflected after writing the value with the PST instruction. This is because it is necessary to branch (JP / JR) or RTN after specifying the PST instruction for an arbitrary bank. When operating this register, it is necessary to disable interrupts. (In PB-1000 / FX-870P / VX-4 / VX-3 / AI-1000, when the user program is called, the system side is set to disable interrupts. (You may not need to be aware)

- Display driver control register (no write mnemonic)
  Outputs control signals for sending display data and commands to the display driver.

  Bit 7 ----- VDD2
  Bit 6 ----- φ1, φ2 CLOCK ON (1), OFF (0)
  Bit 5 ----- None (undefined)
  Bit 4 ----- CE4
  Bit 3 ----- CE3
  Bit 2 ----- CE2
  Bit 1 ----- CE1
  Bit 0 ----- OP

- Except bit 6, the set value is Pin output with negative logic.
  This register can be accessed with the undisclosed instruction PPO.

- Port status specification register PE (read / write)
  Specify input / output for each port.

  Bit 7 ----- Port7 (1: output, 0: input)
  Bit 6 ----- Port6 (1: output, 0: input)
  Bit 5 ----- Port5 (1: output, 0: input)
  Bit 4 ----- Port4 (1: output, 0: input)
  Bit 3 ----- Port3 (1: output, 0: input)
  Bit 2 ----- Port2 (1: output, 0: input)
  Bit 1 ----- Port1 (1: output, 0: input)
  Bit 0 ----- Port0 (1: output, 0: input)

  All bits are cleared to 0 by RESET and power OFF. (Input state)

- Port data register PD (read / write)
  Data input / output of each port is performed according to the state specified in the PE register.

  Bit 7 ----- Port7 data
  Bit 6 ----- Port6 data
  Bit 5 ----- Port5 data
  Bit 4 ----- Port4 data
  Bit 3 ----- Port3 data
  Bit 2 ----- Port2 data
  Bit 1 ----- Port1 data
  Bit 0 ----- Port0 data

It cannot be initialized by RESET or power OFF. (Indefinite)

- Timer data register TM (read)
  Stores the HD61700 built-in timer value. It can be reset (cleared to 0) by the CLT instruction and read to any main register by the GST instruction. Depending on the timing of reading, there may be a change point of the value (FFh can be read), so it is necessary to read twice when using.

  Bit 7, 6 ---- 4-minute count (0-3)
  Bit 0 to 5 ---- Count value for 60 seconds (0 to 59. It returns to 0 in 60 seconds. The 1-minute timer interrupt is triggered by the 60th second (when it changes from 59 to 0))

- Note: CLT instruction reset (0 clear) does not work properly for the last 1/65536 seconds at 60 seconds (when changing from 59 to 0). Therefore, in order to surely perform the reset operation, it is necessary to execute the CLT instruction twice with a delay so as to avoid the above period. (Refer to the CLT instruction for an example.)

- Interrupt control and memory bank range specification register IB (read / write) Not disclosed
  Enable / disable power ON function by 1 minute timer (Bit 5), various interrupt status flags (Bit 4-0), and specify the effective range of memory bank by UA with 2 bits (Bit 7, 6).
  - Bit 7, 6- Specifies the bank switching start address (upper 2 bits) by UA.
    - IB specified status UA register switching range
    - 00XXXXXXB 0000-FFFF
    - 01XXXXXXB 4000-FFFF
    - 10XXXXXXB 8000-FFFF (PB-1000 default)
    - 11XXXXXXB C000-FFFF
  - Bit 5 ----- Power ON control by 1 minute timer 1: Permitted (ON) / 0: Prohibited (OFF)
  - By turning this bit ON, the power ON function by the 1-minute timer is permitted while the power is OFF.
  - By using this function, the time can be updated even when the power is off.
  - Bit 4 ----- IRQ1 interrupt status flag (read only 1: interrupt is occurring, 0: RTNI)
  - Bit 3 ----- Pulse / key interrupt status flag (read only 1: interrupt is occurring, 0: RTNI)
  - Bit 2 ----- IRQ2 interrupt status flag (read only 1: interrupt is occurring, 0: RTNI)
  - Bit 1 ----- 1-minute timer interrupt status flag (read only 1: interrupt is occurring, 0: RTNI)
  - Bit 0 ----- Interrupt status flag from / ON pin (read only 1: interrupt is occurring, 0: RTNI)

- Looking at the processing in the PB-1000 ROM, it is used in the following procedure, and it turns out that the power ON control by 1 minute timer and the bank designation range by UA are fixed to & H8000 to & HFFFF.

- 56 40 80 PST IB, & H80 ; Bit 5 is turned off (power on by 1 minute timer is prohibited),
- ; Fix the bank range to & H8000 to & hFFFF.
- <Set the timer control work area>
- 57 20 10 PST IE, & H10 ; Allow 1 minute timer interrupt
- 56 40 A0 PST IB, & HA0 ; Bit 5 ON (Allow power ON with 1 minute timer permission),

- ; Fix the bank range to & H8000 to & hFFFF. Access to this IB register is performed only on the PB-1000 / C, and does not appear to be performed on the FX-870P / VX-4 / VX-3.
  (Because the clock function is not supported and there is no need to specify the bank range)
  * In the EU (Europe) format, this register is called CS.

- Key input register KY (16Bit: read)
  Returns the 12-bit key input result (KI01 to KI12) and the external interrupt input level (undisclosed).

Bit 15 ------- Keyboard port Pin input (KI04)
Bit 14 ------- Keyboard port Pin input (KI03)

Bit 13 -------	Keyboard port Pin input (KI02)
Bit 12 -------	Keyboard port Pin input (KI01)
Bit 11 -------	IRQ1 input level (unreleased)
Bit 10 -------	IRQ2 input level (unreleased)
Bit 9 -------	Interrupt input level from / ON pin (not disclosed)
Bit 8 -------	Unknown use
Bit 7 -------	Keyboard port Pin input (KI12)
Bit 6 -------	Keyboard port Pin input (KI11)
Bit 5 -------	Keyboard port Pin input (KI10)
Bit 4 -------	Keyboard port Pin input (KI09)
Bit 3 -------	Keyboard port Pin input (KI08)
Bit 2 -------	Keyboard port Pin input (KI07)
Bit 1 -------	Keyboard port Pin input (KI06)
Bit 0 -------	Keyboard port Pin input (KI05)

7-3 Assembler

Assembler Format

- The description of the instruction in the **instruction word format** assembler is as follows.
- **([*LABEL* :]) [*Mnemonic*] [*OP1*] [, *OP2*] [, *OP3*] ・・・ ([; *Comment*])**
  A space or TAB is required between the mnemonic and operand 1 (*OP1*).
  (In practice, space / TAB is not required except for some commands, but it is necessary in the specification.)
  Use commas to separate operand 2 and later.
  Mnemonic / operand descriptions are not case sensitive.
- Label declarations and comments are optional.
- **Numeric** values support 8-bit integer types (IM8: 0 to 255) and 16-bit integer types (IM16: 0 to 65535). In addition to decimal numbers, prefixes & H (hexadecimal) and & B (binary: available for HD61) are also possible.
- **Labels** can be described up to 16 characters (5 characters for PB-1000). Available characters are "@", "_", "A to Z", "a to z", "0 to 9".
  The first character must be other than a number, and is different from mnemonics in that uppercase and lowercase letters are distinguished. (PB-1000 is not case sensitive)
  In addition to addressing labels, numeric labels can be defined with the EQU directive.
  The defined label can be used with all operands for which a numeric value can be specified.
- **Expressions and operators**
  The HD61 can use operations with labels or expressions (operand operations) as numeric operands.
  Operand operations are not limited to specific instructions and can be used with all instructions that use numeric values.
  The operations are sequentially executed according to the following priority order.

Available operators (priority from top to bottom)

Priority	Calculation Type	Operator
High	**Unary operator**	.H (or .U) upper 8 bits specified, .L (or .D) lower 8 bits specified, .N bit inverted
↑	**Inversion of evaluation**	!
	Parenthesis operation	()
	Four arithmetic operations	* Multiplication, / division,% remainder (MOD)
↓	**Four arithmetic operations**	+ Addition,-subtraction
Low	**Logical operation**	& AND (may be #), | OR, ^ XOR
	Relational (comparison) operations	= Equal sign (equal),> <> = <= size comparison, <> inequality sign

Pseudo Instructions

HD61 supports the following pseudo-instructions.
Basically, it is compatible with the PB-1000 built-in assembler pseudo-instructions, but there are some minor differences such as the use of labels and expressions.

Pseudo Instructions

- {} Indicates one of them. However, {} itself is not entered.
- [] Can be omitted. However, do not enter [] itself.

Pseudo-instruction No.	Pseudo-instruction	Format	Function
(1)	**ORG**	ORG [*address* | *LABEL* | *expression*]	Declare the address where code placement starts to the assembler. Multiple ORGs may be used in a program, but an ORG declaration smaller than the assembly address at the described location cannot be made. This declaration must be written at the top of the program. (In fact, it may be after START or EQU) A label or expression can be used as an operand, but the value must be determined at the time of use.
(2)	**START**	START [*execution start address* | *LABEL* | *expression*]	Give the program execution start address. **Can be declared only once during the program.**
(3)	**EQU**	*LABEL* : EQU [*number* | *LABEL* | *expression*]	Gives the numeric value of the operand for the declared label. Label declaration cannot be omitted. A label or expression can be used for the operand value, but the value must be determined at the time of use. In addition, a character string of up to 2 bytes can be specified by enclosing with a quotation mark. Example) LABEL: EQU "AB"; Substitute & H4241. (Same as DB pseudo-instruction, from left to lower and higher)
(4)	**DB**	[*LABEL* :] DB { *number* | " *string* " | *LABEL* | *expression* } [, { *number* | " *string* " | *LABEL* | *expression* } [, ・・・]]	The numerical value (and character) string described after operand 1 is stored in memory in bytes. The label on the left side of the DB instruction can be omitted. When specifying a character string, enclose it in double quotations ["] or single quotations [']. Operand value must be in the range of 0 to 255, and can be described by a label or expression. Example) DB 1, 2, 3, "ABCDEF 0 1 2", & H20 DB 'ABCDEF'
(5)	**DW**	[*LABEL* :] DW { *number* | *LABEL* | *expression* } [, { *number* | *LABEL* | *expression* } [, ・・・]]	The numerical value described after operand 1 is stored in memory in word units. Operands can use labels and expressions, but not strings. This pseudo-instruction is not in PB-1000.

(6)	**DS**	[*LABEL* :] DS { *number* | *LABEL* | *expression* }	A number of bytes equal to the numerical value described in operand 1 is secured in the code memory. 0 is stored in the reserved area. (Undefined data in the PB-1000 built-in assembler) The label on the left side of DS can be omitted. A label or expression can be used for the operand value, but the value must be fixed.
(7)	**LEVEL**	LEVEL Numerical value (0 or 1)	Controls optimization of transfer instructions for CASIO-specific SIR settings (SX = 31, SY = 30, SZ = 0) during assembly. At LEVEL 1, optimization is turned on and transfer instructions for $ 31, $ 30, and $ 0 are optimized. Turn off optimization at LEVEL 0 and output code compatible with PB-1000 built-in assembler. **The default is LEVEL 1.** When changing SIR with the PSR instruction, LEVEL 0 must be specified. (For details, refer to HD61 attachment HD61700.S)
(8)	**IF ~ ELSE ~ ENDIF**	#IF [!] *Expression* *Description 1* [#ELSE *Description 2*] #ENDIF	If the value of operand # 1 of the #IF instruction is true (other than 0), description 1 is validated and description 2 from #else to #endif is invalidated. If operand 1 is false (0), description 2 is valid. The part of (#ELSE description 2) can be omitted. The operator! [Reverse evaluation value] can be used in the expression. (The precedence of the! Operator is between the unary operator and the parentheses.) A label can be used in the expression, but the value must be fixed. # IF ~ # ELSE ~ # ENDIF statements can be nested up to 255 levels.
(9)	**#INCLUDE**	#INCLUDE (*file name*)	Include the file described in parentheses in operand 1 when assembling. When the assembler finds this statement, it stops assembling the source file and assembles the file specified by #INCLUDE. After assembling the specified file, resume assembling the original source file. #INCLUDE nesting is possible up to 256 levels. If you recursively call an INCLUDE file that has already been opened, an "Invalid Include File Name" error will occur. By default, the list file is output even during #INCLUDE processing. To control the list output during #INCLUDE, use the # NOLIST / # LIST pseudo-instruction described later.
(10)	**#INCBIN**	#INCBIN ({ *file* name.BMP | *file name* })	Include the binary file described in the parentheses of operand 1 when assembling. When the assembler finds this sentence, it converts the specified file into DB format as binary data and reads it. If the address exceeds 64KB during conversion, the process terminates with an error. When a Windows bitmap format file (extension .BMP) is specified, pixel data is converted into graphic data for LCD display and read. Only monochrome two-color format can be read. (Up to 64KB

			size limit) For other BMP formats, "Illigal Bitmap File Format" is displayed and the process ends with an error. If a file name with another extension (other than .bmp) is specified, it is converted to DB format as continuous binary data.
(11)	**#NOLIST, #LIST, #EJECT**	#NOLIST #LIST #EJECT	Control output to list (.lst file). #NOLIST command stops output of subsequent lines to the .lst file. Output with the #LIST command. The #EJECT instruction outputs LINE FEED (& h0C). (The page header is also output at the same time.)
(12)	**#KC #AI #EU**	#KC #AI #EU	Specify mnemonic format (KC format / AI format / EU (Europe) format). (Default is #AI specification) By this specification, the subsequent grammar check process operates according to each format. When #EU is specified, EU (Europe) format mnemonics are used. For details, refer to [◆ EU (Europe) mnemonic] in the next section. In the default #AI specification, if the third operand of the LDM / STM instruction is omitted when assembling, the following warning is displayed to indicate that it has been interpreted in KC format. "WARNING: 'LDM' was interpreted to 'LDD' of the KC form." This is due to the fact that the KC format LDM (LoaD Minus) and STM (STore Minus) have the same mnemonic name as the AI format LDM (LoaD Multi byte) and STM (STore Multi byte). . (Determination of AI format or KC format by presence / absence of third operand) By specifying the pseudo-instruction '#KC', the warning is not displayed.

Programming Points

Using optimization by $ 30, $ 31, $ 0
In the CASIO pocket computer (HD61700) system, $ 31 = 0 and $ 30 = 1 are always set, and in principle, these settings are not changed.
Using these settings provides various benefits.
For example, when $ 2 is cleared to zero, it is normally done as follows.

```
LD      $ 2, 0
Or
XR      $ 2, $ 2
```

However, the following method is common for CASIO Pokekon using HD61700.

```
LD      $ 2, $ 31       ; 0 (= $ 31) is assigned
```

The reason is that in the CASIO pocket computer (HD61700) system, SIR: specific index register is fixed as SX = 31 ($ 31 specification), SY = 30 ($ 30 specification), SZ = 0 ($ 0 specification), these (SX / SY This is because the transfer / calculation using / SZ) can reduce the instruction size and the number of execution clocks by specifying the second operation.
In the above example, when assembled with LEVEL 1 specified, the first two become 3-byte instructions, but the third instruction becomes a 2-byte instruction.
Conventional commands include the following.

```
LD      $ 2, $ 30       ; 1 is assigned
AD      $ 2, $ 30       ; Increment: +1
SB      $ 2, $ 30       ; Decrement: -1
ADW     $ 2, $ 30       ; Word increment: +1
SBW     $ 2, $ 30       ; Word decrement: -1
LD      $ 2, (IX + $ 31)  ; Same operation as LD $ 2, (IX + 0)
ST      $ 2, (IX + $ 31)  ; Same as ST $ 2, (IX + 0)
```

For the same reason, when performing arbitrary operations, if $ 0 (or $ 0, $ 1 pair) is used as the second operand, optimization by $ SZ is performed, and the instruction size and the number of execution clocks are reduced. .

This optimization for $ SX = $ 31, $ SY = $ 30, and $ SZ = $ 0 is effective in almost all transfer / operation systems between main registers.

For details on instructions that can be optimized, see 4. Mnemonic , HD61700.pdf attached to HD61, or HD61700.S (and .lst).

In HD61, LEVEL 1 is specified as the default setting, and assembly is performed using CASIO Pokekon system-compliant optimization (SIR setting is fixed to SX = 31, SY = 30, SZ = 0).
To turn off optimization for $ 31, $ 30, and $ 0, specify the '/ n' option when assembling or set LEVEL 0 using the LEVEL pseudo-instruction.

In that case, it is necessary to explicitly specify indirect with $ SX / $ SY / $ SZ for the instruction that needs to be optimized. (Optimization by indirect specification using SIR works regardless of LEVEL 0/1)

Mnemonic Format

This section gives a brief description of each mnemonic format. If you are interested in details, please refer to the lst file output after assembling HD61700.S.

KC Format Mnemonic
An example of the unpublished command format is "KC format".
The KC format is a mnemonic format published in Kota-chan's "KC-Disasemmbler" (reference (3)).
As with the “AI-assembler format” (reference (4)), almost all unpublished commands are supported.
The differences between the “AI-assembler” format and the “KC format” are as follows.

Differences between AI-assembler format and KC format			
Order	**AI - Assembler Format**	**KC format**	**Remarks**
Decrement instructions	LDD *	LDM *	
	STD *	STM *	
Multibyte instructions	** M	** W	In the AI-assembler format, the multibyte number is described as ", *IM3* ". In KC format, *write* "(*IM3*)" in parentheses.

Refer to each mnemonic for details.
In Japan, the KC format was not as popular as the AI-assembler format.

- The AI-assembler (reference (4)), which appeared as the first HD61700 assembler, had a systematic and easy-to-understand grammar, whereas the KC format uses multibyte instructions to enclose multibyte numbers in parentheses. There were disadvantages in parsing.
- The fact that the systematic explanation of the KC format was not made in the first presentation (reference (3)) seems to be one of the reasons why its spread was hindered.

The KC format was partly supported for the first time by the “FX-870P Assembler” (reference (5)) and fully supported by the “X-Assembler” (references (6), (7)).
In this way, the KC format did not spread, but remained until the end.

In addition to the “AI-assembler format” and “KC format”, HD61 supports both unpublished instructions added in “X-Assembler” in both formats. (There may be a subtle omission)

Rev0.41 and later also support DP format (described later).

EU (Europe) format
"EU format", a format used mainly within the European (Germany) community.
HD61 can be used after Rev0.41 by specifying #EU (or / eu option).

In Germany, the PB-1000 ROM disassembly list was published in a magazine with explanation in 1988, and unofficially, an environment that supports this EU (Europe) format mnemonic (Pascal card for PB-2000) The information about this mnemonic was widely known because it was provided.
This European mnemonic is said to contain unpublished information provided by CASIO (= close to CASIO genuine notation) due to the publication timing of magazine articles in Germany, etc., and is a very interesting notation.
(Since it was not confirmed by CASIO, it is unknown whether it is true)
On the other hand, the AI format / KC format has been analyzed and named by several analysts who have

nothing to do with CASIO through magazine articles in Japan, and the results are very wonderful.
EU format and AI format differ in the following points (1) to (6). For details, refer to the description of each instruction.

Differences between AI / KC format and EU format				
No.	**Difference**	**AI / KC Format**	**EU Format**	**Comment**
(1)	Specific index register: SIR (Specific Index Register)	SX, $ SX SY, $ SY SZ, $ SZ	# 0 # 1 # 2	In the EU format, it is called short register: SR (Short Registers).
(2)	Register Name	IB	CS	An undisclosed register in AI format, denoted as IB, is denoted as CS in EU format.
(3)	Undocumented Mnemonic	PSR	PRA	PRA (Put Ram Address)
		GSR	GRA	GRA (Get Ram Address)
		STL	OCB	OCB (Output Casio Bus)
		LDL	ICB	ICB (Input Casio Bus)
		PPO	PCB	PCB (Put Casio Bus)
		BUPS IM8	BUP IM8	
		BDNS IM8	BDN IM8	
		JP $ C5	JPW $ C5	
		JP ($ C5)	JPW ($ C5)	
(4)	Multibyte instructions	* M	* L	In the EU format, "L" (meaning long word?) Is added to the end of the mnemonic for multibyte instructions.
(5)	Multibyte count	2-8	L2 to L8	In EU format, the same notation as AI format is also possible.
(6)	JUMP expansion Tag expression	JR	J.	This tag can be omitted in AI / KC / EU format. In the EU format, "JR" can also be used.

7-4 Mnemonic

This chapter explains the mnemonics of the HD61700. The operand symbols and mnemonics used in mnemonics are shown below.

List of operand symbols used in mnemonics		
Operand	**Symbol**	**Comment**
Main register	*$ C5* : $ 0, $ 1, ・・・, $ 31	Hexadecimal representation of $ & H0, $ & H1, ・・・, $ & H1F is also possible.
Specific index register SIR	SX	5-bit
	SY	
	SZ	
Indirect specification of main register by specific index register	$ SX | $ (SX)	Main register indicated by SX (default: $ 31)
	$ SY | $ (SY)	Main register indicated by SY (default: $ 30)
	$ SZ | $ (SZ)	Main register indicated by SZ (default: $ 0)
Index register IR	IX	16-bit **The IY register can only be used as an end point pointer for block transfer / search instructions.**
	IY	
	IZ	
Stack pointer	SSP	System stack pointer (16-bit)
	USP	User stack pointer (16-bit)
Program counter	PC	Program counter (16-bit)
flag	Z	Zero flag
	NZ	Non-zero flag
	C	Carry flag
	NC	Non-carry flag
	LZ	Lower-digit zero flag
	UZ	Upper-digit zero flag
	NLZ | LNZ	Non lower-digit zero flag
Status register	KY	KEY input register (16-bit)
	IE	Interrupt enable register (8-bit)
	IA	Interrupt selection register (8-bit)
	IB	Interrupt control and bank control register (8-bit); not disclosed
	UA	Upper address specification register (8-bit)
	PD	Port data register (8 bits)

	PE	Port status specification register (8 bits)
	TM	Timer register (8 bits)
Numerical data	*IM3* : 2,3, ..., 8	3-bit direct value. Used to specify the number of multibytes.
	IM5 : 0 to 31 or & H0 to & H1F	5-bit direct value. Used for ADBM and SBBM.
	IM8 : 0 to 255, or & H00 to & HFF	8-bit direct value.
	IM16 : 0 to 65535, or & H0000 to & HFFFF	16-bit direct value.

Mnemonic Table - Transfer instruction (8 bits)

- {} Indicates one of them. However, {} itself is not entered.
- [] Can be omitted. However, do not enter [] itself.

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example format
LD (Load)	LD ***opr1*** , ***opr2*** [, (JR) ***LABEL***]	opr1 ← opr2	It does not change	-	Transfer the contents of opr2 to opr1. Unreleased but with jump extension. By adding an address label to operand 3 when a specific combination of opr1 and opr2 is executed, a relative jump is made after execution of the transfer. Operand 3 and JR tag can be omitted. There are six types of operand combinations that can be used with the LD instruction. Refer to the following for the applicability of jump extension.	
	LD ***$ C5*** , ***$ C5*** [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ $ C5		3 + 3 + 6 = 12 (JR: +3)	Transfer between main registers	LD $ 2, $ 0; $ 0 data transferred to $ 2
	LD ***$ C5*** , (***$ A***) [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ ($ A)		$ A = $ SIR: 3 + 8 + 3 = 14 $ A = $ C5:	Transfer from external memory to main register (1) $ A is $ C5, $ SIR.	LD $ 2, ($ 0); Transfer external memory data addressed to $ 0

				3 + 3 + 8 + 3 = 17 (JR: +3)	opr2 is little endian and 2 bytes. The bank is for the UA register IX.	(lower) and $ 1 (upper) to $ 2 LD $ 2, ($ SZ); SZ = 0 by default, so the same operation as LD $ 2, ($ 0) is executed at high speed
	LD ***$ C5***, ({IX | IZ} ± ***$ C5***)	opr1 @ $ C5 ← opr2 @ ({IX | IZ} ± $ C5)		3 + 3 + 6 + 5 = 17	Transfer from external memory to main register (2) Specification by index register ± main register (8 bits). No jump extension.	LD $ 2, (IX + $ 31); Transfer external memory data addressed to IX + $ 31 to $ 2.
	LD ***$ C5***, ({IX | IZ} ± ***IM8***)	opr1 @ $ C5 ← opr2 @ ({IX | IZ} ± IM8)		3 + 3 + 6 + 5 = 17	Transfer from external memory to main register (3) Specification by index register ± 8-bit immediate value. No jump extension.	LD $ 2, (IX + 123); Transfer external memory data addressed to IX + 123 to $ 2.
	LD ***$ C5***, ***IM8*** [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ IM8		3 + 3 + 6 = 12 (JR: +3)	Transfer 8-bit immediate data to the main register	LD $ 4,123; 123 transferred to $ 4
	LD ***$ C5***, ***$ SIR*** [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ $ SIR		3 + 6 = 9 (JR: +3)	Indirect transfer of main register by specific index register SIR (undisclosed instruction) Compared with normal register specification, the instruction code is 1 byte shorter. (When LEVEL 0 is specified) As a result, the execution clock is shortened and used frequently in ROM. In the EU format, SX = # 0, SY = # 1, SZ = # 2, and the JR tag can be omitted and "J." can be written.	LD $ 4, $ SX; The main register value (8 bits) indicated by $ SX is transferred to $ 4. By default, $ SX = $ 31 = 0. EU format LD $ 4, # 0; LD $ 4, $ SX LD $ 4, # 0, J.LABEL;
LDI (Load Increment)	LDI ***$ C5***, (***IR*** ± ***A***)	$ C5 ← (IR ± A) IR ← IR ± A + 1	It does not change	A = SIR: 3 + 6 + 5 = 14 A = $ C5: 3 + 3 + 6 + 5	After the contents of the external memory with (IR ± A) as the address are transferred	LDI $ 4, (IX + $ 2); Specify main register LDI $ 4, (IZ- $ 2);

				= 17 A = IM8: 3 + 3 + 6 + 5 = 17	to the main register $ C5, the incremented transfer memory address is assigned to IR. IR is IX, IZ. For A, $ C5, SIR, and IM8 are applicable.	LDI $ 4, (IX + $ SX); Indirect designation by SIR (unpublished) LDI $ 4, (IZ- $ SY); LDI $ 4, (IX + 123); 8-bit immediate designation LDI $ 4, (IZ-123);
LDD (Load Decrement)	LDD ***$ C5*** , (***IR*** ± ***A***)	$ C5 ← (IR ± A) IR ← IR ± A	It does not change	A = SIR: 3 + 6 + 3 = 12 A = $ C5: 3 + 3 + 6 + 3 = 15 A = IM8: 3 + 3 + 6 + 3 = 15	Transfer the contents of the external memory whose address is (IR ± A) to the main register $ C5, and then assign the transfer memory address to IR. Unlike the association from the LDI instruction, it does not actually decrease, but the execution clock is shorter. IR is IX, IZ. $ C5, SIR, IM8 can be applied to A.	LDD $ 4, (IX- $ 2); Specify main register LDD $ 4, (IZ + $ 2); LDD $ 4, (IX- $ SX); Indirect designation by SIR (unpublished) LDD $ 4, (IZ + $ SZ); LDD $ 4, (IX-123); 8-bit immediate designation LDD $ 4, (IZ + 123);
LDC (Load Check)	LDC ***$ C5*** , ***opr2*** [, (JR) ***LABEL***]	No operation	It does not change	A = SIR: 3 + 6 = 9 A = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	Operands can be specified in the same format as the LD instruction, but no processing is actually performed and only instruction decoding is performed. Delay processing as with the NOP instruction. However, if there is a label in the third operand, a relative jump is made. (JR tag can be omitted)	LDC $ 4, $ 2; Main register specified LDC $ 4, $ SX; Indirect designation by SIR LDC $ 4,128; 8-bit immediate designation LDC $ 4, $ 3, ERROR; Register specification + Jump expansion
ST (Store)	ST ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A)	It does not change	A = SIR: 3 + 6 + 5 = 14 A = $ C5, IM8: 3 + 3 + 6 + 5 = 17	The contents of the first operand $ C5 are stored in an external memory whose address is (IR ± A). Note that the transfer direction is opposite to the LD command. IR is IX, IZ.	ST $ 4, (IX + $ 2); Specify main register ST $ 4, (IZ- $ 2); ST $ 4, (IX + $ SX); Indirect designation by SIR (unpublished) ST $ 4, (IZ- $ SY);

					$ C5, SIR, IM8 can be applied to A.	ST $ 4, (IX + 123); 8-bit immediate designation ST $ 4, (IZ-123);
ST (Store $)	ST ***$ C5*** , (***A***) [, (JR) ***LABEL***]	$ C5 → (A)	It does not change	A = SIR: 3 + 8 + 3 = 14 A = $ C5: 3 + 3 + 8 + 3 = 17 (JR: +3)	The contents of the first operand $ C5 are stored in the external memory with A as the address. Note that the transfer direction is opposite to the LD command. A can be $ C5, SIR. If there is a label in the third operand, a relative jump is made after the transfer. (JR tag can be omitted)	ST $ 2, ($ 0); Second operation specification (2 bytes) ?? 3 bytes ST $ 2, ($ SZ); Indirect specification by SIR (2 bytes). Virtually only $ SZ = $ 0 can be used. ($ SX = $ 31 ($ 31, $ 0 pair), $ SY = $ 30 ($ 30, $ 31 pair = 0001) can be specified, but the utility value is low.) ST $ 2, ($ 10); Normal (3 bytes) ST $ 2, ($ 10), LABEL; Jump expansion (4 bytes)
ST (Store IM8) undisclosed instruction	ST ***IM8*** , (***$ SIR***)	IM8 → ($ SIR)	It does not change	3 + 3 + 8 + 3 = 17	The 8-bit immediate value of the first operand is stored in the external memory indicated by the main register specified indirect by SIR. Note that the transfer direction is opposite to the LD command. Virtually only $ SZ = $ 0 can be used. ($ SX = $ 31 ($ 31, $ 0 pair), $ SY = $ 30 ($ 30, $ 31 pair = 0001) can be specified, but the utility value is low.)	ST 123, ($ SZ);
ST (Store IM8 to Register) undisclosed instruction	ST ***IM8*** , ***$ C5***	IM8 → $ C5	It does not change	3 + 3 + 11 = 17	The 8-bit immediate value of the first operand is stored in the main register specified by the second operand. Note that the transfer direction is opposite to	ST 123, $ 0;

					the LD command. Same behavior as LD $ C5, IM8, but no Jump extension.	
STI (Store Increment)	STI ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A) IR ← IR ± A + 1	It does not change	A = SIR: 3 + 6 + 5 = 14 A = $ C5, IM8: 3 + 3 + 6 + 5 = 17	The contents of the first operand $ C5 are stored in an external memory whose address is (IR ± A). In IR, the incremented transfer destination address is stored. Note that the transfer direction is opposite to the LD command. IR is IX, IZ. $ C5, SIR, IM8 can be applied to A.	STI $ 4, (IX + $ 2); Specify main register STI $ 4, (IZ- $ 2); STI $ 4, (IX + $ SX); Indirect designation by SIR STI $ 4, (IZ- $ SY); STI $ 4, (IX + 123); 8-bit immediate designation STI $ 4, (IZ-123);
STD (Store Decrement) undisclosed instruction	STD ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A) IR ← IR ± A	It does not change	A = SIR: 3 + 6 + 3 = 12 A = $ C5, IM8: 3 + 3 + 6 + 3 = 15	The contents of the first operand $ C5 are stored in an external memory whose address is (IR ± A). The transfer destination address is stored in IR. As with LDD , the decrement associated with the name is not actually performed. Note that the transfer direction is opposite to the LD command. IR is IX, IZ. $ C5, SIR, IM8 can be applied to A.	STD $ 4, (IX + $ 2); Specify main register STD $ 4, (IZ- $ 2); STD $ 4, (IX + $ SX); Indirect designation by SIR STD $ 4, (IZ- $ SY); STD $ 4, (IX + 123); 8-bit immediate designation STD $ 4, (IZ-123);
PPS (Pop by System stack pointer)	PPS ***$ C5***	$ C5 ← (SS) SS ← SS + 1	It does not change	3 + 6 + 5 = 14	After the contents of external memory specified by SS are stored in main register $ C5, SS is incremented.	PPS $ 2;
PPU (Pop by User stack pointer)	PPU ***$ C5***	$ C5 ← (US) US ← US + 1	It does not change	3 + 6 + 5 = 14	After storing the contents of the external memory specified by US in main register $ C5, US is incremented.	PPU $ 2;
PHS (Push by System stack pointer)	PHS ***$ C5***	$ C5 → (SS-1) SS ← SS-1	It does not change	3 + 6 + 3 = 12	After storing the value of main register $ C5 in the external memory specified by SS-1, SS is decremented.	PHS $ 2;

PHU (Push by User stack pointer)	PHU ***$ C5***	$ C5 → (US-1) US ← US-1	It does not change	3 + 6 + 3 = 12	After storing the value of main register $ C5 in the external memory specified by US-1, decrement US.	PHU $ 2;
GFL (Get Flag)	GFL ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← F	It does not change	3 + 6 = 9 (JR: +3)	The contents of the flag register are stored in the main register $ C5 designated by the first operand. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	GFL $ 2; GFL $ 2, LABEL; Jump expansion
PFL (Put Flag)	PFL ***A*** [, (JR) ***LABEL***]	A → F	Change with the value of A	A = $ C5: 3 + 6 = 9 A = IM8: 3 + 3 + 6 = 12 (JR: +3)	Store the contents of A in operand 1 in the flag register. (Only the upper 4 bits can be set .) A is $ C5, IM8. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	PFL $ 2; PFL $ 2, LABEL; Jump expansion
GPO (Get Port)	GPO ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← Port	It does not change	3 + 6 = 9 (JR: +3)	The contents of the port terminal are stored in the main register $ C5 specified by the first operand. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	GPO $ 2; GPO $ 2, LABEL; Jump expansion
GST (Get Status)	GST ***Sreg*** , ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← Sreg	It does not change	3 + 6 = 9 (JR: +3)	The contents of the status register are stored in the main register $ C5 specified by the second operand. Sreg = PE, PD, UA, IA, IE, TM, IB. If there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted) The storage direction is the same as ST and minority.	GST PE, $ 2; GST IB, $ 2; Interrupt control / bank control register (undisclosed instruction) GST TM, $ 2, LABEL; Jump expansion GST IB, $ 2, LABEL; Relative jump (undisclosed instruction) after storing interrupt

						control / bank control register in $ 2 EU format GST CS, $ 2; Interrupt control / bank control register (undisclosed instruction) GST CS, $ 2, J.LABEL; Relative jump (undisclosed instruction) after storing interrupt control / bank control register in $ 2
PST (Put Status)	GST ***Sreg*** , ***A*** [, (JR) ***LABEL***]	Sreg ← A	It does not change	A = $ C5: 3 + 6 = 9 A = IM8: 3 + 3 + 6 = 12 (JR: +3)	Stores the value of A specified by the second operand in the status register. Sreg = PE, PD, UA, IA, IE, TM, IB. A is $ C5, IM8. If there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted)	PST PE, $ 2; Main register transfer PST IB, $ 2; Interrupt control / bank control register (undisclosed instruction) PST TM, $ 2, LABEL; Jump expansion PST IB, $ 2, LABEL; Relative jump (undisclosed instruction) after storing value of $ 2 in interrupt control / bank control register PST UA, 123; 8-bit immediate value transfer PST IB, 123; Interrupt control / bank control register (undisclosed instruction) EU format PST CS, $ 2; Interrupt control / bank control

						register (undisclosed instruction) PST CS, 123, J.LABEL; Relative jump (undisclosed instruction) after storing 123 in interrupt control / bank control register
STL (Store data to LCD) undisclosed instruction	STL ***A*** [, (JR) ***LABEL***]	A → LCD	It does not change	Without JR: 3 + 15 = 18 With JR: 3 + 3 + 14 = 20	Outputs the A value specified by the first operand to the LCD data area. A is $ C5, IM8. If $ C5 is specified and there is a label for the second operand, a relative jump occurs after transfer. (JR tag can be omitted)	STL $ 2; Main register STL $ 2, LABEL; Jump expansion STL 123; 8-bit immediate value output EU format OCB $ 2; main register OCB $ 2, LABEL; Jump extension OCB 123; 8-bit immediate value output
LDL (Load data from LCD) undisclosed instruction	LDL ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← LCD port data	It does not change	Without JR: 3 + 15 = 18 With JR: 3 + 3 + 14 = 20	The value of the LCD data port is stored in the first operand $ C5 according to the transfer protocol set in advance in the LCDC. Since reading is performed in units of 4 bits, graphic data on the screen is read with the upper and lower 4 bits replaced. For example, if the dot on the screen is & H4A display, executing LDL $ C5 results in $ C5 = & HA4. The readout procedure is as follows. (1) Specify drawing mode (anything) and LCD coordinate position to LCDC. (STLM after PPO & HDF) (2) Set read command	LDL $ 2; Main register LDL $ 2, LABEL; Jump extension EU format ICB $ 2; Main register ICB $ 2, LABEL; Jump expansion

					(& HE1) to LCDC. (After PPO & hDF, STL & HE1) (3) Execute LDL with data RAM specified. (LDL after PPO & HDE) If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	
PPO (Put lcd control Port) undisclosed instruction	PPO ***A*** [, (JR) ***LABEL***]	A → LCD control port	It does not change	A = $ C5: 3 + 6 = 9 A = IM8: 3 + 3 + 6 = 12 (JR: +3)	Outputs the A value specified by the first operand to the LCD control port. A is $ C5, IM8. If $ C5 is specified and there is a label for the second operand, a relative jump is performed after the transfer. (JR tag can be omitted)	PPO $ 2; Main register PPO $ 2, LABEL; Jump expansion PPO 123; 8-bit immediate value output EU format PCB $ 2; Main register PCB $ 2, LABEL; Jump expansion PCB 123; 8-bit immediate value output
PSR (Put Specific index Register) undisclosed instruction	PSR ***SIR*** , ***A*** [, (JR) ***LABEL***]	SIR ← A	It does not change	3 + 6 = 9 (JR: +3)	PSR SX, $ 2; Main register PSR SY, $ 2, LABEL; Jump expansion PSR SZ, 15; 5-bit immediate value (0-31) EU format PRA # 1, $ 2; Main register PRA # 2, $ 2, J.LABEL; Jump expansion PRA # 0,15; 5-bit immediate value (0-31)	
	The value of the second operand A is stored in the specific index register SIR designated by the first operand . SIR = SX, SY, SZ. A is $ C5, IM8. If $ C5 is specified and there is a label for the third operand, a relative jump is performed after transfer. (JR tag can be omitted) If **this command is used to change the SIR setting (usually fixed at SX = 31, SY = 30, SZ = 0) and control is returned to the system, it will run out of control.** When users change SIR, the following cautions are required. (1) Disable interrupts while changing SIR. (2) When returning to ROM processing or calling ROM processing, return SIR to its original setting. (3) Coding the optimization switch with OFF (LEVEL 0) specified. (Because it is optimized at $ 31, $ 30, and $ 0 at LEVEL 1, the code gets confused.)					
GSR (Get Specific index Register) undisclosed instruction	GSR ***SIR*** , ***$ C5*** [, (JR) ***LABEL***]	SIR → $ C5	It does not change	3 + 6 = 9 (JR: +3)	The contents of the specific index register SIR designated by the first operand are stored in the main register $	GSR SX, $ 2; GSR SY, $ 2, LABEL; Jump expansion EU format

					C5 of the second operand. SIR = SX, SY, SZ. If there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted)	GRA # 2, $ 2; GRA # 0, $ 2, J.LABEL; Jump expansion
Mnemonic Table - Transfer Instruction (16 bits)						
Mnemonic	**Format**	**Function**	**Flag**	**Number of Clocks**	**Description**	**Example format**
LDW (Load Word)	LDW ***opr1*** , ***opr2*** [, (JR) ***LABEL***]	opr1 ← opr2	It does not change	-	Transfer the contents of opr2 to opr1. Unreleased but with jump extension. By adding an address label to operand 3 when a specific combination of opr1 and opr2 is executed, a relative jump is made after execution of the transfer. Operand 3 and JR tag can be omitted. There are five types of operand combinations that can be used with the LD instruction. Refer to the following for the applicability of the jump extension.	
	LDW ***$ C5*** , ***$ C5*** [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ $ C5		3 + 3 + 11 = 17 (JR: +3)	Transfer between main registers	LDW $ 2, $ 0; $ 0, $ 1 data transferred to $ 2, $ 3
	LDW ***$ C5*** , (***$ A***) [, (JR) ***LABEL***]	opr1 @ $ C5 ← opr2 @ ($ A)		$ A = $ SIR: 3 + 8 + 3 + 3 = 17 $ A = $ C5: 3 + 3 + 8 + 3 + 3 = 20 (JR: +3)	Transfer from external memory to main register (1) opr1 and opr2 are little endian and 2 bytes. $ A is $ C5, $ SIR. $ Bank applies to UA register IX.	LDW $ 2, ($ 0); Transfer external memory data with addresses $ 0 (lower) and $ 1 (upper) to $ 2, $ 3 LDW $ 2, ($ SZ); SZ = 0 by default, so the same operation as LDW $ 2, ($ 0) is

						executed at high speed
	LDW ***$ C5*** , ({IX | IZ} ± ***$ C5***)	opr1 @ $ C5 ← opr2 @ ({IX | IZ} ± $ C5)		3 + 3 + 6 + 3 + 5 = 20	Transfer from external memory to main register (2) Specification by index register ± main register (8 bits). No jump extension.	LDW $ 2, (IX + $ 31); Transfer external memory data addressed to IX + $ 31 to $ 2, $ 3
	LDW ***$ C5*** , ***IM16***	$ C5 ← IM16		3 + 3 + 3 + 14 = 23	Transfer 8-bit immediate data to the main register	LD $ 4, & H7012; & H12 stored in $ 4, & H70 stored in $ 5
	LDW ***$ C5*** , ***$ SIR*** [, (JR) ***LABEL***]	$ C5 ← $ SIR		3 + 11 = 14 (JR: +3)	Indirect transfer of main register by specific index register SIR (unpublished instruction) Compared with normal register specification, the instruction code is shortened by 1 byte. (When LEVEL 0 is specified) The execution clock is shortened accordingly, and it is frequently used in ROM. In the EU format, SX = # 0, SY = # 1, SZ = # 2, and the JR tag can be omitted and "J." can be written.	LDW $ 4, $ SX; Stores the main register value (8 bits) indicated by $ SX in $ 4 and the main register value (8 bits) of the main register + 1 indicated by $ SX in $ 5. By default, $ SX = $ 31 = 0, $ SX + 1 = $ 0 (variable). LDW $ 4, $ SZ; Since SZ = 0 by default, $ 4 = $ 0, $ 5 = $ 1 is assigned. EU format LDW $ 4, # 0; LD $ 4, $ SX LDW $ 4, # 0, J.LABEL;
LDIW (Load Increment Word)	LDIW ***$ C5*** , (***IR*** ± ***A***)	$ C5, $ C5 + 1 ← (IR ± A) IR ← IR ± A + 2	It does not change	A = SIR: 3 + 6 + 3 + 5 = 17 A = $ C5: 3 + 3 + 6 + 3 + 5 = 20	After the contents of the external memory with (IR ± A) as the address are transferred to the main registers $ C5 and $ C5 + 1, the value obtained by adding 2 to the transfer memory address is assigned to IR. IR is IX, IZ. For A, $ C5 and SIR are applicable. For example, in LDIW $ 2, (IX + $ 0), if IX = &	LDIW $ 4, (IX + $ 2); Specify main register LDIW $ 4, (IZ- $ 2); LDIW $ 4, (IX + $ SX); Indirect designation by SIR (unpublished) LDIW $ 4, (IZ- $ SY);

					H7000, $ 0 = 1, $ 2 ← (& H7001 memory contents) $ 3 ← (& H7002 memory contents) IX ← & H7003	
LDDW (Load Decrement Word)	LDDW ***$ C5*** , (***IR*** ± ***A***)	$ C5, $ C5-1 ← (IR ± A) IR ← IR ± A-1	It does not change	A = SIR: 3 + 3 + 6 + 3 = 15 A = $ C5: 3 + 3 + 6 + 3 + 3 = 18	Transfer the contents of external memory whose address is (IR ± A) to the main registers $ C5 and $ C5-1, and substitute IR with the decremented transfer memory address. IR is IX, IZ. For A, $ C5 and SIR are applicable. Note that, unlike LDW and LDIW, the main register pair numbers are $ C5 and $ C5-1. For example, if IX = & H7000, $ 0 = 1 in LDDW $ 2, (IX + $ 0), $ 2 ← (& H7001 memory contents) $ 1 ← (& H7000 memory contents) IX ← & H7000 (last accessed address)	LDDW $ 4, (IX- $ 10); Specify main register LDDW $ 4, (IZ + $ 10); LDDW $ 4, (IX- $ SX); Indirect designation by SIR (unreleased) LDDW $ 4, (IZ + $ SZ);
LDCW (Load Check Word)	LDCW ***$ C5*** , ***A*** [, (JR) ***LABEL***]	No operation	It does not change	A = SIR: 3 + 11 = 14 A = $ C5: 3 + 3 + 11 = 17 (JR: +3)	Operands can be specified in the same format as the LDW instruction, but no processing is actually performed and only instruction decoding is performed. Delay processing as with the NOP instruction. A = $ C5, SIR. However, if there is a label in the third operand, a relative jump is made. (JR tag can be omitted)	LDCW $ 4, $ 2; Specify main register LDCW $ 4, $ SX; Indirect designation by SIR LDCW $ 4, $ 3, ERROR; Register specification + Jump expansion LDCW $ 4, $ SZ, LABEL; Indirect specification with SIR + Jump expansion
STW (Store Word)	ST ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A) $ C5 + 1 → (IR ± A + 1)	It does not change	A = SIR: 3 + 6 + 3 + 5 = 17 A = $ C5: 3	The contents of the first operand main resist pair $ C5, $ C5 + 1 are stored in an external	STW $ 4, (IX + $ 2); Specify main register

				+ 3 + 6 + 3 + 5 = 20	memory whose address is (IR ± A). Note that the transfer direction is opposite to the LD command. IR is IX, IZ. A can be $ C5, SIR.	STW $ 4, (IZ- $ 2); STW $ 4, (IX + $ SX); Indirect designation by SIR (unpublished) STW $ 4, (IZ- $ SY);
STW (Store Word $)	STW ***$ C5*** , (***A***) [, (JR) ***LABEL***]	$ C5 → (A) $ C5 + 1 → (A + 1)	It does not change	A = SIR: 3 + 8 + 3 + 3 = 17 A = $ C5: 3 + 3 + 8 + 3 + 3 = 20 (JR: +3)	The contents of the main register pair $ C5, $ C5 + 1 of the first operand are stored in an external memory having addresses A (lower) and A + 1 (upper). Note that the transfer direction is opposite to the LD command. A can be $ C5, SIR. If there is a label in the third operand, a relative jump is made after the transfer. (JR tag can be omitted)	STW $ 2, ($ 0); second operation specification (2 bytes) ?? 3 bytes STW $ 2, ($ SZ); Indirect specification by SIR (2 bytes). Virtually only $ SZ = $ 0 can be used. ($ SX = $ 31 ($ 31, $ 0 pair), $ SY = $ 30 ($ 30, $ 31 pair = 0001) can be specified, but the utility value is low.) STW $ 2, ($ 10); Normal (3 bytes) STW $ 2, ($ 10), LABEL; Jump expansion (4 bytes) STW $ 2, ($ SZ), LABEL; Indirect specification with SIR + Jump extension (3 bytes)
STW (Store IM16) undisclosed instruction	STW ***IM16*** , (***$ SIR***)	IM16 → ($ SIR)	It does not change	3 + 3 + 3 + 8 + 3 + 3 = 23	The 16-bit immediate value of the first operand is stored in the external memory indicated by the main register specified indirect by SIR. Note that the transfer direction is opposite to the LD command. Virtually only $ SZ = $ 0 can be used. ($ SX = $	STW & H7023, ($ SZ); Indirect designation by SIR STW & H7023, ($ 0); Available at LEVEL 1. Cannot be used at LEVEL 0.

					31 ($ 31, $ 0 pair), $ SY = $ 30 ($ 30, $ 31 pair = 0001) can be specified, but the utility value is low.)	
STIW (Store Increment Word)	STIW ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A) $ C5 + 1 → (IR ± A + 1) IR ← IR ± A + 2	It does not change	A = SIR: 3 + 6 + 3 + 5 = 17 A = $ C5, IM8: 3 + 3 + 6 + 3 + 5 = 20	The contents of the first operand main resist pair $ C5, $ C5 + 1 are stored in an external memory whose address is (IR ± A). IR ± A + 2 is stored in IR. Note that the transfer direction is opposite to the LD command. IR is IX, IZ. A can be $ C5, SIR.	STI $ 4, (IX + $ 2); Specify main register STI $ 4, (IZ- $ 2); STI $ 4, (IX + $ SX); Indirect designation by SIR STI $ 4, (IZ- $ SY);
STDW (Store Decrement Word)	STDW ***$ C5*** , (***IR*** ± ***A***)	$ C5 → (IR ± A) $ C5-1 → (IR ± A-1) IR ← IR ± A-1	It does not change	A = SIR: 3 + 6 + 3 + 3 = 15 A = $ C5, IM8: 3 + 3 + 6 + 3 + 3 = 18	The contents of the first operand main resist pair $ C5, $ C5-1 are stored in an external memory whose address is (IR ± A). IR ± A-1 is stored in IR. Note that the transfer direction is opposite to the LD command. IR is IX, IZ. A can be $ C5, SIR. Note that unlike STW and STIW, the main register pair numbers are $ C5 and $ C5-1. For example, in STDW $ 2, (IX + $ 0), when IX = & H7000 and $ 0 = 1, the operation is as follows. $ 2 → (& H7001 address) $ 1 → (& H7000 address) IX ← & H7000 (last address accessed)	STDW $ 4, (IX + $ 2); Specify main register STDW $ 4, (IZ- $ 2); STDW $ 4, (IX + $ SX); Indirect designation by SIR STDW $ 4, (IZ- $ SY);
PPSW (Pop by System stack pointer Word)	PPSW ***$ C5***	$ C5 ← (SS) $ C5 + 1 ← (SS + 1) SS ← SS + 2	It does not change	3 + 6 + 3 + 5 = 17	After storing the contents of the external memory specified by SS in the main register pair $ C5, $ C5 + 1, add 2 to SS.	PPSW $ 2;

PPUW (Pop by User stack pointer Word)	PPUW ***$ C5***	$ C5 ← (US) $ C5 + 1 ← (US + 1) US ← US + 2	It does not change	3 + 6 + 3 + 5 = 17	After storing the contents of the external memory specified by US in the main register pair $ C5, $ C5 + 1, add 2 to US.	PPUW $ 2;
PHSW (Push by System stack pointer Word)	PHSW ***$ C5***	$ C5 → (SS-1) $ C5-1 → (SS-2) SS ← SS-2	It does not change	3 + 6 + 3 + 3 = 15	After storing the value of the main register pair $ C5, $ C5-1 in the external memory specified by SS-1, SS-2, subtract 2 from SS.	PHSW $ 2;
PHUW (Push by User stack pointer Word)	PHUW ***$ C5***	$ C5 → (US-1) $ C5-1 → (US-2) US ← US-2	It does not change	3 + 6 + 3 + 3 = 15	After storing the value of the main register pair $ C5, $ C5-1 in the external memory specified by US-1, US-2, subtract 2 from US.	PHUW $ 2;
GRE (Get Register)	GRE ***Reg*** , ***$ C5*** [, (JR) ***LABEL***]	Reg → $ C5	It does not change	3 + 11 = 14 (JR: +3)	Stores the contents of the status register in the second operand $ C5. Reg = IX, IY, IZ, SS, US, KY If there is a third operand label, a relative jump occurs after transfer. (JR tag can be omitted)	GRE IX, $ 2; GRE US, $ 2; GRE KY, $ 2, LABEL; Jump expansion
PRE (Put Register)	PRE ***Reg*** , ***A*** [, (JR) ***LABEL***]	Reg ← A	It does not change	A = $ C5: 3 + 11 = 14 (JR: +3) A = IM16: 3 + 3 + 3 + 11 = 20	Store the value of A of the second operand in the status register. Reg = IX, IY, IZ, SS, US, KY A is $ C5 ($ C5, $ C5 + 1 pair), IM16. If the second operand is the main register and there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted)	PRE IX, $ 2; PRE US, $ 2; PRE KY, $ 2, LABEL; Jump expansion PRE IZ, & H703F;
STLW (Store Word data to LCD) undisclosed instruction	STLW ***$ C5*** [, (JR) ***LABEL***]	$ C5 → LCD $ C5 + 1 → LCD	It does not change	3 + 22 = 25 (JR: +3)	The main register pair $ C5, $ C5 + 1 of the first operand is output to the LCD data area. Output is performed in order of 8 bits. If there is a label for the	STLW $ 2; Main register STLW $ 2, LABEL; Jump expansion EU format OCBW $ 2; Main register

					second operand, a relative jump is made after the transfer. (JR tag can be omitted)	OCBW $ 2, LABEL; Jump expansion
LDLW (Load Word data from LCD) undisclosed instruction	LDLW ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← LCD port data $ C5 + 1 ← LCD port data	It does not change	Without JR: 3 + 23 = 26 With JR: 3 + 3 + 22 = 28	The value of the LCD data port is stored in the main register pair $ C5, $ C5 + 1 designated by the first operand according to the transfer protocol set in advance in the LCDC. Since reading is performed in units of 4 bits, graphic data on the screen is read with the upper and lower 4 bits replaced. The reading procedure is as follows. (1) Specify drawing mode (anything) and LCD coordinate position to LCDC. (STLM after PPO & HDF) (2) Set read command (& HE1) to LCDC. (After PPO & HDF, STL & HE1) (3) Execute LDLW with data RAM specified. (LDLW after PPO & HDE) If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	LDLW $ 2; Main register LDLW $ 2, LABEL; Jump expansion EU format ICBW $ 2; Main register ICBW $ 2, LABEL; Jump expansion
PPOW (Put lcd control Port Word) undisclosed instruction	PPOW ***$ C5*** [, (JR) ***LABEL***]	$ C5 → LCD control port $ C5 + 1 → LCD control port	It does not change	3 + 11 = 14 (JR: +3)	The value of the main register pair $ C5, $ C5 + 1 specified by the first operand is output to the LCD control port. Output is performed in order of 8 bits. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted) Note: The I / O port accessible by this command is different	PPOW $ 2; Main register PPOW $ 2, LABEL; Jump extension EU format PCBW $ 2; Main register PCBW $ 2, LABEL; Jump expansion

					from the PD register. (I / O of LCD system)	
GFLW (Get Flag Word)	GFLW ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← F $ C5 + 1 ← F	It does not change	3 + 11 = 14 (JR: +3)	The contents of the flag register are stored in the main register pair $ C5, $ C5 + 1 specified by the first operand. At this time, the same data is stored in $ C5 and $ C5 + 1. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	GFLW $ 2; GFLW $ 2, LABEL; Jump expansion
GPOW (Get Port Word) undisclosed instruction	GPOW ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← Port $ C5 + 1 ← Port	It does not change	3 + 11 = 14 (JR: +3)	The contents of the port terminal are stored in the main register pair $ C5, $ C5 + 1 specified by the first operand. The register pair $ C5, $ C5 + 1 contains the same data. If there is a label for the second operand, a relative jump is made after the transfer. (JR tag can be omitted)	GPOW $ 2; GPOW $ 2, LABEL; Jump expansion
PSRW (Put Specific index Register Word) undisclosed instruction	PSRW ***SIR*** , ***$ C5*** [, (JR) ***LABEL***]	SIR ← $ C5 SIR ← $ C5 + 1	It does not change	3 + 11 = 14 (JR: +3)	The contents of the main register pair $ C5, $ C5 + 1 of the second operand are stored in the specific index register SIR designated by the first operand . **However, only $ C5 (lower 5 bits) is stored in the SIR.** SIR = SX, SY, SZ. If there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted) If **this command is used to change the SIR setting (usually fixed at SX = 31, SY = 30, SZ = 0) and control is returned to the system, it will run out of control.**	PSRW SX, $ 2; Main register PSRW SY, $ 2, LABEL; Jump expansion EU format PRAW # 1, $ 4; Main register PRAW # 2, $ 4, J.LABEL; Jump expansion

					When users change SIR, the following cautions are required. (1) Disable interrupts while changing SIR. (2) When returning to ROM processing or calling ROM processing, return SIR to its original setting. (3) Coding the optimization switch with OFF (LEVEL 0) specified. (Because it is optimized at $ 31, $ 30, and $ 0 at LEVEL 1, the code gets confused.)	
GSRW (Get Specific index Register Word) undisclosed instruction	GSRW ***SIR*** , ***$ C5*** [, (JR) ***LABEL***]	SIR → $ C5 SIR + 1 → $ C5 + 1	It does not change	3 + 11 = 14 (JR: +3)	The specific index registers SIR and SIR + 1 designated by the first operand are stored in the main register pair $ C5 and $ C5 + 1 of the second operand. SIR = SX, SY, SZ. If there is a label for the third operand, a relative jump is made after the transfer. (JR tag can be omitted) For example, in GSRW SY, $ 2, if SY = 30, $ 2 = 30 and $ 3 = 31 are stored. When SY = 31, $ 2 = 31 and $ 3 = 0 are stored.	GSRW SX, $ 2; GSRW SY, $ 2, LABEL; Jump expansion EU format GRAW # 2, $ 2; GRAW # 0, $ 2, J.LABEL; Jump expansion

Mnemonic Table - Arithmetic operation instruction (8 bits)

	Operand formats not described in the format examples are not supported. To be precise, INV and CMP are classified into shift instruction groups, but arithmetic instructions are easier to understand.					
Mnemonic	**Format**	**Function**	**Flag**	**Number of Clocks**	**Description**	**Example Format**
INV (Invert)	INV ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← & HFF-$ C5	Z, C = 1, LZ, UZ change	3 + 6 = 9 (JR: +3)	Bit-inverts the contents of the main register specified by the first operand (1's complement). If there is a label for the	INV $ 2; INV $ 2, LABEL; Jump expansion

					second operand, a relative jump is made after the operation. (JR tag can be omitted)	
CMP (Complement)	CMP ***$ C5*** [, (JR) ***LABEL***]	$ C5 ← 2 ^ 8- $ C5	Z, C, LZ, UZ change	3 + 6 = 9 (JR: +3)	1 is added to the contents of the main register specified by the first operand after bit inversion (2's complement). If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	CMP $ 2; CMP $ 2, LABEL; Jump expansion
AD (Add)	AD ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A + B	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 6 = 9 (A, B) = ($ C5, $ C5): 3 + 3 + 6 = 12 (A, B) = ($ C5, IM8): 3 + 3 + 6 = 12 (JR: +3) A = (IR ± SIR): 3 + 6 + 3 + 3 = 15 A = (IR ± $ C5): 3 + 3 + 6 + 3 + 3 = 18 A = (IR ± $ IM8): 3 + 3 + 6 + 3 + 3 = 18	The result of adding the value of the first operand A and the value of the second operand B is stored in A. For operations other than external memory, a relative jump is performed after the operation according to the description in the label of the third operand. (JR tag can be omitted)	AD $ 4, $ 2; Main registers AD $ 4, $ 2, LABEL; Main registers (Jump expansion) AD $ 4, $ SZ; Indirect designation by main register + SIR AD $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion) AD $ 4,123; Main register + IM8 AD $ 4,123, LABEL; Main letter + IM8 (Jump expansion) AD (IX + $ 4), $ 2; External memory (1) + Main register → External memory AD (IX- $ SZ), $ 2; External memory (indirect designation by SIR) + main register → external memory AD (IZ + 123), $ 2; External

						memory (2) + Main register → External memory
SB (Subtract)	SB ***A*** , ***B*** [, (JR) ***LABEL***]	A ← AB	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 6 = 9 (A, B) = ($ C5, $ C5): 3 + 3 + 6 = 12 (A, B) = ($ C5, IM8): 3 + 3 + 6 = 12 (JR: +3) A = (IR ± SIR): 3 + 6 + 3 + 3 = 15 A = (IR ± $ C5): 3 + 3 + 6 + 3 + 3 = 18 A = (IR ± $ IM8): 3 + 3 + 6 + 3 + 3 = 18	The result of subtracting the value of the second operand B from the value of the first operand A is stored in A. For operations other than external memory, a relative jump is performed after the operation according to the description in the label of the third operand. (JR tag can be omitted)	SB $ 4, $ 2; Main registers SB $ 4, $ 2, LABEL; Main registers (Jump expansion) SB $ 4, $ SZ; Indirect designation by main register-SIR SB $ 4, $ SZ, LABEL; Main register-Indirect specification with SIR (Jump extension) SB $ 4,123; Main register-IM8 SB $ 4,123, LABEL; Main letter-IM8 (Jump expansion) SB (IX + $ 4), $ 2; External memory (1) -Main register → External memory SB (IX- $ SZ), $ 2; External memory (Indirect designation by SIR)-Main register → External memory SB (IZ + 123), $ 2; External memory (2)-Main register → External memory
ADB (Add BCD)	ADB ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A + B (BCD calculation)	Z, C, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The result of BCD addition of the value of the first operand A and the value of the second operand B is stored in A. The BCD format is a decimal number in which the upper 4 bits are the 10's place and	ADB $ 4, $ 2; Main registers ADB $ 4, $ 2, LABEL; Main registers (Jump extension) ADB $ 4, $ SZ; Indirect designation by

					the lower 4 bits are the 1's place. Relative jump is performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	main register + SIR ADB $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion) ADB $ 4, & H12; Main register + IM8. & H12 (18) is BCD decimal number 12. ADB $ 4, & H12, LABEL; Main letter + IM8 (Jump expansion)
SBB (Subtract BCD)	SBB ***A*** , ***B*** [, (JR) ***LABEL***]	A ← AB (BCD calculation)	Z, C, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The result of BCD subtraction of the value of the second operand B from the value of the first operand A is stored in A. The BCD format is a decimal number in which the upper 4 bits are the 10's place and the lower 4 bits are the 1's place. Relative jump is performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	SBB $ 4, $ 2; Main registers SBB $ 4, $ 2, LABEL; Main registers (Jump expansion) SBB $ 4, $ SZ; Main register-Indirect specification with SIR SBB $ 4, $ SZ, LABEL; Main register-Indirect specification with SIR (Jump extension) SBB $ 4, & H12; Main register-IM8. & H12 (18) is BCD decimal number 12. SBB $ 4, & H12, LABEL; Main letter-IM8 (Jump expansion)
ADC (Add Check)	ADC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A + B)	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 6 = 9 (A, B) = ($ C5, $ C5): 3 + 3 + 6 = 12 (A, B) = ($ C5, IM8):	Adds the value of the first operand A and the value of the second operand B, but does not store the result anywhere, only the flag changes. For operations other than external memory,	ADC $ 4, $ 2; Main registers ADC $ 4, $ 2, LABEL; Main registers (Jump expansion) ADC $ 4, $ SZ; Indirect designation by

				3 + 3 + 6 = 12 (JR: +3) A = (IR ± SIR): 3 + 6 + 6 = 15 A = (IR ± $ C5): 3 + 3 + 6 + 6 = 18 A = (IR ± $ IM8): 3 + 3 + 6 + 3 + 3 = 18	a relative jump is performed after the operation according to the description in the label of the third operand. (JR tag can be omitted)	main register + SIR ADC $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion) ADC $ 4,123; Main register + IM8 ADC $ 4,123, LABEL; Main letter + IM8 (Jump expansion) ADC (IX + $ 4), $ 2; External memory (1) + Main register ADC (IX- $ SZ), $ 2; External memory (indirect specification by SIR) + main register ADC (IZ + 123), $ 2; External memory (2) + Main register
SBC (Subtract Check)	SBC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← AB)	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 6 = 9 (A, B) = ($ C5, $ C5): 3 + 3 + 6 = 12 (A, B) = ($ C5, IM8): 3 + 3 + 6 = 12 (JR: +3) A = (IR ± SIR): 3 + 6 + 6 = 15 A = (IR ± $ C5): 3 + 3 + 6 + 6 = 18 A = (IR ± $ IM8): 3 + 3 + 6 + 3 + 3 = 18	Subtracts the value of the second operand B from the value of the first operand A, but does not store the result anywhere, only the flag changes. For operations other than external memory, a relative jump is performed after the operation according to the description in the label of the third operand. (JR tag can be omitted)	SBC $ 4, $ 2; Main registers SBC $ 4, $ 2, LABEL; Main registers (Jump expansion) SBC $ 4, $ SZ; Indirect designation by main register-SIR SBC $ 4, $ SZ, LABEL; Main register-Indirect specification with SIR (Jump extension) SBC $ 4,123; Main register-IM8 SBC $ 4,123, LABEL; Main letter-IM8 (Jump expansion)

						SBC (IX + $ 4), $ 2; External memory (1) - Main register SBC (IX- $ SZ), $ 2; External memory (indirect specification by SIR) -Main register SBC (IZ + 123), $ 2; External memory (2) - Main register
AN (And)	AN ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A and B	Z, C = 0, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The result of the logical product (AND) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	AN $ 4, $ 2; Main registers AN $ 4, $ 2, LABEL; Main registers (Jump expansion) AN $ 4, $ SZ; Indirect designation by main register and SIR AN $ 4, $ SZ, LABEL; Indirect specification by main register and SIR (Jump expansion) AN $ 4,123; Main register and IM8 AN $ 4,123, LABEL; Main letter and IM8 (Jump expansion)
ANC (And Check)	ANC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A and B)	Z, C = 0, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	Performs a logical AND of the values of the first operand A and the second operand B, but does not store the result anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ANC $ 4, $ 2; Main registers ANC $ 4, $ 2, LABEL; Main registers (Jump extension) ANC $ 4, $ SZ; Indirect designation by main register and SIR ANC $ 4, $ SZ, LABEL; Indirect specification by main register and

						SIR (Jump extension) ANC $ 4,123; Main register and IM8 ANC $ 4,123, LABEL; Main letter and IM8 (Jump expansion)
NA (Nand)	NA ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A nand B	Z, C = 1, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The result of NAND (AND bit inversion) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	NA $ 4, $ 2; Main registers NA $ 4, $ 2, LABEL; Main registers (Jump expansion) NA $ 4, $ SZ; Indirect designation by main register nand SIR NA $ 4, $ SZ, LABEL; Main register nand SIR indirect specification (Jump extension) NA $ 4,123; Main register nand IM8 NA $ 4,123, LABEL; Main letter nand IM8 (Jump expansion)
NAC (Nand Check)	NAC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A nand B)	Z, C = 1, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	NAND of the value of the first operand A and the value of the second operand B (bit inversion of AND), but the result is not stored anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	NAC $ 4, $ 2; Main registers NAC $ 4, $ 2, LABEL; Main registers (Jump extension) NAC $ 4, $ SZ; Indirect specification by main register nand SIR NAC $ 4, $ SZ, LABEL; Indirect specification by main register nand SIR (Jump expansion)

						NAC $ 4,123; Main register nand IM8 NAC $ 4,123, LABEL; Main letter nand IM8 (Jump expansion)
OR (Or)	OR ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A or B	Z, C = 1, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The logical sum (OR) result of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	OR $ 4, $ 2; Main registers OR $ 4, $ 2, LABEL; Main registers (Jump extension) OR $ 4, $ SZ; Indirect designation by main register or SIR OR $ 4, $ SZ, LABEL; Indirect specification by main register or SIR (Jump expansion) OR $ 4,123; Main register or IM8 OR $ 4,123, LABEL; Main letter or IM8 (Jump expansion)
ORC (Or Check)	ORC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A or B)	Z, C = 1, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	ORs the value of the first operand A and the value of the second operand B, but does not store the result anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ORC $ 4, $ 2; Main registers ORC $ 4, $ 2, LABEL; Main registers (Jump extension) ORC $ 4, $ SZ; Indirect specification by main register or SIR ORC $ 4, $ SZ, LABEL; Indirect specification by main register or SIR (Jump expansion) ORC $ 4,123; Main register or IM8 ORC $ 4,123, LABEL; Main

						letter or IM8 (Jump expansion)
XR (Exclusive Or)	XR ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A xor B	Z, C = 0, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	The result of the exclusive OR (OR) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	XR $ 4, $ 2; Main registers XR $ 4, $ 2, LABEL; Main registers (Jump extension) XR $ 4, $ SZ; Indirect designation by main register xor SIR XR $ 4, $ SZ, LABEL; Indirect specification by main register xor SIR (Jump extension) XR $ 4,123; Main register xor IM8 XR $ 4,123, LABEL; Main letter xor IM8 (Jump expansion)
XRC (Exclusive Or Check)	XRC ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A xor B)	Z, C = 0, LZ, UZ change	B = SIR: 3 + 6 = 9 B = $ C5, IM8: 3 + 3 + 6 = 12 (JR: +3)	XOR is performed on the value of the first operand A and the value of the second operand B, but the result is not stored anywhere and only the flag changes. A = $ C5. B = $ C5, $ SIR, IM8. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	XRC $ 4, $ 2; Main registers XRC $ 4, $ 2, LABEL; Main registers (Jump extension) XRC $ 4, $ SZ; Main register xor SIR indirect specification XRC $ 4, $ SZ, LABEL; Indirect specification by main register xor SIR (Jump extension) XRC $ 4,123; Main register xor IM8 XRC $ 4,123, LABEL; Main letter xor IM8 (Jump expansion)

Mnemonic Table - Arithmetic operation instruction (16 bits)

Operand formats not described in the format examples are not supported.
The flag operation differs from 8-bit arithmetic as follows.
• Z: 0 when all 16 bits of the operation result are 0.
• Z: C: 1 when there is a carry or borrow from the most significant bit (bit 15).
• Z: LZ: 0 when the lower 4 bits of the upper 8 bits are 0.
• Z: UZ: 0 when the upper 4 bits of the upper 8 bits are 0.
To be precise, INVW and CMPW are classified into shift instruction groups, but arithmetic instructions are easier to understand.

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example format
INVW (Invert Word)	INVW ***$ C5*** [, (JR) ***LABEL***]	($ C5 + 1, $ C5) ← & HFFFF-($ C5 + 1, $ C5)	Z, C = 1, LZ, UZ change	3 + 11 = 14 (JR: +3)	Bit-inverts the contents of the main register pair specified by the first operand (1's complement). If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	INVW $ 2; INVW $ 2, LABEL; Jump expansion
CMPW (Complement Word)	CMPW ***$ C5*** [, (JR) ***LABEL***]	($ C5 + 1, $ C5) ← 2 ^ 16-($ C5 + 1, $ C5)	Z, C, LZ, UZ change	3 + 11 = 14 (JR: +3)	1 is added to the contents of the main register pair specified by the first operand after bit inversion (2's complement). If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	CMPW $ 2; CMPW $ 2, LABEL; Jump expansion
ADW (Add Word)	ADW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A + B	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 11 = 14 (A, B) = ($ C5, $ C5): 3 + 3 + 11 = 17 (JR: +3) A = (IR ± SIR): 3 + 6 + 3 + 3 + 3 + 3 = 21 A = (IR ± $ C5): 3 + 3 + 6 + 3 + 3 + 3 + 3 = 24	The result of adding the value of the first operand A and the value of the second operand B is stored in A. Almost the same as 8-bit operation, except that the operation is performed with 16 bits. Only in the case of operations between main registers (including indirect specification by SIR), a relative jump is made after the operation	ADW $ 4, $ 2; Main registers ADW $ 4, $ 2, LABEL; Main registers (Jump expansion) ADW $ 4, $ SZ; Indirect designation by main register + SIR ADW $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion)

					according to the description of the label of the third operand. (JR tag can be omitted)	ADW (IX + $ 4), $ 2; External memory + Main register → External memory ADW (IX- $ SZ), $ 2; External memory (indirect designation by SIR) + main register → external memory
SBW (Subtract Word)	SBW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← AB	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 11 = 14 (A, B) = ($ C5, $ C5): 3 + 3 + 11 = 17 (JR: +3) A = (IR ± SIR): 3 + 6 + 3 + 3 + 3 + 3 = 21 A = (IR ± $ C5): 3 + 3 + 6 + 3 + 3 + 3 + 3 = 24	The result of subtracting the value of the second operand B from the value of the first operand A is stored in A. Almost the same as 8-bit operation, except that the operation is performed with a 16-bit pair register. Only in the case of operations between main registers (including indirect specification by SIR), a relative jump is made after the operation according to the description of the label of the third operand. (JR tag can be omitted)	SBW $ 4, $ 2; Main registers SBW $ 4, $ 2, LABEL; Main registers (Jump expansion) SBW $ 4, $ SZ; Indirect designation by main register-SIR SBW $ 4, $ SZ, LABEL; Main register-Indirect specification with SIR (Jump extension) SBW (IX + $ 4), $ 2; External memory-Main register → External memory SBW (IX- $ SZ), $ 2; External memory (Indirect designation by SIR)-Main register → External memory
ADBW (Add BCD Word)	ADBW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A + B (BCD calculation)	Z, C, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The result of BCD addition of the value of the first operand A and the value of the second operand B is stored in A. The BCD format is a decimal number in which the upper 4 bits of $ C5 + 1 are in the thousands, the lower 4 bits are in the 100s, the	ADBW $ 4, $ 2; Main registers ADBW $ 4, $ 2, LABEL; Main registers (Jump extension) ADBW $ 4, $ SZ; Indirect designation with main register + SIR

					upper 4 bits of $ C5 are in the 10s, and the lower 4 bits are in the 1s. Almost the same as 8-bit operation, except that the operation is performed with a 16-bit pair register. Relative jump is performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	ADBW $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion)
SBBW (Subtract BCD Word)	SBBW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← AB (BCD calculation)	Z, C, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The result of BCD subtraction of the value of the second operand B from the value of the first operand A is stored in A. Almost the same as 8-bit operation except that the operation is performed in a 16-bit pair register. Relative jump is performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	SBBW $ 4, $ 2; Main registers SBBW $ 4, $ 2, LABEL; Main registers (Jump expansion) SBBW $ 4, $ SZ; Main register-Indirect specification with SIR SBBW $ 4, $ SZ, LABEL; Indirect specification by main register-SIR (Jump extension)
ADCW (Add Check Word)	ADCW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A + B)	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 11 = 14 (A, B) = ($ C5, $ C5): 3 + 3 + 11 = 17 (JR: +3) A = (IR ± SIR): 3 + 6 + 6 + 6 = 21 A = (IR ± $ C5): 3 + 3 + 6 + 6 + 6 = 24	Adds the value of the first operand A and the value of the second operand B, but does not store the result anywhere, only the flag changes. Almost the same as 8-bit operation except that the operation is performed in a 16-bit pair register. Only in the case of operations between main registers (including indirect specification by SIR), a relative jump is	ADCW $ 4, $ 2; Main registers ADCW $ 4, $ 2, LABEL; Main registers (Jump expansion) ADCW $ 4, $ SZ; Indirect designation by main register + SIR ADCW $ 4, $ SZ, LABEL; Indirect specification with main register + SIR (Jump expansion)

					performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	ADCW (IX + $ 4), $ 2; External memory + Main register ADCW (IX- $ SZ), $ 2; External memory (indirect specification by SIR) + main register
SBCW (Subtract Check Word)	SBCW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← AB)	Z, C, LZ, UZ change	(A, B) = ($ C5, SIR): 3 + 11 = 14 (A, B) = ($ C5, $ C5): 3 + 3 + 11 = 17 (JR: +3) A = (IR ± SIR): 3 + 6 + 6 + 6 = 21 A = (IR ± $ C5): 3 + 3 + 6 + 6 + 6 = 24	Subtracts the value of the second operand B from the value of the first operand A, but does not store the result anywhere, only the flag changes. Almost the same as 8-bit operation, except that the operation is performed with a 16-bit pair register. Only in the case of operations between main registers (including indirect specification by SIR), a relative jump is performed after the operation according to the description of the label of the third operand. (JR tag can be omitted)	SBCW $ 4, $ 2; Main registers SBCW $ 4, $ 2, LABEL; Main registers (Jump expansion) SBCW $ 4, $ SZ; Main register-Indirect specification with SIR SBCW $ 4, $ SZ, LABEL; Indirect specification by main register-SIR (Jump extension) SBCW (IX + $ 4), $ 2; External memory-Main register SBCW (IX- $ SZ), $ 2; External memory (indirect specification by SIR) -Main register
ANW (And Word)	ANW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A and B	Z, C = 0, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The result of the logical product (AND) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ANW $ 4, $ 2; Main registers ANW $ 4, $ 2, LABEL; Main registers (Jump expansion) ANW $ 4, $ SZ; Indirect designation by main register and SIR ANW $ 4, $ SZ, LABEL; Indirect specification with main register and

						SIR (Jump expansion) ANW $ 4,123; Main register and IM8 ANW $ 4,123, LABEL; Main letter and IM8 (Jump expansion)
ANCW (And Check Word)	ANCW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A and B)	Z, C = 0, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	Performs a logical AND of the values of the first operand A and the second operand B, but does not store the result anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ANC $ 4, $ 2; Main registers ANC $ 4, $ 2, LABEL; Main registers (Jump extension) ANC $ 4, $ SZ; Indirect designation by main register and SIR ANC $ 4, $ SZ, LABEL; Indirect specification by main register and SIR (Jump extension)
NAW (Nand Word)	NAW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A nand B	Z, C = 1, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The result of NAND (AND bit inversion) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	NAW $ 4, $ 2; Main registers NAW $ 4, $ 2, LABEL; Main registers (Jump extension) NAW $ 4, $ SZ; Indirect designation by main register nand SIR NAW $ 4, $ SZ, LABEL; Indirect specification by main register nand SIR (Jump expansion)
NACW (Nand Check Word)	NACW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A nand B)	Z, C = 1, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	NAND of the value of the first operand A and the value of the second operand B (bit inversion of AND), but the result is not stored anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR.	NACW $ 4, $ 2; Main registers NACW $ 4, $ 2, LABEL; Main registers (Jump extension) NACW $ 4, $ SZ; Indirect

					If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	specification by main register nand SIR NACW $ 4, $ SZ, LABEL; Indirect specification by main register nand SIR (Jump expansion)
ORW (Or Word)	ORW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A or B	Z, C = 1, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The logical sum (OR) result of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ORW $ 4, $ 2; Main registers ORW $ 4, $ 2, LABEL; Main registers (Jump extension) ORW $ 4, $ SZ; Indirect specification by main register or SIR ORW $ 4, $ SZ, LABEL; Indirect specification by main register or SIR (Jump expansion)
ORCW (Or Check Word)	ORCW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A or B)	Z, C = 1, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5, IM8: 3 + 3 + 11 = 17 (JR: +3)	ORs the value of the first operand A and the value of the second operand B, but does not store the result anywhere, only the flag changes. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	ORCW $ 4, $ 2; Main registers ORCW $ 4, $ 2, LABEL; Main registers (Jump extension) ORCW $ 4, $ SZ; Indirect designation by main register or SIR ORCW $ 4, $ SZ, LABEL; Indirect specification by main register or SIR (Jump expansion)
XR (Exclusive Or Word)	XRW ***A*** , ***B*** [, (JR) ***LABEL***]	A ← A xor B	Z, C = 0, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	The result of the exclusive OR (OR) of the value of the first operand A and the value of the second operand B is stored in A. A = $ C5. B = $ C5, $ SIR.	XR $ 4, $ 2; Main registers XR $ 4, $ 2, LABEL; Main registers (Jump extension) XR $ 4, $ SZ; Indirect

					If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	designation by main register xor SIR XR $ 4, $ SZ, LABEL; Indirect specification by main register xor SIR (Jump extension)
XRCW (Exclusive Or Check Word)	XRCW ***A*** , ***B*** [, (JR) ***LABEL***]	(A ← A xor B)	Z, C = 0, LZ, UZ change	B = SIR: 3 + 11 = 14 B = $ C5: 3 + 3 + 11 = 17 (JR: +3)	XOR is performed on the value of the first operand A and the value of the second operand B, but the result is not stored anywhere and only the flag changes. A = $ C5. B = $ C5, $ SIR. If there is a label for the third operand, a relative jump is made after the operation. (JR tag can be omitted)	XRC $ 4, $ 2; Main registers XRC $ 4, $ 2, LABEL; Main registers (Jump extension) XRC $ 4, $ SZ; Main register xor SIR indirect specification XRC $ 4, $ SZ, LABEL; Indirect specification by main register xor SIR (Jump extension)

Rotate shift instruction (8 bits)

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
ROU (Rotate Up)	ROU ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 6 = 9 (JR: +3)	Rotate left between the main register specified by the first operand and the carry flag. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	ROU $ 2; ROU $ 2, LABEL; Jump expansion
ROD (Rotate Down)	ROD ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 6 = 9 (JR: +3)	Rotate right between the main register specified by the first operand and the carry flag. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	ROD $ 2; ROD $ 2, LABEL; Jump expansion

BIU (Bit Up)	BIU ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 6 = 9 (JR: +3)	The contents of the main register specified by the first operand are incremented 1 bit to the left, 0 is stored in the least significant bit, and the carry is stored in the carry. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BIU $ 2; BIU $ 2, LABEL; Jump expansion
BID (Bit Down)	BID ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 6 = 9 (JR: +3)	The contents of the main register specified by the first operand are moved down 1 bit to the right, the most significant bit is set to 0, and the carry is stored in the carry. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BID $ 2; BID $ 2, LABEL; Jump expansion
DIU (Digit Up)	DIU ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ = 0, UZ changes	3 + 6 = 9 (JR: +3)	The contents of the main register specified by the first operand are raised 4 bits to the left, and 0 is placed in the lower bits. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	DIU $ 2; DIU $ 2, LABEL; Jump expansion
DID (Digit Down)	DID ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ, UZ = 0 change	3 + 6 = 9 (JR: +3)	Decreases the contents of the main register specified by the first operand by 4 bits to the right and puts 0 in the upper bits. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	DID $ 2; DID $ 2, LABEL; Jump expansion
BYU (Byte Up)	BYU ***$ C5*** [,	See figure	Z = 0, C = 0, LZ,	3 + 6 = 9 (JR: +3)	0 is stored in the main register specified by the first operand.	BYU $ 2; BYU $ 2, LABEL; Jump expansion

undisclosed instruction	(JR) ***LABEL***]		UZ change		If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	
BYD (Byte Down) undisclosed instruction	BYD ***$ C5*** [, (JR) ***LABEL***]	See figure	Z = 0, C = 0, LZ, UZ change	3 + 6 = 9 (JR: +3)	0 is stored in the main register specified by the first operand. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BYD $ 2; BYD $ 2, LABEL; Jump expansion

Rotate shift instruction (16 bits)

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
ROUW (Rotate Up Word)	ROUW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 11 = 14 (JR: +3)	16-bit left rotation is performed between the main register pair ($ C5 + 1, $ C5) specified by the first operand and the carry flag. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	ROUW $ 2; ROUW $ 2, LABEL; Jump expansion
RODW (Rotate Down Word)	RODW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 11 = 14 (JR: +3)	16-bit right rotation is performed between the main register pair ($ C5, $ C5-1) specified by the first operand and the carry flag. Note that the register pair is $ C5, $ C5-1. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	RODW $ 2; RODW $ 2, LABEL; Jump expansion
BIUW (Bit Up Word)	BIUW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5 + 1, $ C5) specified by the first operand are incremented 1 bit to the left, 0 is placed in the least significant bit, and the carry is stored in the carry.	BIUW $ 2; Register pair is ($ 3, $ 2), $ 2 is lower byte. BIUW $ 2, LABEL; Jump expansion

					If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	
BIDW (Bit Down Word)	BIDW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5, $ C5-1) specified by the first operand are lowered 1 bit to the right, 0 is placed in the most significant bit, and the carry is stored in the carry. Note that the register pair is $ C5, $ C5-1. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BIDW $ 2; The register pair is ($ 2, $ 1), and $ 2 is the upper byte. BIDW $ 2, LABEL; Jump expansion
DIUW (Digit Up Word)	DIUW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5 + 1, $ C5) specified by the first operand are raised 4 bits to the left, and 0 is placed in the lower bits. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	DIUW $ 2; Regist pair is ($ 3, $ 2), $ 2 is the lower byte. DIUW $ 2, LABEL; Jump expansion
DIDW (Digit Down Word)	DIDW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5, $ C5-1) specified by the first operand are lowered 4 bits to the right and 0 is placed in the upper bits. Note that the register pair is $ C5, $ C5-1. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	DIDW $ 2; Register pair is ($ 2, $ 1), $ 2 is the upper byte. DIDW $ 2, LABEL; Jump expansion
BYUW (Byte Up Word)	BYUW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5 + 1, $ C5) specified by the first operand are	BYUW $ 2; Regist pair is ($ 3, $ 2), $ 2 is the lower byte.

					increased 8 bits to the left, and all lower bytes are set to 0. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BYUW $ 2, LABEL; Jump expansion
BYDW (Byte Down Word)	BYDW ***$ C5*** [, (JR) ***LABEL***]	See figure	Z, C = 0, LZ, UZ change	3 + 11 = 14 (JR: +3)	The contents of the main register pair ($ C5, $ C5-1) specified by the first operand are down 8 bits to the right, and all upper bytes are set to 0. Note that the register pair is $ C5, $ C5-1. If there is a label for the second operand, a relative jump is made after the operation. (JR tag can be omitted)	BYDW $ 2; Register pair is ($ 2, $ 1), $ 2 is upper byte. BYDW $ 2, LABEL; Jump expansion

Mnemonic Table - Jump / call instructions

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
JP (Jump)	JP { ***IM16*** | ***LABEL*** }	PC ← IM16	No change	3 + 3 + 6 = 12	The 16-bit immediate value of the first operand is taken into the program counter (PC), and jumps to that address.	JP & H703F; Unconditional jump
JP (Jump flag)	JP ***Flag*** , { ***IM16*** | ***LABEL*** }	If Flag then PC ← IM16	No change	3 + 3 + 6 = 12	When the condition of the flag register of the first operand is satisfied, the 16-bit immediate value of the second operand is taken into the program counter (PC) and jumped to that address.	JP Z, & H703F; Jump if Z = 0 (calculation result is 0) JP NZ, & H703F; Jump if Z = 1 (calculation result is other than 0) JP C, LABEL; C = 1 (carry occurrence) jump JP NC, LABEL; Jump if C = 0 (no carry) JP LZ, & H703F; Jump when

						lower digit flag is 0 (lower 4 bits are 0) JP UZ, & H703F; Jump when upper digit flag is 0 (upper 4 bits are 0) JP NLZ, LABEL; Jump when lower digit flag is 1 (flag name can be written in LNZ)
JP (Jump register) undisclosed instruction	JP ***$ C5***	PC ← $ C5	No change	3 + 8 = 11	The value of the main register pair specified by the first operand is taken into the program counter (PC) and jumped to that address. Note Although this instruction (opcode DEH) has been written as "JP ($ C5)", Mr. Piotr Piatek used a jump instruction (opcode DFH) by indirect memory addressing ($ C5) by the main register. Because it was found, it was changed to the current "JP $ C5" notation.	JP $ 17; EU format JPW $ 17;
JP (indirect Jump register) unpublished instruction	JP (***$ C5***)	PC ← ($ C5)	No change	3 + 8 = 11	Indirect designation with the main register pair $ C5 of the first operand, that is, the 16-bit data stored in the external memory with the address ($ C5 + 1, $ C5) is taken into the program counter (PC) and the address is Jump. In JP ($ 17), if $ 17 = 00, $ 18 = & H70, memory address & H7000 = & H34, & H7001 = & H20, the program jumps to & H2034.	JP ($ 17); EU format JPW ($ 17);

JR (Relative Jump)	JR { ***± IM7*** | ***LABEL*** }	PC ← PC ± IM7	No change	3 + 6 = 9	Adds or subtracts the 7-bit immediate value of the operand to the program counter (PC) and performs a relative jump. Specify a numeric value ± IM7 (0 to 127) or a label for the operand.	JR +32; + IM7 JR -32; -IM7 JR LABEL; LABEL specified
JR (Relative Jump flag)	JR ***Flag*** , { ***± IM7*** | ***LABEL*** }	If Flag then PC ← PC ± IM7	No change	3 + 6 = 9	When the flag condition of the first operand is satisfied, the 7-bit immediate value of the second operand is added to or subtracted from the program counter (PC), and a relative jump is made. For the second operand, specify a numeric value ± IM7 (0 to 127) or a label.	JR Z, LABEL; Jump if Z = 0 (result is 0) JR NZ, LABEL; Jump if Z = 1 (calculation result is not 0) JR C, LABEL; Jump if C = 1 (carry occurs) JR NC, LABEL; Jump if C = 0 (no carry) JR LZ, LABEL; Jump when lower digit flag is 0 (lower 4 bits are 0) JR UZ, LABEL; Jump when upper digit flag is 0 (upper 4 bits are 0) JR NLZ, LABEL; Jump when lower digit flag is 1 (flag name can be written in LNZ) JR Z, + 32; If Z = 0 (result is 0), relative jump in + IM7 format
CAL (Call)	CAL { ***IM16*** | ***LABEL*** }	(SS-2) ← PC + 3 SS ← SS-2 PC ← IM16	No change	3 + 3 + 6 + 3 + 3 = 18	After the address of the next instruction is pushed to the system stack (SS), the 16-bit immediate value of the first operand is stored in the program counter (PC) and a subroutine call is made to that address.	CAL & H703F; Unconditional call

CAL (Call flag)	CAL ***Flag*** , { ***IM16*** | ***LABEL*** }	If Flag then (SS-2) ← PC + 3 SS ← SS-2 PC ← IM16	No change	CALL execution: 3 + 3 + 6 + 3 + 3 = 18 CALL not executed: 3 + 3 + 6 = 12	When the flag condition of the first operand is satisfied, a subroutine call is made to the address specified by the second operand.	CAL Z, & H703F; Call if Z = 0 (result is 0) CAL NZ, & H703F; Call if Z = 1 (calculation result is not 0) CAL C, & H703F; Call if C = 1 (carry occurs) CAL NC, & H703F; Call if C = 0 (no carry) CAL LZ, & H703F; Call if lower digit flag is 0 (lower 4 bits are 0) CAL UZ, & H703F; Call if upper digit flag is 0 (upper 4 bits are 0) CAL NLZ, & H703F; Call if lower digit flag is 1 (flag name can be written in LNZ)
RTN (Return)	RTN	PC ← (SS) SS ← SS + 2	No change	6 + 3 + 5 = 14	Stores the 16-bit immediate value of the system stack (SS) in the program counter (PC) and returns to that address.	RTN; Unconditional
RTN (Return flag)	RTN ***Flag***	If Flag then PC ← (SS) SS ← SS + 2	No change	No return: 6 RTN: 6 + 3 + 5 = 14	When the flag condition of the operand is satisfied, the 16-bit immediate value of the system stack (SS) is stored in the program counter (PC), and it returns to that address.	RTN Z; Return if Z = 0 (result is 0) RTN NZ; Return if Z = 1 (operation result is not 0) RTN C; Return if C = 1 (carry occurs) RTN NC; Returns if C = 0 (no carry) RTN LZ; Return if lower digit flag is 0 (lower 4 bits are 0) RTN UZ; If the upper digit flag is 0 (the upper 4

						bits are 0), return RTN NLZ; Return if lower digit flag is 1 (flag name can be written in LNZ)

Mnemonic Table - Block transfer / search instructions

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
BUP (Block Up)	BUP	See explanation	No change	? unknown	Transfers the memory block specified by IX register = transfer source start address and IY register = transfer source end address to the area where IZ = transfer destination start address. Since transfer is performed in ascending order from IX address to IY address, it must be used with IX <IY setting. Same operation as REP MOVSB when CLD is specified on X86.	BUP;
BDN (Block Down)	BDN	See explanation	No change	? unknown	Transfers the memory block specified by IX register = transfer source start address and IY register = transfer source end address to the area where IZ = transfer destination start address. Since transfer is performed in descending order from IX address to IY address, it is necessary to use IX> IY setting. Same operation as REP MOVSB when STD is specified on X86.	BDN;

SUP (Search Up)	SUP { ***$ C5*** | ***IM8*** }	See explanation	Z, C, LZ, UZ change	? unknown	The main register value or 8-bit immediate value specified by the first operand is searched within the memory block range specified by IX register = search start address and IY register = search end address. If there is, set Z = 0 (Z) and terminate the execution on the spot. If there is no corresponding data, the execution ends with Z = 1 (NZ) and IX = IY. Since the search is performed in ascending order from IX address to IY address, it must be used with IX <IY setting. Same operation as REPNZ SCASB when CLD is specified on X86.	SUP $ 2; Specify main register SUP 123; 8-bit immediate designation
SDN (Search Down)	SDN { ***$ C5*** | ***IM8*** }	See explanation	Z, C, LZ, UZ change	? unknown	The main register value or 8-bit immediate value specified by the first operand is searched within the memory block range specified by IX register = search start address and IY register = search end address. If there is, set Z = 0 (Z) and terminate the execution on the spot. If there is no corresponding data, the execution ends with Z = 1 (NZ) and IX = IY. Since the search is performed in descending order from IX address to IY address, it is necessary to use IX> IY setting. Same operation as REPNZ SCASB when STD is specified on X86.	SDN $ 2; Specify main register SDN 123; 8-bit immediate designation

BUPS (Block Up & Search) (undisclosed instruction)	BUPS ***IM8***	See explanation	Z, C, LZ, UZ change	? unknown	The memory block specified by IX register = transfer source start address and IY register = transfer source end address is transferred to the area where IZ = transfer destination start address. During transfer, when the transfer data is searched and the same data as IM8 in operand 1 is detected, the instruction execution ends at Z = 0 (Z) after the data is transferred. If there is no corresponding data, the execution ends with Z = 1 (NZ) and IX = IY. Since the search is performed in ascending order from IX address to IY address, it must be used with IX <IY setting.	BUPS & H20; EU format BUP & H20;
BDNS (Block Down & Search) (undisclosed order)	BDNS ***IM8***	See explanation	Z, C, LZ, UZ change	? unknown	The memory block specified by IX register = transfer source start address and IY register = transfer source end address is transferred to the area where IZ = transfer destination start address. During transfer, when the transfer data is searched and the same data as IM8 in operand 1 is detected, the instruction execution ends at Z = 0 (Z) after the data is transferred. If there is no corresponding data, the execution ends with Z = 1 (NZ) and IX = IY. Since the search is performed in descending order from IX address to IY address,	BDNS & H20; EU format BDN & H20;

					it is necessary to use IX> IY setting.	

Mnemonic Table - Block transfer / search instructions

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
NOP (No Operation)	NOP	PC ← PC + 1	No change	6	Just increment the program counter (PC), and nothing else.	NOP
CLT (Clear Timer)	CLT	TM ← 0	No change	6	Set all timer (TM register) counters to 0. Caution During the last 1/65536 seconds of the 60th second (when it changes from 59 to 0), the reset (clear 0) by the CLT instruction does not operate normally. Therefore, in order to perform the reset operation reliably, it is necessary to execute it twice with a delay so as to avoid the above period. [Example] CLT; First execution XRCM $ 0, $ 0,8; Delay processing CLT; Second execution (can be reset reliably by the first or second CLT)	CLT
FST (Fast mode)	FST	See explanation	No change	6	Use the system clock without dividing it. (High-speed operation mode) The system normally operates in high-speed mode.	FST
SLW (Slow mode)	SLW	See explanation	No change	6?	Use the system clock divided by 1/16. (Low Power mode) **Note that if you return to the system while executing the SLW instruction (low speed state), you will run out of control.** The LCD port clock	SLW

					frequency is not changed even in the low-speed mode, and the bus is confused during LCD access. BASIC seems to be able to maintain the low-speed mode unless LCD access occurs. In the PB-1000, when the system interrupt handling routine is executed, it is automatically reset to high-speed mode.	
OFF (OFF)	OFF	See explanation	APO bit ← SW bit (APO bit is cleared when the power is turned on)	6?	Turn off the VDD power supply of the internal logic. Executing this command changes the following register values. • PC = 0 • IX, IY, IZ = 0 • UA = 0 • IA = 0 However, KO1 pin (BRK key input signal) is selected. • IE = Bits 0, 1, 5, 6, and 7 are cleared to 0. Only the following interrupts are valid. • Power ON control by 1-minute timer (depending on the state of bit 5 of the IB register) • Power on by power switch ON event. Or, power is turned on by BRK key event when SW is ON.	OFF
TRP (TRaP)	TRP	See explanation	No change	6?	When the TRP instruction (& HFF) is fetched, the address	TRP

					where the TRP instruction is written is saved in the SS stack, and the process from the fixed address (& H6FFA for PB-1000) is executed. Execution returns from the address following the TRP instruction by the RTN instruction.	
CANI (CANcel Interrupt)	CANI	See explanation	No change	6?	Of the hardware interrupt request latches, the one with the highest priority is cleared.	CANI
RTNI (ReTurN from Interrupt)	RTNI	See explanation	No change	6 + 3 + 5 = 14?	Return from interrupt processing. Store the contents of the system stack (SS) in the program counter (PC), return to that address, and add 2 to the system stack (SS). When this processing is executed, the corresponding interrupt status flag in the IB register (Bit4 to Bit0) is cleared to zero.	RTNI

Mnemonic Table - Multibyte transfer instruction (2 to 8 bytes) not disclosed

This instruction group expands the target register pair to 2 to 8 bytes by specifying operand 3 (operand 2 for PHUM, PHSM, PPUM, PPSM).
With this single command, data of up to 8 bytes (64 bits) can be transferred.
The number that can be specified for operand 3 (operand 2 for PHUM, PHSM, PPUM, PPSM) is 1 to 8. However, if a value smaller than 2 (= 1) is set, execution will be 2. HD61 is designed to output an error when 1 is specified.
Second, even when the specific index register SIR is used as an operand, neither the instruction code nor the operation clock is reduced.

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
LDM (LoaD Multi-byte register)	LDM ***$ C5*** , ***$ C5*** , ***IM3*** [, (JR) ***LABEL***]	opr1 @ $ C5 (IM3) ← opr2 @ $ C5 (IM3)	No change	3 + 3 + 11 + 5 * (IM3-2) = 17 + 5 * (IM3-2) (JR: +3)	Transfers the contents of the main register block starting from $ C5 of operand 1, starting from $ C5 of operand 1, starting with $ C5 of operand 2 and the number of IM3 bytes	LDM $ 0, $ 8,6; The contents of $ 8 to $ 13 are stored in $ 0 to $ 5. LDM $ 0, $ 8,6, JR LABEL; The contents of $ 8 to

					specified by operand 3. If the last operand has a label, a relative jump is made after the transfer. (JR tag can be omitted) For example, LDM $ 2, $ 6, 3 • $ 2 ← $ 6 • $ 3 ← $ 7 • $ 4 ← $ 8 And works.	$ 13 are stored in $ 0 to $ 5. (jump extension) LDM $ 0, $ SX, 6; Indirect designation by SIR LDM $ 0, $ SX, 6, JR LABEL; Indirect specification by SIR (jump extension) KC format LDW $ 0, $ 8 (6); The contents of $ 8 to $ 13 are stored in $ 0 to $ 5. LDW $ 0, $ 8 (6), JR LABEL; The contents of $ 8 to $ 13 are stored in $ 0 to $ 5. (jump extension) LDW $ 0, $ SX (6); Indirect designation by SIR LDW $ 0, $ SX (6), JR LABEL; Indirect specification by SIR (jump extension) EU format LDL $ 0, $ 8, L6; The contents of $ 8 to $ 13 are stored in $ 0 to $ 5. LDL $ 0, $ 8, L6, J.LABEL; The contents of $ 8 to $ 13 are stored in $ 0 to $ 5. (jump extension) LDL $ 0, # 0, L6; Indirect designation by SR LDL $ 0, # 0, L6, J.LABEL; Indirect specification by SR (jump extension)

LDM (LoaD Multi-byte memory)	LDM ***$ C5*** , (***IR*** ± ***$ C5***), ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← (IR ± $ C5) (IM3)	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2) (JR: +3)	Transfers the contents of consecutive external memory data for the number of IM3 bytes specified by operand 3 to the register block starting at $ C5 of operand 1, with (IR + $ C5) of operand 2 as the start address. For example, when IX = & H7000, $ 0 = 1, LDM $ 2, (IX + $ 0), 3 performs the following operation. • $ 2 ← (& H7001 memory contents) • $ 3 ← (& H7002 memory contents) • $ 4 ← (& H7003 memory contents) • IX ← & H7000 (no change)	LDM $ 0, (IX ± $ C5), IM3 LDM $ 0, (IZ ± $ C5), IM3 LDM $ 0, (IX ± $ SIR), IM3; Indirect designation by SIR LDM $ 0, (IZ ± $ SIR), IM3; Indirect designation by SIR KC format LDW $ 0, (IX ± $ C5) (IM3) LDW $ 0, (IZ ± $ C5) (IM3) LDW $ 0, (IX ± $ SIR) (IM3); Indirect designation by SIR LDW $ 0, (IZ ± $ SIR) (IM3); Indirect designation by SIR EU format LDL $ 0, (IX ± $ C5), IM3 LDL $ 0, (IZ ± $ C5), IM3 LDL $ 0, (IX ± # 0), IM3; Indirect designation by SR (# 0- # 2) LDL $ 0, (IZ ± # 0), IM3; Indirect specification by SR (# 0- # 2)
LDIM (LoaD Increment Multi byte)	LDIM ***$ C5*** , (***IR*** ± ***A***), ***IM3***	$ C5 (IM3) ← (IR ± A) (IM3) IR ← IR ± A + IM3	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2)	After storing the contents of external memory (IM3 byte) starting from (IR ± A) in the main register block starting at $ C5, add IR to ± A and IM3. A can be specified only for $ C5 (including indirect specification by	LDIM $ 4, (IX + $ 2), 6 LDIM $ 4, (IX- $ SX), 6; Indirect designation by SIR KC format LDIW $ 4, (IX + $ 2) (6)

					SIR). For example, when IX = & H7000, $ 0 = 1, LDIM $ 2, (IX + $ 0), 3 performs the following operation. • $ 2 ← (& H7001 memory contents) • $ 3 ← (& H7002 memory contents) • $ 4 ← (& H7003 memory contents) • IX ← & H7004 (last accessed address + 1 enters)	LDIW $ 4, (IX- $ SX) (6); Indirect designation by SIR EU format LDIL $ 4, (IX + $ 2), L6 LDIL $ 4, (IX- # 0), L6; Indirect designation by SR
LDDM (LoaD Decrement Multi byte)	LDDM ***$ C5*** , (***IR*** ± ***A***), ***IM3***	$ C5 (-IM3) ← (IR ± A) (-IM3) IR ← IR ± A-(IM3-1)	No change	3 + 3 + 6 + 3 + 3 + 3 * (IM3-2) = 18 + 3 * (IM3-2)	After storing the contents of external memory (IM3 byte) starting from (IR ± A) of operand 2 in main register block $ C5 to $ C5- (IM3-1) of operand 1, IR contains IR ± A Substitute-(IM3-1). A can be specified only for $ C5 (including indirect specification by SIR). **Note that LDDM differs from LDM and LDIM in that the transfer direction is the reverse direction (decrement direction).** For example, when IX = & H7000, $ 0 = 1, LDDM $ 3, (IX + $ 0), 3 performs the following operation. • $ 3 ← (& H7001 memory contents) • $ 2 ← (& H7000 memory contents) • $ 1 ← (& H6FFF memory contents)	LDDM $ 7, (IX + $ 2), 6 LDDM $ 7, (IZ- $ SX), 6; Indirect designation by SIR KC format LDMW $ 7, (IX + $ 2) (6) LDMW $ 7, (IZ- $ SX) (6); Indirect designation by SIR EU format LDDL $ 4, (IX + $ 2), L6 LDDL $ 4, (IZ- # 0), L6; Indirect designation by SR

					• IX ← & H6FFF (Enter the last accessed address)	
LDCM (LoaD Check Multi byte: undisclosed instruction)	LDCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	No Operation (Do nothing)	No change	3 + 3 + 11 + 5 * (IM3-2) = 17 + 5 * (IM3-2) (JR: +3)	Operands are specified in the same format as the LDM instruction, but nothing is actually processed and only instruction decoder operation (operation to advance the program counter after execution) is performed. Neither flag nor register contents are changed. (Delay processing is possible like the NOP instruction) A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, jump relative. (JR tag can be omitted)	LDCM $ 4, $ 2,6; Register specification LDCM $ 4, $ SX, 6; Indirect designation by SIR LDCM $ 2, $ 3,6, LABEL; Register specification + Jump expansion LDCM $ 4, $ SX, 6, LABEL; Indirect designation by SIR + Jump expansion KC format LDCW $ 4, $ 2 (6); Register specification LDCW $ 4, $ SX (6); Indirect designation by SIR LDCW $ 2, $ 3 (6), LABEL; Register specification + Jump expansion LDCW $ 4, $ SX (6), LABEL; Indirect designation by SIR + Jump expansion EU format LDCL $ 4, $ 2, L6; Register specification LDCL $ 4, # 0, L6; Indirect designation by SIR LDCL $ 2, $ 3, L6, LABEL; Register specification + Jump expansion

						LDCL $ 4, # 0, L6, LABEL; Indirect designation by SIR + Jump expansion
STM (STore Multi byte memory)	STM ***$ C5*** , (***IR*** ± ***A***), ***IM3***	$ C5 (IM3) → (IR ± A) (IM3)	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2)	Stores the contents of the main register block (IM3 byte) starting from $ C5 of operand 1 to the external memory at the address specified by operand 2. A can be specified only for $ C5 (including indirect specification by SIR). For example, when IX = & H7000, $ 0 = 1, STM $ 2, (IX + $ 0), 3 performs the following operations. • $ 2 → (memory at & H7001) • $ 3 → (memory at & H7002) • $ 4 → (memory at & H7003) • IX ← & H7000 (no change)	STM $ 4, (IX + $ 2), 6 STM $ 4, (IZ- $ SY), 6; Indirect designation by SIR KC format STW $ 4, (IX + $ 2) (6) STW $ 4, (IZ- $ SY) (6); Indirect designation by SIR EU format STL $ 4, (IX + $ 2), L6 STL $ 4, (IZ- # 1), L6; Indirect designation by SR
STIM (STore Increment Multi byte)	STIM ***$ C5*** , (***IR*** ± ***A***), ***IM3***	$ C5 (IM3) → (IR ± A) (IM3) IR ← IR ± A + IM3	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2)	Stores the contents of the main register block (IM3 byte) starting from $ C5 of operand 1 to the external memory at the address specified by operand 2. After data transfer, IR ± A + IM3 is assigned to IR. A can be specified only for $ C5 (including indirect specification by SIR). For example, when IX = & H7000, $ 0 = 1, STIM $ 2, (IX + $ 0), 3 performs the following operation. • $ 2 → (memory at & H7001) • $ 3 → (memory at & H7002)	STIM $ 4, (IX + $ 2), 6 STIM $ 4, (IZ- $ SY), 6; Indirect designation by SIR KC format STIW $ 4, (IX + $ 2) (6) STIW $ 4, (IZ- $ SY) (6); Indirect designation by SIR EU format STIL $ 4, (IX + $ 2), L6 STIL $ 4, (IZ- # 1), L6; Indirect designation by SR

					• $ 4 → (memory at & H7003) • IX ← & H7004 (last accessed address + 1)	
STDM (STore Decrement Multi byte)	STDM ***$ C5*** , (***IR*** ± ***A***), ***IM3***	$ C5 (-IM3) → (IR ± A) (-IM3) IR ← IR ± A-(IM3-1)	No change	3 + 3 + 6 + 3 + 3 + 3 * (IM3-2) = 18 + 3 * (IM3-2)	Store the contents of the main register block (IM3 byte) starting from $ C5 of operand 1 in the external memory with (IR ± A) as the start address, and then assign IR ± A- (IM3-1) to IR. **Note that the STDM transfer direction is the reverse direction (decrement direction) of STM and STIM.** A can be specified only for $ C5 (including indirect specification by SIR). For example, when IX = & H7000, $ 0 = 1, STDM $ 2, (IX + $ 0), 3 performs the following operation. • $ 2 → (memory at & H7001) • $ 3 → (& H7000 memory) • $ 4 → (memory at & H6FFF address) • IX ← & H6FFF (the last address accessed)	STDM $ 4, (IX + $ 2), 6 STDM $ 4, (IZ- $ SY), 6; Indirect designation by SIR KC format STMW $ 4, (IX + $ 2) (6) STMW $ 4, (IZ- $ SY) (6); Indirect designation by SIR EU format STDL $ 4, (IX + $ 2), L6 STDL $ 4, (IZ- # 1), L6; Indirect designation by SR
PPSM (PoP by System stack pointer Multi byte)	PPSM ***$ C5*** , ***IM3***	$ C5 (IM3) ← (SS) (IM3) SS ← SS + IM3	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2)	SS is the start address, the contents of the IM3 byte external memory address block are stored in the main register block of operand 1, and IM3 is added to SS. For example, PPSM $ 2,6 performs the following operations. • (SS) → $ 2 • (SS + 1) → $ 3	PPSM $ 2,6 KC format PPSW $ 2 (6) EU format PPSL $ 2, L6

					• (SS + 2) → $ 4 • (SS + 3) → $ 5 • (SS + 4) → $ 6 • (SS + 5) → $ 7 • SS ← SS + 6	
PPUM (PoP by User stack pointer Multi byte)	PPUM ***$ C5*** , ***IM3***	$ C5 (IM3) ← (US) (IM3) US ← US + IM3	No change	3 + 3 + 6 + 3 + 5 + 3 * (IM3-2) = 20 + 3 * (IM3-2)	Stores the contents of the IM3 byte external memory address block in the main register block of operand 1 and adds IM3 to US. For example, PPUM $ 2,6 has the following behavior. • (US) → $ 2 • (US + 1) → $ 3 • (US + 2) → $ 4 • (US + 3) → $ 5 • (US + 4) → $ 6 • (US + 5) → $ 7 • US ← US + 6	PPUM $ 2,6 KC format PPUW $ 2 (6) EU format PPUL $ 2, L6
PHSM (PusH System stack pointer Multi byte)	PHSM ***$ C5*** , ***IM3***	$ C5 (-IM3) → (SS-1) (-IM3) SS ← SS-IM3	No change	3 + 3 + 6 + 3 + 3 + 3 * (IM3-2) = 18 + 3 * (IM3-2)	Saves the contents of the main register block of operand 1 to the external memory whose address is SS-1 to SS-IM3 (push). At this time, the transfer direction of the main register and SS is the descending order (decrement) direction. After saving the data, SS is subtracted by IM3. For example, PHSM $ 7,6 operates as follows. • $ 7 → (SS-1) • $ 6 → (SS-2) • $ 5 → (SS-3) • $ 4 → (SS-4) • $ 3 → (SS-5) • $ 2 → (SS-6) • SS ← SS-6	PHSM $ 7,6 KC format PHSW $ 7 (6) EU format PHSL $ 2, L6
PHUM (PusH User stack pointer Multi byte)	PHUM ***$ C5*** , ***IM3***	$ C5 (-IM3) → (US-1) (-IM3) US ← US-IM3	No change	3 + 3 + 6 + 3 + 3 + 3 * (IM3-2) = 18 + 3 * (IM3-2)	The contents of the main register block of operand 1 are saved to the external memory whose addresses are US-1 to US-IM3 (push). At this time, the transfer direction of the	PHUM $ 7,6 KC format PHUW $ 7 (6) EU format PHUL $ 2, L6

					main register and US is the decrement direction. After saving the data, US subtracts IM3. For example, PHUM $ 7,6 performs the following operations. • $ 7 → (US-1) • $ 6 → (US-2) • $ 5 → (US-3) • $ 4 → (US-4) • $ 3 → (US-5) • $ 2 → (US-6) • US ← US-6	
STLM (STore Lcd data port Multi byte: undisclosed instruction)	STLM ***$ C5*** , ***IM3***	$ C5 (IM3) → LCD data port	No change	3 + 3 + 22 + 8 * (IM3-2) = 28 + 3 * (IM3-2)	Operand $ C5 to $ C5 + (IM3-1) are output to the LCD data area. Output is performed in order of 8 bits.	STLM $ 2,6 KC format STLW $ 2 (6) EU format OCBL $ 2,6
LDLM (LoaD Lcd data port Multi byte: undisclosed instruction)	LDLM ***$ C5*** , ***IM3***	$ C5 (IM3) ← LCD data port	No change	3 + 3 + 22 + 8 * (IM3-2) = 28 + 3 * (IM3-2)	Assign the value of the LCD data port to $ C5 to $ C5 + (IM3-1) of operand 1 according to the transfer protocol set in advance in LCDC. Reading is performed in 4-bit units, so when the graphic data on the screen is read, the upper and lower bits are switched in 4-bit units. (Depending on the data transfer protocol settings, the read value can be output directly to the LCD.) The reading procedure is as follows. (1) Specify drawing mode (anything) and LCD coordinate position to LCDC. (STLM after PPO & HDF) (2) Set read command (& HE1) to LCDC. (STL & HE1 after PPO & hDF)	LDLM $ 2,6 KC format LDLW $ 2 (6) EU format ICBL $ 2,6

					(3) Execute LDLM with data RAM specified. (LDLM after PPO & HDE)	
PPOM (Put lcd control POrt Multi byte: undisclosed instruction)	PPOM ***$ C5*** , ***IM3***	$ C5 (IM3) → LCD control Port	No change	3 + 3 + 11 + 8 * (IM3-2) = 17 + 8 * (IM3-2)?	Outputs the contents of operand 1 main registers $ C5 to $ C5 + (IM3-1) to the LCD control port. Output is performed in order of 8 bits.	PPOM $ 2,6 KC format PPOW $ 2 (6) EU format PCBL $ 2,6
PSRM (Put Specific index Register Multi byte)	PSRM ***SIR*** , ***$ C5*** , ***IM3***	SIR ← $ C5 (IM3)	No change	3 + 3 + 11 = 17	The contents of operand $ 2 registers $ C5 to $ C5 + (IM3-1) are stored in the SIR specific index register. Since the data is overwritten and as a result the contents of $ (C5 + (IM3-1)) are written to the SIR, this instruction is essentially unnecessary. SIR = SX, SY, SZ For example, when PSRM SX, $ 2,3 is executed with $ 2 = 0, $ 3 = 1, $ 4 = 2, $ (2 + 3-1) = $ 4 = 2 Assigned. **Refer to** PSR and PSRW **for precautions when using this instruction .**	PSRM SX, $ 2, IM3 KC format PSRW SX, $ 2 (IM3) EU format PRAL # 0, $ 2, IM3

Mnemonic Table - Multi-byte arithmetic operation instruction (2 to 8 bytes) not disclosed

	This instruction group expands the target register pair to 2 to 8 bytes by specifying operand 3 (INVM and CMPM are operand 2). Arithmetic operations up to 8 bytes (64 bits) can be performed with this single instruction. Strictly speaking, INVM and CMPM are classified into shift instructions, but they are explained here because they are easier to understand with arithmetic instructions. The number that can be specified for operand 3 (operand 2 for INVM and CMPM) is 1 to 8, but if a value smaller than 2 (= 1) is set, execution will be 2. HD61 is designed to output an error when 1 is specified. Second, even when the specific index register SIR is used as an operand, neither the instruction code nor the operation clock is reduced. The flag behavior seems to be as follows, but it is unknown whether this is accurate. Z : 0 if all bits are 0 as a result of operation. C : 1 when there is a carry or borrow from the most significant bit (MSB). LZ : 0 when the lowest 4 bits of the lowest 8 bits are 0.

	UZ : 0 when the upper 4 bits of the most significant 8 bits are 0.					
Mnemonic	**Format**	**Function**	**Flag**	**Number of Clocks**	**Description**	**Example Format**
INVM (INVert Multi byte)	INVM ***$ C5*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← NOT ($ C5 (IM3))	Z, C = 1, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	The contents of the main register block (IM3 byte) specified by operand 1 are bit-inverted (1's complement). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	INVM $ 2,6 INVM $ 2,6, LABEL; Jump expansion KC format INVW $ 2 (6) INVW $ 2 (6), JR LABEL; Jump expansion EU format INVL $ 2, L6 INVL $ 2, L6, LABEL; Jump expansion
CMPM (CoMPlement Multi byte)	CMPM ***$ C5*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← NOT ($ C5 (IM3)) + 1	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	The contents of the main register block (IM3 byte) specified by operand 1 are bit-inverted + 1 (2's complement). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	CMPM $ 2,6 CMPM $ 2,6, LABEL; Jump expansion KC format CMPW $ 2 (6) CMPW $ 2 (6), JR LABEL; Jump expansion EU format CMPL $ 2, L6 CMPL $ 2, L6, LABEL; Jump expansion
ADBM (ADd Bcd Multi byte)	ADBM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) + A (IM3) (BCD calculation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD adds the IM3 byte length $ C5 register block of operand 1 and the IM3 byte length A register block specified by operand 2 and stores the result in the A block. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	ADBM $ 8, $ 0,6; Main registers ADBM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format ADBW $ 8, $ 0 (6); Main registers ADBW $ 8, $ SZ (6), LABEL; Indirect specification with main

						register + SIR (with Jump extension) EU format ADBL $ 8, $ 0, L6; Main registers ADBL $ 8, # 2, L6, J.LABEL; main register + indirect specification with SR (with Jump extension)
ADBM (ADd Bcd immediate Multi byte)	ADBM ***$ C5*** , ***IM5*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) + IM5 (BCD calculation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD adds the contents of the main register block (IM3 byte) starting from $ C5 of operand 1 and the 5-bit value of operand 2 and stores the result in the $ C5 block. Operand 2 can specify a BCD value from 0 to 31. The calculation method of the BCD immediate value IM5 specified by operand 2 is Bit4 * & H10 + HextoBCD (Bit3 to Bit0). For example, if IM5 = & H1A, the number to be added is 1 * & H10 + & H10 (← 10 is expressed as a hexadecimal BCD) = & H20, and & H20 is BCD added to the main register $ C5 (IM3) . If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	ADBM $ 4, & H1F, 6; Main register + IM5 (In the example, add 1 * & H10 + HextoBCD (& HF) = & H25) ADBM $ 4,15,6, LABEL; Jump expansion KC format ADBW $ 4, & H1F (6); Main register + IM5 ADBW $ 4,15 (6), JR LABEL; Jump expansion EU format ADBL $ 4, & H1F, L6; Main register + IM5 ADBL $ 4,15, L6, J.LABEL; Jump expansion
ADBCM (ADd Bcd Check Multi byte)	ADBCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) + A (IM3) (BCD operation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD adds the IM3 byte length $ C5 register block of operand 1 and the IM3 byte length A register block specified by operand 2, but does not store the result anywhere. A can be specified only for $ C5 (including	ADBCM $ 8, $ 0,6; Main registers ADBCM $ 8, $ SZ, 6, LABEL; Indirect designation by main register + SIR (with Jump extension)

					indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	KC format ADBCW $ 8, $ 0 (6); Main registers ADBCW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format ADBCL $ 8, $ 0, L6; Main registers ADBCL $ 8, # 2, L6, J.LABEL; main register + indirect specification with SR (with Jump extension)
SBBM (SuB Bcd Multi byte)	SBBM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) -A (IM3) (BCD operation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD subtracts the IM3 byte-length A register block specified by operand 2 from the IM3 byte-length $ C5 register block of operand 1 and stores the result in the A block. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	SBBM $ 8, $ 0,6; Main registers SBBM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format SBBW $ 8, $ 0 (6); Main registers SBBW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format SBBL $ 8, $ 0, L6; Main registers SBBL $ 8, # 2, L6, J.LABEL; Main register + SR indirect

						specification (with Jump extension)
SBBM (SuB Bcd immediate Multi byte)	SBBM ***$ C5*** , ***IM5*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) -IM5 (BCD calculation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD adds the 5-bit value of operand 2 from the contents of the main register block (IM3 byte) starting from $ C5 of operand 1 and stores the result in the $ C5 block. Operand 2 can specify a BCD value from 0 to 31. The calculation method of the BCD immediate value IM5 specified by operand 2 is Bit4 * & H10 + HextoBCD (Bit3 to Bit0). For example, if IM5 = & H1A, the number to be added is 1 * & H10 + & H10 (← 10 is expressed as a hexadecimal BCD) = & H20, and & H20 is BCD added to the main register $ C5 (IM3) . If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	SBBM $ 4, & H1F, 6; Main register + IM5 (In the example, 1 * & H10 + HextoBCD (& HF) = & H25 is subtracted) SBBM $ 4,15,6, LABEL; Jump expansion KC format SBBW $ 4, & H1F (6); Main register + IM5 SBBW $ 4,15 (6), JR LABEL; Jump expansion EU format SBBL $ 4, & H1F, L6; Main register + IM5 SBBL $ 4,15, L6, J.LABEL; Jump expansion
SBBCM (SuB Bcd Check Multi byte)	SBBCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) - A (IM3) (BCD operation)	Z, C, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	BCD subtracts the IM3 byte length A register block specified by operand 2 from the IM3 byte length $ C5 register block of operand 1, but does not store the result anywhere. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	SBBCM $ 8, $ 0,6; Main registers SBBCM $ 8, $ SZ, 6, LABEL; Indirect designation with main register + SIR (with Jump extension) KC format SBBCW $ 8, $ 0 (6); Main registers SBBCW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR

						(with Jump extension) EU format SBBCL $ 8, $ 0, L6; Main registers SBBCL $ 8, # 2, L6, J.LABEL; main register + indirect specification with SR (with Jump extension)
ANM (ANd Multi byte)	ANM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) and A (IM3)	Z, C = 0, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	The AND of the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes) is taken, and the result is stored in the $ C5 block. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	ANM $ 8, $ 0,6; Main registers ANM $ 8, $ SZ, 6, LABEL; Main register + Indirect specification with SIR (with Jump extension) KC format ANW $ 8, $ 0 (6); Main registers ANW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format ANL $ 8, $ 0, L6; Main registers ANL $ 8, # 2, L6, J.LABEL; Main register + SR indirect specification (with Jump extension)
ANCM (ANd Check Multi byte)	ANCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) and A (IM3)	Z, C = 0, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	ANDs the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), but does not store the result anywhere. A can be specified only	ANCM $ 8, $ 0,6; Main registers ANCM $ 8, $ SZ, 6, LABEL; Indirect designation by main register + SIR (with Jump extension)

					for $ C5 (including indirect specification by SIR). However, the flag changes. If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	KC format ANCW $ 8, $ 0 (6); Main registers ANCW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format ANCL $ 8, $ 0, L6; Main registers ANCL $ 8, # 2, L6, J.LABEL; Indirect specification with main register + SR (with Jump extension)
NAM (NAnd Multi byte)	NAM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) nand A (IM3)	Z, C = 1, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	NAND the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), and store the result in the $ C5 block. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	NAM $ 8, $ 0,6; Main registers NAM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format NAW $ 8, $ 0 (6); Main registers NAW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format NAL $ 8, $ 0, L6; Main registers NAL $ 8, # 2, L6, J.LABEL; Main register + SR indirect specification (with Jump extension)

NACM (NAnd Check Multi byte)	NACM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) nand A (IM3)	Z, C = 1, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	NAND the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), but do not store the result anywhere. However, the flag changes. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	NACM $ 8, $ 0,6; Main registers NACM $ 8, $ SZ, 6, LABEL; Indirect designation by main register + SIR (with Jump extension) KC format NACW $ 8, $ 0 (6); Main registers NACW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format NACL $ 8, $ 0, L6; Main registers NACL $ 8, # 2, L6, J.LABEL; Indirect specification with main register + SR (with Jump extension)
ORM (OR Multi byte)	ORM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) or A (IM3)	Z, C = 1, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	Performs a logical OR operation on the contents of operand 1's $ C5 block and operand 2's A block (both A and B are IM3 bytes) and stores the result in the $ C5 block. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	ORM $ 8, $ 0,6; Main registers ORM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format ORW $ 8, $ 0 (6); Main registers ORW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format

						ORL $ 8, $ 0, L6; Main registers ORL $ 8, # 2, L6, J.LABEL; Main register + Indirect specification with SR (Jump extension available)
ORCM (OR Check Multi byte)	ORCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) or A (IM3)	Z, C = 1, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	Performs a logical OR (OR) operation on the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), but the result is not stored anywhere. However, the flag changes. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	ORCM $ 8, $ 0,6; Main registers ORCM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format ORCW $ 8, $ 0 (6); Main registers ORCW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format ORCL $ 8, $ 0, L6; Main registers ORCL $ 8, # 2, L6, J.LABEL; Main register + SR indirect specification (with Jump extension)
XRM (eXclusive oR Multi byte)	XRM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) ← $ C5 (IM3) xor A (IM3)	Z, C = 0, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	Performs an XOR operation on the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), and stores the result in the $ C5 block. A can be specified only	XRM $ 8, $ 0,6; Main registers XRM $ 8, $ SZ, 6, LABEL; Indirect specification with main register + SIR (with Jump extension) KC format

					for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	XRW $ 8, $ 0 (6); Main registers XRW $ 8, $ SZ (6), LABEL; Indirect specification with main register + SIR (with Jump extension) EU format XRL $ 8, $ 0, L6; Main registers XRL $ 8, # 2, L6, J.LABEL; Indirect specification with main register + SR (with Jump extension)
XRCM (eXclusive oR Check Multi byte)	XRCM ***$ C5*** , ***A*** , ***IM3*** [, (JR) ***LABEL***]	$ C5 (IM3) xor A (IM3)	Z, C = 0, LZ, UZ change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2) (JR: +3)	Performs an XOR operation on the contents of the $ C5 block of operand 1 and the contents of the A block of operand 2 (both A and B are IM3 bytes), but the result is not stored anywhere. However, the flag changes. A can be specified only for $ C5 (including indirect specification by SIR). If the last operand has a label, a relative jump is made after the operation. (JR tag can be omitted)	XRCM $ 8, $ 0,6; Main registers XRCM $ 8, $ SZ, 6, LABEL; Main register + SIR indirect specification (with Jump extension) KC format XRCW $ 8, $ 0 (6); Main registers XRCW $ 8, $ SZ (6), LABEL; Main register + SIR indirect specification (with Jump extension) EU format XRCL $ 8, $ 0, L6; Main registers XRCL $ 8, # 2, L6, J.LABEL; Main register + SR indirect specification (with Jump extension)

Mnemonic Table - Multibyte shift instruction (2 to 8 bytes) not disclosed

	This instruction group expands the target register pair to 2 to 8 bytes by specifying operand 2. This single instruction can shift up to 8 bytes (64 bits), and only DIUM, DIDM, BYUM, and BYDM are available. The bit shift system (BIUM, BIDM) and the rotate system (ROUM, RODM) are not prepared, and BUP and BDN are assigned to the instruction code to which they should have been assigned.

Mnemonic	Format	Function	Flag	Number of Clocks	Description	Example Format
DIUM (Digit Up Multi byte)	DIUM ***$ C5 , IM3***	See figure	Z, C = 0, LZ = 0, UZ changes	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2)	The contents of the register block $ C5 to $ (C5 + (IM3-1)) starting with the main register number specified by operand 1 are increased 4 bits to the left, and 0 is placed in the lowest 4 bits.	DIUM $ 2,6; The register block is $ 2 to $ 7 (6byte ascending order). KC format DIUW $ 2 (6) EU format DIUL $ 2, L6
DIDM (Digit Down Multi byte)	DIDM ***$ C5 , IM3***	See figure	Z, C = 0, LZ, UZ = 0 change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2)	The contents of the register blocks $ C5 to $ (C5- (IM3-1)) starting with the main register number specified by operand 1 are lowered 4 bits to the right, and 0 is entered in the most significant 4 bits.	DIDM $ 7,6; Register block is $ 7 to $ 2 (6byte descending order). KC format DIDW $ 7 (6) EU format DIDL $ 7, L6
BYUM (BYte Multi byte)	BYUM ***$ C5 , IM3***	See figure	Z, C = 0, LZ = 0, UZ changes	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2)	The contents of the register block $ C5 to $ (C5 + (IM3-1)) starting with the main register number specified by operand 1 are increased 8 bits to the left, and all 0s are entered in the least significant byte.	BYUM $ 2,6; The register block is $ 2 to $ 7 (in ascending order of 6 bytes). KC format BYUW $ 2 (6) EU format BYUL $ 2, L6
BYDM (BYte Multi byte)	BYDM ***$ C5 , IM3***	See figure	Z, C = 0, LZ, UZ = 0 change	3 + 3 + 11 + 5 * (IM-2) = 17 + 5 * (IM3-2)	The contents of the register block $ C5 to $ (C5- (IM3-1)) starting with the main register number specified by operand 1 are down 8 bits to the right, and 0 is placed in the most significant byte.	BYDM $ 7,6; The register block is $ 7 to $ 2 (6byte descending order). KC format BYDW $ 7 (6) EU format BYDL $ 7, L6

7-5 Instruction set Table

- HD61700.PDF: HD61700 Instruction Set Table
- HD61700.PDF: HD61700 Instruction Set Table (Open in a new window)

7-6 Appendix

Output Format and Loader

This section describes the BAS format, PBF format, QL format and their loaders that the HD61 cross assembler outputs as default (no option), / p, and / q, respectively.
For the sake of easy understanding, the list in Table 6-1 will be given in a file output in each format.

Table 6-1. Sample list
HD61700 ASSEMBLER Rev 0.41-ASSEMBLY LIST OF [quick-loader.s]

```
0001: 0000          ;
0002: 0000          ; quick-loader.s
0003: 0000          ; relocatble quick loader for FX-870P / VX-4
0004: 0000          ;
0005: 0000
0006: 0000          CGRAM: EQU & H153C          ; address of DEFCHR $ ()
0007: 0000          LEDTP: EQU & H123C          ; address of LCD dot matrix
0008: 0000
0009: 153C          ORG CGRAM
0010: 153C          START CGRAM
0011: 153C
0012: 153C          D6403C12 PRE IZ, LEDTP
0013: 1540          D6003C15 PRE IX, CGRAM
0014: 1544          D6205315 PRE IY, CGRAM + 23
0015: 1548          D8 BUP                      ; BlockUP
0016: 1549          566054 PST UA, & H54
0017: 154C          F7 RTN
0018: 154D          ; end of program
```

BAS Format

The BAS format files listed in Table 6-1 are shown in Table 6-2. As can be easily understood from Table 6-1 and Table 6-2, the BAS format is as follows.

- A text file to be included in a BASIC program.
- The first line of data consists of the machine language file name, machine language start address, machine language end address, and machine language execution address from the beginning.
- The second and subsequent lines are machine language data, and each line consists of a string consisting of 8 bytes expressed in hexadecimal notation and two pieces of data, the least significant byte of the 8-byte checksum. If the last line is less than 8 bytes, no extra data is added at the end.

BAS Format Files in Table

999 DATA QUICK-LOADER.EXE, & H153C, & H154C, & H153C

```
1000 DATA D6403C12D6003C15,8B
1001 DATA D6205315D8566054,40
1002 DATA F7, F7
```

Writing a machine language prepared as data in the DATA statement in this way with a POKE statement is common in pocket computers, especially when there is no way to read a machine language such as BLOAD from an external device. This is the basis of how words are placed in memory. When stored as a DATA statement, a 1-byte code requires 2 bytes even in hexadecimal format, which is not good in terms of the use efficiency of the pocket computer memory. However, usability is generally good due to a device such as a loader.
A loader called Trans.b is attached to the HD61 cross assembler, and the list is shown in Table 6-3.

Table 6-3. Trans.b (loader for BAS format; for PB-1000 / C, AI-1000)

```
'*** ASC2BIN for PB-1000 / C, AI-1000 ***
10 CLEAR: READ F $, ST, ED, EX: A = ST: ED = ED + 1: L = 1000
20 READ A $, S $: S = 0
30 FOR I = 1 TO LEN (A $) STEP2
40 D = VAL ("& H" + MID $ (A $, I, 2)): POKE A, D
50 S = S + D: A = A + 1: NEXT
60 IF RIGHT $ (HEX $ (S), 2) <> S $ THEN BEEP: PRINT "SUM ERROR: LINE ="; L: END
70 IF A <ED THEN L = L + 1: GOTO 20
80 IF EX <> 0 THEN BSAVE F $, ST, ED-ST, EX ELSE BSAVE F $, ST, ED-ST
90 BEEP1: PRINT "FILE CREATED": END
```

Trans.b is for PB-1000 / C and AI-1000 and saved as a binary file using BSAVE at line number 80, but FX-870P / VX-4 / VX-3 There is no BSAVE instruction. Therefore, **to use Trans.b on FX-870P / VX-4 / VX-3, it is necessary to comment or delete line number 80.**
Line number 60 checks whether the data in each DATA statement is correct using checksum data. This helped to make it easier to find typographical errors during program execution in an era when the Internet and personal computer communications were not common and you had to manually enter the program published in the magazine. Therefore, the checksum is basically useless in the present age when the network has developed and it is no longer necessary to input another person's machine code.

PBF format

PBF format files listed in Table 6-1 are shown in Table 6-4. Although there are parts that cannot be understood from Table 6-1 and Table 6-4, the PBF format is as follows.

- A text file to send to a pocket computer from a personal computer.
- The first line of data consists of the machine language file name, machine language start address, machine language end address, and machine language execution address from the beginning.
- The second and subsequent lines are machine language data, consisting of a string of characters expressed in hexadecimal notation at every 120 bytes from the start address and two pieces of checksum for each data. If the last line is less than 120 bytes, no extra data is added at the end.

Table 6-4. PBF format file in Table 6-1
QUICK-LOADER.EXE, 5436,5452,5436
D6403C12D6003C15D6205315D8566054F7,1730

The PBF format is a text file of a machine language program used for software distribution on the home page of CASIO PB-1000 FOREVER by Jun Amano. This file is transferred from the personal computer to the pocket computer via RS-232C, loaded as a machine language code into the memory, and then filed on the pocket computer side. At that time, a PBF file reception program is required on the pocket computer side.

A loader for FX-870P / VX-4 / VX-3 called TransVX.b is attached to the HD61 cross assembler. The list is shown in Table 6-5.

Table 6-5. TransVX.b (loader for PBF format; for FX-870P / VX-4 / VX-3)

```
'PbfToBinVX.BAS (c) JUN AMANO / BLUE
10 CLS: CLEAR: OPEN "COM0:" FORINPUT AS # 1
20 INPUT # 1, F $, ST, ED, EX: AD = ST: BEEP
30 PRINT "Converting:"; F $: PRINT "Start:"; HEX $ (ST); "H End:"; HEX $ (ED); "H"
40 INPUT # 1, A $, S: SUM = 0
50 FOR I = 1 TO LEN (A $) STEP2
60 A = VAL ("& H" + MID $ (A $, I, 2))
70 POKE AD, A: SUM = SUM + A: AD = AD + 1: NEXT
80 IF S <> SUM THEN PRINT "SUM ERROR": BEEP: CLOSE: END
90 IF ED> AD THEN GOTO 40
100 CLOSE: PRINT "Completed!": BEEP1
110 IF EX <> 0 THEN PRINT "Execute MODE110 ("; EX; ")";
```

Note that the setting value of F.COM is used for the communication parameter of TransVX.b. When changing the setting, modify the file descriptor "COM0:" part of line number 10 . In addition, TransVX.b (for FX-870P / VX-4 / VX-3) attached to HD61 is written in BASIC only from the viewpoint of portability, but the version accelerated in machine language is Jun Amano. Published on his website "CASIO PB-1000 Forever!" The URL is http://homepage3.nifty.com/lsigame/pb-1000/softlib/pbsoft1.htm
The machine language data is divided in units of 120 bytes when reading data into A $. This may be due to the BASIC limit of 255 characters.

QL Format

QL format files listed in Table 6-1 are shown in Table 6-6. As can be understood from Table 6-1 and Table 6-6, the QL format is as follows.

- A text file to be included in a BASIC program.
- The data of the first line is the machine language start address, machine language end address, machine language execution address from the beginning.
- The second and subsequent lines are machine language data, and four character strings expressed in hexadecimal notation every 6 bytes from the start address are stored in one line, and 24 bytes are stored in one line. If the last line is less than 24 bytes, add 0 to the shortage to make it 24 bytes.

Table 6-6. QL format files in Table 6-1

```
1000 DATA 5436,5452,5436
1001 DATA D6403C12D600,3C15D6205315, D8566054F700,000000000000
```

The QL format is a data format for use with a quick loader that is about 10 times faster than loading machine language into memory in BAS format.

The quick loader was devised by Mr. Ao, the creator of HD61.
In the HD61 cross assembler, there is no detailed explanation about the QL format, and no loader is attached, but the list of the quick loader used in the programs that can be downloaded with "CASIO PB-1000 FOREVER!" And "HD61700 SPIRITS" Is shown in Table 6-7.

Table 6-7. Quick loader example (QL type loader; FX-870P / VX-4)

```
5 'Expanded CLEAR 0.04 for VX-4 / FX-870P 2003 BLUE
100 GOSUB900: BEEP1: PRINT "MODE110 (& H"; HEX $ (EX); ")" ;: END
900 'Machine Code Loader (FX-870P / VX-4)
910 RESTORE1000: READ ST, ED, EX: C = INT ((ED-ST) / 24)
```

```
920 DEFCHR $ (252) = "D6403C12D600": DEFCHR $ (253) = "3C15D6205315"
930 DEFCHR $ (254) = "D8566054F700": MODE110 (& H153C)
940 FOR I = 0 TO C: READ A $, B $, C $, D $
950 DEFCHR $ (252) = A $: DEFCHR $ (253) = B $: DEFCHR $ (254) = C $: DEFCHR $ (255) = D $
960 POKE & H123E, (ST MOD 256): POKE & H123F, INT (ST / 256)
970 IF (ED-ST) <24 THEN POKE & H1246, & H3C + (ED-ST)
980 MODE110 (& H123C): ST = ST + 24: NEXT: CLS: RETURN
```

Table 6-5 shows the loader part of the extended CLEAR that secures the machine language area in the memory with FX-870P / VX-4 that can be downloaded with "CASIO PB-1000 FOREVER!" .
Quick loader

- Load machine language data to the CGRAM in the system area at high speed with the DEFCHR $ instruction .
- The machine language loader loads the machine language (up to 24 bytes) into CGRAM to the target address at high speed.

High speed is realized by this method.
In the case of BAS format, the 1-byte data fetched with "D = VAL (" & H "+ MID $ (A $, I, 2))" as shown in Table 6-3, line number 60 is "POKE AD, The process of writing to memory with D "is to extract the byte data character string from the character string, digitize it, convert the BCD floating point data of the numeric variables AD, D to the integer type with the POKE statement, and then write to the memory. The work to write to is done inside the BASIC system, and is more complicated than the program has seen, making it inefficient.
On the other hand, the quick loader shown in Table 6-5 uses a system area as a relay point for memory transfer, but is a ROM routine that is optimized for 6 bytes x 4 = 24 bytes in the DEFCHR $ statements of line numbers 920, 930, and 950. After the transfer, the transfer destination address is rewritten with the line number 960, and the machine language transfer routine performs the transfer to the target address, minimizing unnecessary character string manipulation and numerical conversion, and speeding up. It has been realized.
In fact, even in the BAS format, do not perform "D = VAL (" & H "+ MID $ (A $, I, 2))", store once in CGRAM in the system area with DEFCHR $, and then transfer with PEEK, POKE It has been confirmed that the speed can be increased by about 35% just by using the method.
The list in Table 6-1 is the source equivalent of the machine language transfer routine, and it can be confirmed that they match by comparing line numbers 920 and 930 in Table 6-6 and Table 6-7. . The behavior is

- After specifying the transfer source start address and end address (CGRAM 24 bytes) and transfer destination (system area LEDTP) with IX, IY, IZ ,
- Use block transfer instruction BUP to transfer 24 bytes of data and return

Perform the operation. This action transfers its own code to the LEDTP in the system area. As can be seen from Table 6-1, since the absolute jump instruction is not used, this machine language transfer routine is relocatable and can be executed at the transfer destination. Therefore, a routine for high-speed transfer from CGRAM to an arbitrary address is realized by rewriting the IZ transfer destination address with a POKE statement such as line number 960.
Although the quick loader in Table 6-7 is compact, it is difficult for humans to read for the first time, and it is not easy to modify the program. Table 6-8 shows quick loaders with improved readability, operability, and portability.

Table 6-8. Quick loader Example (QL loader; FX-870P / VX-4)

```
90 'quick-loader rewritten for readability, usability and portability
100 CLS: GOSUB 850: MODE110 (EX): END
110 '
840 'Quick Loader (FX-870P / VX-4)
845 'LDAD + 2,3: destination addr; LDAD + 6,7: source start addr; LDAD + 10,11: source end addr
```

```
850 CGRAM = & H153C: LDAD = & H1A3C: 'addr of DEFCHR $ () and Mac-loader (in SAVE / LOAD buffer)
855 DEFCHR $ (252) = "D6403C1AD600": DEFCHR $ (253) = "3C15D6205315": DEFCHR $ (254) =
"D8566054F700"
860 MODE110 (CGRAM): 'relocatable mac-loader is transfered to LDAD by itself
865 IOBF = & H1895: IOBF = PEEK (IOBF) + PEEK (IOBF + 1) * 256
870 RESTORE 1000: READ ST, ED, EX: C = INT ((ED-ST) / 24)
875 IF ED> = IOBF THEN BEEP: PRINT "Cannot alloc memory!": PRINT "Make mac area at least"; ED-ST +
1; "bytes": END
880 GOSUB 980
885 P = 0
890 FOR I = 0 TO 23
895 IF PEEK (ST + I) = PEEK (CGRAM + I) THEN P = P + 1
900 NEXT
905 IF P <> 24 THEN 915 ELSE BEEP 1: PRINT "Mac codes already loaded.": PRINT "Hit any key."
910 A $ = INKEY $: IF A $ = "" THEN 910 ELSE RETURN
915 CLS
920 ST0 = ST
925 FOR I = 0 TO C
930 POKE LDAD + 2, (ST MOD 256): POKE LDAD + 3, INT (ST / 256): 'change destination
935 IF (ED-ST) <23 THEN POKE LDAD + 10, & H3C + (ED-ST): ST = ED-23: 'change transfer size
940 MODE110 (LDAD): ST = ST + 24: 'execute data transfer by 24 bytes, basically
945 LOCATE 0,2: PRINT "BLOAD:"; ST-ST0; "bytes";
950 IF I <C THEN GOSUB 980: 'data preparation for mac-loader
955 NEXT
960 PRINT "-completed."
965 RETURN
970 '
975 '* DATPRE:' data preparation
980 READ A $, B $, C $, D $
985 DEFCHR $ (252) = A $: DEFCHR $ (253) = B $: DEFCHR $ (254) = C $: DEFCHR $ (255) = D $
990 RETURN
```

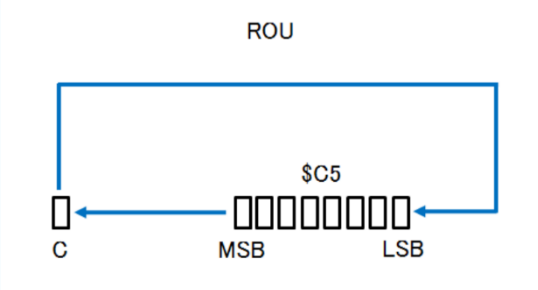
995 ' Line number 850 defines the start address CGRAM of DEFCHR $ and the transfer destination (execution) address LDAD of the machine language transfer routine. Line number 845 indicates the location of the transfer destination address, transfer source start address, and transfer source end address of the machine language transfer routine. If the transfer destination address is changed with the POKE statement, such as line numbers 930 and 935, Good.

7-7 References and Links

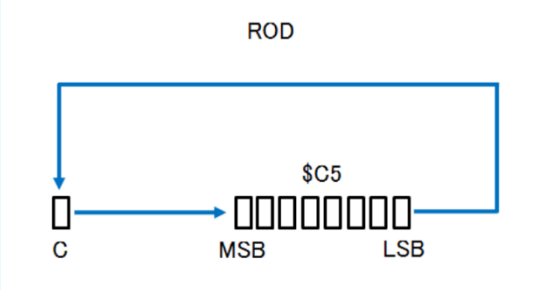
- (1) Ao: “HD61700 Assembly Language Manual”, http://www.geocities.jp/hd61700lab/
- (2) Piotr Piatek: "Description of the HD61700 microprocessor assembly language", http://www.pisi.com.pl/piotr433/index.htm
- (3) Kota-chan: PJ August 1990 issue, p.35, "KC-Disassembler".
- (4) P, H, M ,: PJ December 1992, p.51, `` Assassembler ''.
- (5) Aya Toji: PJ April 1993, p.83, `` FX-870P Assembler ''.
- (6) Hakkun: PJ September 1993, p.83, `` HD61700 X-Assembler Ver.4.05 ''.
- (7) N. Hayashi: PJ February 1995, p.42, `` HD61700 X-Assembler Ver.6 ''.

7-8 Figure

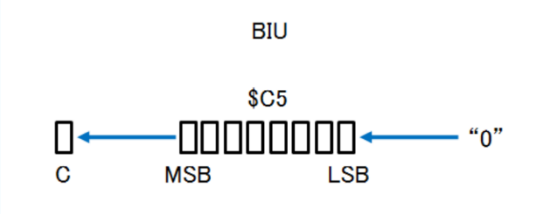
ROU-Betriebsdiagramm



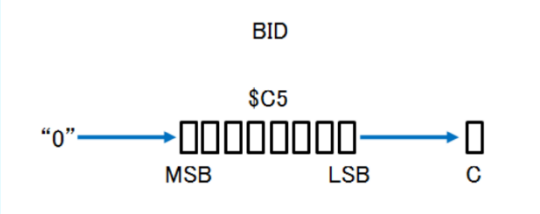
ROD-Betriebsdiagramm



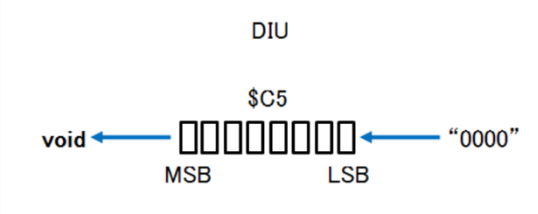
BIU-Betriebsdiagramm



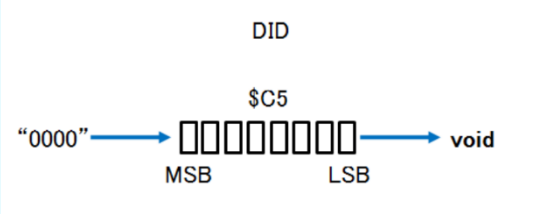
BID-Funktionsdiagramm



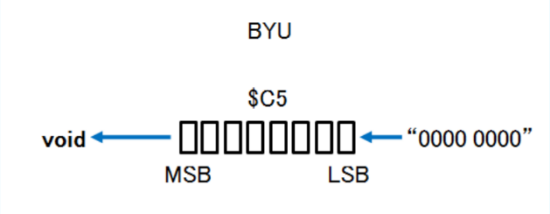
DIU-Betriebsdiagramm



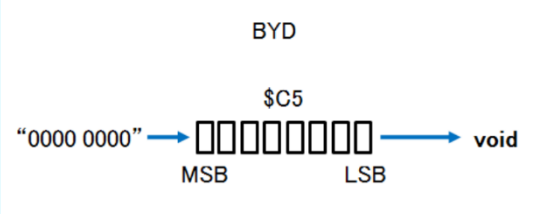
DID Betriebsdiagramm



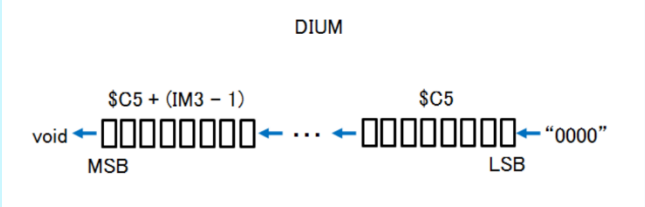
BYU-Betriebsdiagramm



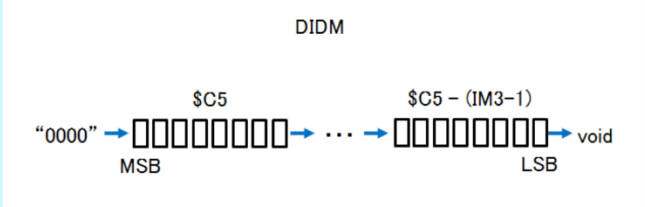
BYD-Betriebsdiagramm



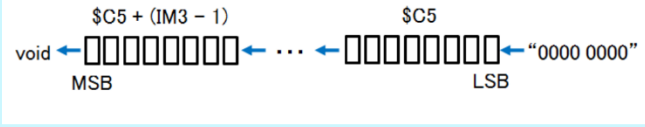
DIUM Funktionsdiagramm



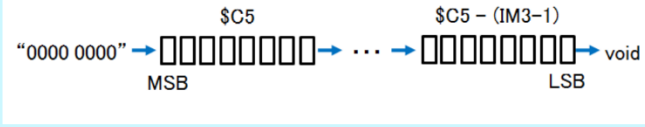
DIDM-Betriebsdiagramm



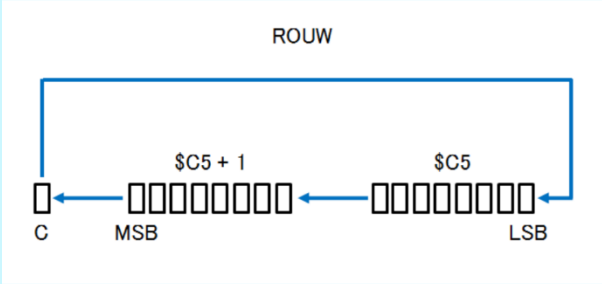
BYUM-Betriebsdiagramm



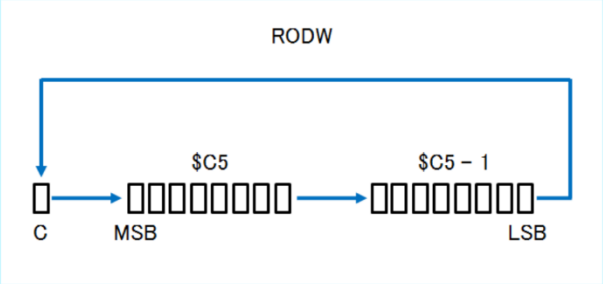
BYDM-Betriebsdiagramm



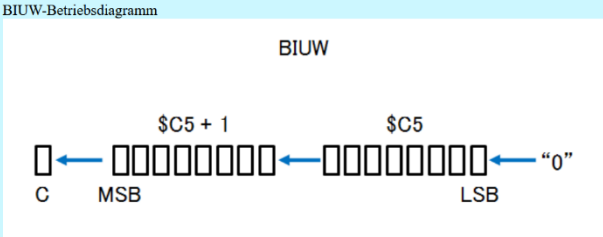
ROUW-Funktionsdiagramm



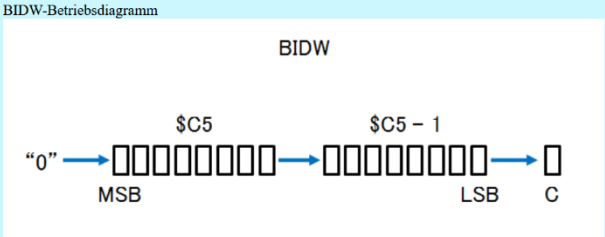
RODW-Betriebsdiagramm



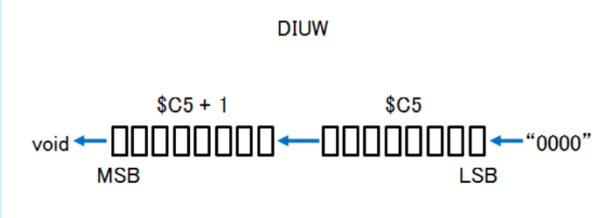
BIUW-Betriebsdiagramm



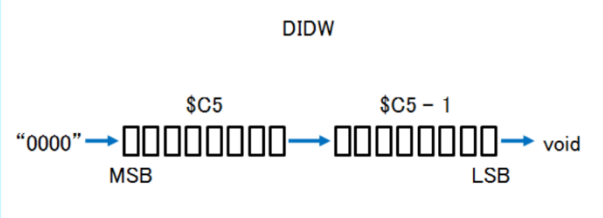
BIDW-Betriebsdiagramm



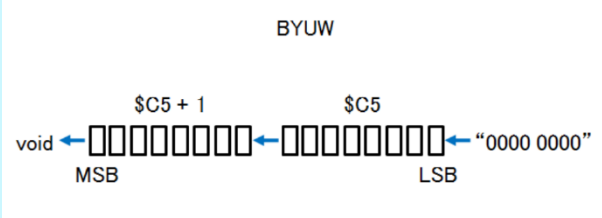
DIUW-Betriebsdiagramm



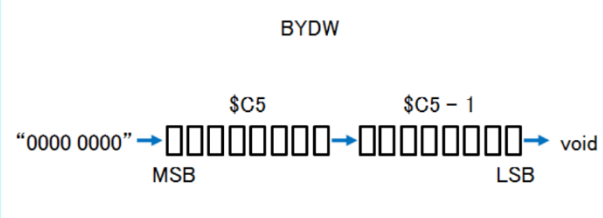
DIDW Betriebsdiagramm



BYUW-Betriebsdiagramm



BYDW-Betriebsdiagramm



7-9 Revision Information

Ed.1	2011/6/12 Completed the HTML of the manual attached to HD61. The original description mistakes have been corrected, but there is a possibility that you have made a mistake. In the future, correction of description errors and addition of information are planned.

VIII. CASL

In advance:
Information about the programming language CASL, as a book or on the Internet, is only available in Japanese. Furthermore, there is no German or English manual for the Casio FX-870P and the VX-4.

Despite extensive research, no comprehensive reference was found. The few PDFs on the Internet (Springer, CoFI, CANape, Crosstalk) do not describe the VX-4 - CASL language.

The few CASL websites found and the "readable" pages of the original manual are listed here. The compiled writings on CASL are just an attempt to give some insight into the language itself. For a deeper insight into the CASL language, you probably have to learn Japanese and the ones described in Section IX. Work through the books shown in the manuals.

Information shown in this chapter is translated from:
- Japanese WIKIPEDIA article
- Pages from the original manual
- TeamCASL website found:
  http://www5a.biglobe.ne.jp/~teamcasl/caslkozatop.htm

8-1 What is CASL / COMET?

CASL is simple implementation of CASL assembler and COMET simulator written in Perl. The CASL assembler and the COMET simulator are designed for users to study principle operations of computers. In particular, CASL and COMET are used in a qualifying examination called as Japan Information-Technology Engineers Examination so that these programs would be of value for people who would like to acquire this qualification. Since both the CASL assembler and the COMET simulator are written only in Perl version 5, these should work on almost all operating system including UNIX flavors, MS-DOS, Windows, and Macintosh.

CASL, the Common Algebraic Specification Language, was designed by the members of CoFI, the Common Framework Initiative for algebraic specification and development, and is a general-purpose language for practical use in software development for specifying both requirements and design. CASL is already regarded as a de facto standard, and various sublanguages andextensions are available for specific tasks.

COMET is the name of a virtual computer designed to be used for assembler language questions in information processing engineer tests .

Since the assembler language depends on hardware , COMET was developed as a non- existent computer , so-called virtual computer , to be fair to candidates for information processing engineer tests .

COMET is 16 bits per word and has five general-purpose registers , a program counter, and a flag register . Its main memory capacity is 65536 words, and it has a two-word instruction word that is sequentially controlled . The assembler language for COMET is called CASL, and in the assembler language section of the information processing engineer test , the program is written in CASL .

Although COMET is a virtual computer , several simulators have been created that run on Windows OS , etc., and are useful for understanding the operating principles of computers .

As of 2007, COMET II , the successor to COMET , is being used in the trial . In the past tests , a virtual machine called COMP-X was used , and the specifications are constantly being reviewed in this way for educational considerations . Among such virtual machines , MIX, which was devised by the author of the famous book " The Art of Computer Programming " on algorithms , is known.

WIKIPEDIA:

The Common Algebraic Specification Language (CASL) is a general-purpose specification language based on first-order logic with induction. Partial functions and subsorting are also supported.

CASL has been designed by CoFI, the Common Framework Initiative (CoFI), with the aim to subsume many existing specification languages.

CASL comprises four levels:

basic specifications, for the specification of single software modules,

structured specifications, for the modular specification of modules,

architectural specifications, for the prescription of the structure of implementations,

specification libraries, for storing specifications distributed over the Internet.

The four levels are orthogonal to each other. In particular, it is possible to use CASL structured and architectural specifications and libraries with logics other than CASL. For this purpose, the logic has to be formalized as an institution. This feature is also used by the CASL extensions.

8-2 Japanese CASL Wikipedia Article

This document describes the COMET/CASL implementation on the Casio PB-1000C which may differ from the original specification. It is based on the Japanese Wikipedia article <http://ja.wikipedia.org/wiki/CASL> and on the analysis of the PB-1000C ROM disassembly.

Overview

COMET is a virtual computer specially designed for educational purposes and programming ability testing in the Japanese Information Technology Standards Examination (JITSE). CASL is an assembly language for this computer. The revised versions of COMET and CASL, called COMET II and CASL II, are not supported by the PB-1000C and therefore are out of the scope of this document.

COMET Specification

COMET is a virtual machine with a von Neumann architecture. It operates on words of a fixed length of 16 bits. The processing is sequential. Negative numbers are represented in two's complement format.

The following Data Types are Supported:

1. arithmetic, refers to signed integers in range -32768 to 32767
2. logical, refers to unsigned integers in range 0 to 65535
3. character, using an 8-bit Japanese standard JIS X 0201 that defines encoding for Latin and Katakana characters, stored one character per word in the lower 8 bits while the upper 8 bits are filled with zeros

The Registers are as Follows:

1. General purpose 16-bit registers GR0, GR1, GR2, GR3, GR4

These registers contain one of the operands and store results of the arithmetic, logical and shift operations. The other operand is a memory location referenced by the effective address, specified either directly by an absolute address, or by a sum of an absolute address and the contents of an index register (XR). GR1 to GR4 can be used as index registers.
GR4 is used as a stack pointer. It holds the address of the top of the stack. When a value is pushed onto the stack, GR4 is decremented by one, then the value is placed at the memory location pointed to by it. When a

value is popped off the stack, the contents of the memory location pointed to by GR4 is transferred, then GR4 is incremented by one.
An address range from #FF80 to #FFFF is allocated for the stack, but actually the stack and the object code occupy different address spaces. Therefore it is not possible to access the object code memory with the commands PUSH or POP, nor the stack area through an effective address.

2. Program counter PC

This register holds the memory address of the instruction currently being executed. After completing the instruction it is incremented so as to point to the next one, except on branches, subroutine calls and subroutine returns which load it with a new value.

3. Flag register FR

When the executed instruction is an arithmetic or logical operation, it is set to 10 (binary) if the result is negative, 00 if positive, and 01 if zero. Similarly, for comparison instructions it is set according to the comparison result.

Instruction Format:

All instructions have a fixed length of two 16-bit words. These 32 bits are divided into the following fields:

1. The OP field (8 bits) is the instruction opcode that specifies the operation to be performed.

2. The GR field (4 bits) specifies the number of the register to be used in the operation. It is ignored for the branch and PUSH instructions.

3. The XR field (4 bits) specifies the number of the register whose contents is added to the adr field to form an effective address. A value of 0 does not mean GR0, but that no address modification is performed.

4. The adr field (16 bits) specifies the memory address, optionally modified by the XR. Both the adr and XR fields are ignored for the POP and RET instructions.

```
bit #    0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
      ------------------------------------------------
word 1  |      OP field       | GR field  | XR field  |
      ------------------------------------------------
word 2  |                adr field                    |
      ------------------------------------------------
```

Instruction set summary:

The items within brackets [] are optional and can be omitted.

LD GR, adr [, XR] - LoaD
Load the contents of the effective address to the specified GR register.

ST GR, adr [, XR] - STore
Store the contents of the GR register at the effective address.

LEA GR, adr [, XR] - Load Effective Address
Calculate the effective address and store it in the GR register.

ADD GR, adr [, XR] - ADD arithmetic
Adds the contents of the effective address to the contents of the GR and stores the result in the GR. The FR is set according to the result of the operation.

SUB GR, adr [, XR] - SUBtract arithmetic

Subtracts the contents of the effective address from the contents of the GR and stores the result in the GR. The FR is set according to the result of the operation.

AND GR, adr [, XR]
Performs a bitwise AND operation between the contents of the GR and the contents of the effective address. The result is stored in the GR. In other words, the operation clears the bits of the contents of GR which corresponding bits of the contents of the effective address are cleared. The FR is set according to the result of the operation.

OR GR, adr [, XR]
Performs a bitwise inclusive OR operation between the contents of the GR and the contents of the effective address. The result is stored in the GR. In other words, the operation sets the bits of the contents of GR which corresponding bits of the contents of the effective address are set. The FR is set according to the result of the operation.

EOR GR, adr [, XR] - Exclusive OR
Performs a bitwise exclusive OR operation between the contents of the GR and the contents of the effective address. The result is stored in the GR. In other words, the operation toggles the bits of the contents of the GR which corresponding bits of the contents of the effective address are set. The FR is set according to the result of the operation.

CPA GR, adr [, XR] - ComPare Arithmetic
Compare the contents of the GR with the contents of the effective address. The FR is set to 00 if the contents of GR is larger, 01 if equal, and 10 if smaller. The operands are treated as signed values.

CPL GR, adr [, XR] - ComPare Logical
Similar to the CPA except that the operands are treated as unsigned values.

SLL GR, adr [, XR] –
Shift Left LogicalThe contents of the GR is shifted to the left by the effective address. The shifted out bits are discarded and the vacated bits are filled with zeros. The FR is set according to the result of the operation.

SLA GR, adr [, XR] - Shift Left Arithmetic
The contents of the GR, except for the sign bit, is shifted to the left by the effective address. The shifted out bits are discarded and the vacated bits are filled with zeros. The FR is set according to the result of the operation.

SRL GR, adr [, XR] - Shift Right Logical
Right shift version of SLL.

SRA GR, adr [, XR] - Shift Right Arithmetic
Right shift version of SLA. The vacated bits are filled with the sign bit instead of zeros.

JPZ adr [, XR] - Jump on Plus or Zero
Branch to effective address (i.e. change the value of PC to the contents of the effective address) when the value of FR is 00 or 01.

JMI adr [, XR] - Jump on MInus
Branch to effective address when the value of FR is 10.

JNZ adr [, XR] - Jump on Non Zero
Branch to effective address when the value of FR is 10 or 01.

JZE adr [, XR] - Jump on ZEro
Branch to effective address when the value of FR is 00.

JMP adr [, XR] - unconditional JuMP
Branch to effective address unconditionally.

PUSH adr [, XR] - PUSH effective address
Calculate the effective address and store it on the top of the stack.

POP GR - POP a value

Retrieve the address stored at the top of the stack to a GR.

CALL adr [, XR] - CALL subroutine
Push the address of the subsequent instruction (=PC+2) onto the stack then pass the control to specified effective address.

RET - RETurn form subroutine
Branch to address popped from the stack.

CASL Specification

A CASL program consists of a sequence of statements. Each statement is written in a single line and consists of up to four fields: [label] [instruction] [operands] [;comment]
A label is an identifier that is assigned the address of the first word of the instruction. Labels are limited to 6 characters. A label must start at the first column and begin with an upper case letter, followed by upper case letters or digits.
An address in an instruction operand may be specified by a decimal number or by a label.
General purpose registers may be specified using a shorthand notation. The GR part may be omitted, so for example 0 is equivalent to GR0.

CASL supports the following pseudo instructions:

label START [optional entry point]
This instruction begins a program block. The preceding label is mandatory and specifies the name of the block. It is assigned the address of the optional entry point specified by a label defined within the block, and if it is omitted, the address of the beginning of the block. A CASL program can consist of multiple blocks. The block names are global, while the labels defined in a block are local to this block.

END
Marks the end of a program block.

DC ... - Define Constant
Allocates a word (or words) of memory with initialized values. The operand may be a numeric constant or a string of characters.
Numeric operands may be specified in decimal or hexadecimal notation, or by a label. Decimal constants may be signed or unsigned. Hexadecimal constants are unsigned only and preceded with a # character. The value is truncated to 16 bits and stored in a single word of the object program. String operands must be surrounded by apostrophes.

DS n - Define Storage
Allocates the required number of words without initialization. The operand is a decimal number.

EXIT
Terminates the program execution.

CASL includes macro instructions for Input and Output:

IN input buffer, input length
When this instruction is encountered during program execution, the program halts and waits for the user to enter a string of characters. When the user presses the EXE key, program execution continues. The input length contains the string length. Both IN operands are specified by label names. The size of the input buffer must be at least 80 words.

OUT output buffer, output length
The contents of the output buffer is displayed as characters. The output length contains the data size. After displaying the string, the program execution pauses until any key is pressed. Both OUT operands are specified by label names.

Error Messages

Errors detected during assembly (CASL):

OM out of memory
LA label undefined or multiply defined
OC operation error
OR operand error
SO block definition error, for example missing START or END

Run-time errors (COMET):

ST stack overflow/underflow
CD illegal opcode
AD illegal address

CASL Menu

[asmbl]
Assemble the selected sequential file.

[source]
View and edit the sequential file with an empty name. If such file doesn't already exist, it will be created.

[edit]
View and edit the selected sequential file.

[PRT SW]
Select whether to output the assembly listing to a printer.

key EXE
Assemble the selected sequential file then execute the resuling object code from the beginning (i.e. at the entry point of the first block) without asking the user any questions.

COMET Menu

[go]
Run the object code at the specified address.

[dump]
Invokes the following submenu:

[object]
Display the memory contents starting from the specified address. The screen can be scrolled with the up/down arrow keys. The value in the top row can be modified by pressing the left or right arrow key.

[regist]
Display and edit the contents of the registers.

[bpoint]
Specify a breakpoint address. The breakpoint can be cleared by typing an address outside the allowed range, for example -1.

key EXE
Invokes the same function as the menu entry [object], but sets the starting address to #0000 without asking the user.

[edit]
View and edit the source file.

[TR SW]

Select the trace mode allowing single-stepping through the code. The trace information can be directed to a printer (with the menu entry LTRON).

key EXE
Run the object code from the beginning.

Example Programs

```
; Program to solve the Tower of Hanoi puzzle using recursive calls,
; taken from the Japanese Wikipedia
; http://ja.wikipedia.org/wiki/CASL
MAIN  START
    LD    GR0,N
    LD    GR1,A
    LD    GR2,B
    LD    GR3,C
    CALL  HANOI    ;hanoi(3,A,B,C)
    EXIT

; hanoi(N,X,Y,Z)
HANOI CPA   GR0,ONE  ;if N==1 then
    JZE   DISP     ;move it, return
    SUB   GR0,ONE  ;N-1
    PUSH  0,GR2    ;swap GR2 GR3
    LEA   GR2,0,GR3
    POP   GR3
    CALL  HANOI    ;hanoi(N-1,X,Z,Y)
    ST    GR1,MSG1
    ST    GR2,MSG2 ;now GR2 holds Z
    OUT   MSG,LNG  ;'from X to Z'
    PUSH  0,GR2    ;rotate GR1-GR3
    LEA   GR2,0,GR1
    LEA   GR1,0,GR3
    POP   GR3
    CALL  HANOI    ;hanoi(N-1,Y,X,Z)
    PUSH  0,GR2    ;restore registers
    LEA   GR2,0,GR1
    POP   GR1
    ADD   GR0,ONE  ;also restore N
    RET
DISP  ST    GR1,MSG1 ;'from X to Z'
    ST    GR3,MSG2
    OUT   MSG,LNG
    RET

ONE   DC    1
N     DC    3       ;number of disks
LNG   DC    11      ;message length
A     DC    'A'
B     DC    'B'
C     DC    'C'
MSG   DC    'from '
MSG1  DS    1
    DC    ' to '
MSG2  DS    1
    END

; Executing this code yields the following result (where from A to C means to
; move the disk at the top of A to C):
;
```

```
; From A to C
; From A to B
; From C to B
; From A to C
; From B to A
; From B to C
; From A to C
```

; Program to solve the eight queens puzzle,

```
; taken from the Calculator Benchmark web page
; http://www.hpmuseum.org/cgi-sys/cgiwrap/hpmuseum/articles.cgi?read=700
BGN  START
    LEA  GR0,8
    ST   GR0,DIM
    LEA  GR0,0
    LEA  GR1,0
L00  CPA  GR1,DIM
    JZE  L05
    LEA  GR1,1,GR1
    LD   GR3,DIM
    ST   GR3,ARY,GR1
L01  ADD  GR0,ONE
    ST   GR1,TMP
    LD   GR2,TMP
L02  LEA  GR2,-1,GR2
    JZE  L00
    LD   GR3,ARY,GR1
    SUB  GR3,ARY,GR2
    JZE  L04
    JPZ  L03
    EOR  GR3,FFH
    LEA  GR3,1,GR3
L03  ST   GR2,TMP
    ADD  GR3,TMP
    ST   GR1,TMP
    SUB  GR3,TMP
    JNZ  L02
L04  LD   GR3,ARY,GR1
    LEA  GR3,-1,GR3
    ST   GR3,ARY,GR1
    JNZ  L01
    LEA  GR1,-1,GR1
    JNZ  L04
L05  EXIT
ONE  DC   1
FFH  DC   #FFFF
DIM  DS   1
TMP  DS   1
ARY  DS   9
    END

; The result is stored in the array ARY. Also the register GR0 contains the
; number of iterations (876).
```

```
; This program calculates and displays a square root of an integer number
; entered by the user. It illustrates the usage of multiple blocks.
MAIN  START
    IN    BUF1,SIZE1
    LEA   GR1,BUF1
    LD    GR2,SIZE1
    CALL  ATOI
    ST    GR0,TEMP
    LEA   GR1,BUF3
    CALL  ITOA
    LD    GR0,TEMP
    CALL  SQRT
    LEA   GR0,0,GR1
    LEA   GR1,BUF4
    CALL  ITOA
    OUT   BUF2,SIZE2
    EXIT
BUF1  DS    80
SIZE1 DS    1
BUF2  DC    'SQRT ('
BUF3  DS    5
    DC    ') = '
BUF4  DS    5
SIZE2 DC    20
TEMP  DS    1
    END
```

```
; convert a string to an unsigned integer in GR0
; string address in GR1, length in GR2
ATOI  START
    LEA   GR0,0
L01   LEA   GR2,-1,GR2
    JMI   L02
    LD    GR3,0,GR1
    LEA   GR3,-48,GR3
    JMI   L03
    ST    GR3,TEMP1
    LEA   GR3,-10,GR3
    JPZ   L03
    SLL   GR0,1
    ST    GR0,TEMP2
    SLL   GR0,2
    ADD   GR0,TEMP2
    ADD   GR0,TEMP1
    LEA   GR1,1,GR1
    JMP   L01
L02   LEA   GR2,1,GR2
L03   RET
TEMP1 DS    1
TEMP2 DS    1
    END
```

```
; convert an unsigned integer GR0 to decimal
; result at the address GR1
ITOA  START
```

```
    LEA   GR2,4
L01   LD    GR3,ZERO
L02   CPL   GR0,TENS,GR2
    JMI   L03
    SUB   GR0,TENS,GR2
    LEA   GR3,1,GR3
    JMP   L02
L03   ST    GR3,0,GR1
    LEA   GR1,1,GR1
    LEA   GR2,-1,GR2
    JNZ   L01
    ADD   GR0,ZERO
    ST    GR0,0,GR1
    RET
ZERO  DC    '0'
TENS  DC    1
    DC    10
    DC    100
    DC    1000
    DC    10000
    END
```

; square root of an unsigned integer

```
; radicand = GR0, root = GR1
SQRT  START
    LEA   GR1,0      ;root
    LEA   GR2,0      ;remainder
    LEA   GR3,8      ;number of root bits
; shift 2 bits from the radicand to the remainder
L01   SLL   GR2,2
    ST    GR0,TEMP1
    SRL   GR0,14
    ST    GR0,TEMP2
    ADD   GR2,TEMP2
    LD    GR0,TEMP1
    SLL   GR0,2
; try to subtract 4*root+1 from the remainder
    SLL   GR1,2
    LEA   GR1,1,GR1
    ST    GR1,TEMP2
    SRL   GR1,1
    CPL   GR2,TEMP2
    JMI   L02
    SUB   GR2,TEMP2
    LEA   GR1,1,GR1
; next bit of the root
L02   LEA   GR3,-1,GR3
    JNZ   L01
    RET
TEMP1 DS    1
TEMP2 DS    1
    END
```

8-3 CASL From the Original Manual

get it start with ON / CASL

第3章
CASL編

```
                    〈 CASL 〉
F 0 1 2 3 4 5 6 7 8 9          3355B
F1>Assemble/Source/Cal
```

Aキー(Assemble)…ソースプログラムのアセンブル
Sキー(Source)……ソースの作成・編集
Cキー(Cal)……マニュアル計算モードへ戻る

Gキー(Go)	オブジェクトプログラムの実行画面に移ります。
Dキー(Dump)	ダンプのメニュー画面に移ります。
Sキー(Source)	エディタに入り、ソースプログラムを表示します。
Cキー(Cal)	マニュアル計算モードに入ります。
Pキー(Print)	プリンタのON／OFFを指定します。
Tキー(Trace)	トレースのON／OFFを指定します。
EXEキー(実行)	オブジェクトプログラムを実行します。

a CASL Project „Jozan"

［プログラム例］ 割り算のプログラム（A÷B＝SHO余りAMARI）

ラベル	命令コード	オペランド	説　明
JOZAN	START		割り算(除算)プログラムを開始
	LEA	GR1, 0	GR1に商を入れる、初期値0
	LD	GR0, A	GR0にA番地の内容を入れる
LOOP	SUB	GR0, B	GR0からB番地の内容を引く
	JMI	ANS	結果が負ならばANS番地に飛ぶ
	LEA	GR1, 1, GR1	GR1に1を加える
	JMP	LOOP	LOOP番地に飛ぶ
ANS	ADD	GR0, B	GR0にB番地の内容を加える
	ST	GR0, AMARI	GR0を余りとしてAMARIに格納
	ST	GR1, SHO	GR1を商としてSHOに格納
	EXIT		プログラムの実行終了
A	DC	13	割られる数13
B	DC	5	割る数5
SHO	DS	1	商を格納する番地
AMARI	DS	1	余りを格納する番地
	END		プログラムの終了

アセンブル後のメニュー

```
〈 CASL 〉

Go/Dump/Source/Cal/Print/Trace
 PC:0000  JOZAN   [off] [off]
```

ダンプメニュー

```
〈 CASL 〉

Object/Register/Break point
```

ダンプのキー操作

EXE または ⇩ キー	下スクロール
⇧ キー	上スクロール

オブジェクトのダンプ

```
0000   JOZAN   1210
0001           0000
0002           1000
0003           0014
```

レジスタの初期値

```
GR0:0000  0          GR1:0000  0
GR2:0000  0          GR3:0000  0
GR4:FFFF  65535      FR  :0
GR0?_
```

```
〈  CASL  〉

Go/Dump/Source/Cal/Print/Trace
 PC:0000   JOZAN        [off]  [on]
```

```
[G][EXE]   0000   JOZAN   LEA   GR1,0000
           0000 0000 0000 0000 FFFF 1
[EXE]      0002           LD    GR0,0014
           000D 0000 0000 0000 FFFF 1
[EXE]      0004   LOOP    SUB   GR0,0015
           0008 0000 0000 0000 FFFF 0
[EXE]      0006           JMI   000C
           0008 0000 0000 0000 FFFF 0
[EXE]      0008           LEA   GR1,0001,GR1
           0008 0001 0000 0000 FFFF 0
[EXE]      000A           JMP   0004
           0008 0001 0000 0000 FFFF 0
[EXE]      0004   LOOP    SUB   GR0,0015
           0003 0001 0000 0000 FFFF 0
                  ⋮
           以下、表示は省略します。
```

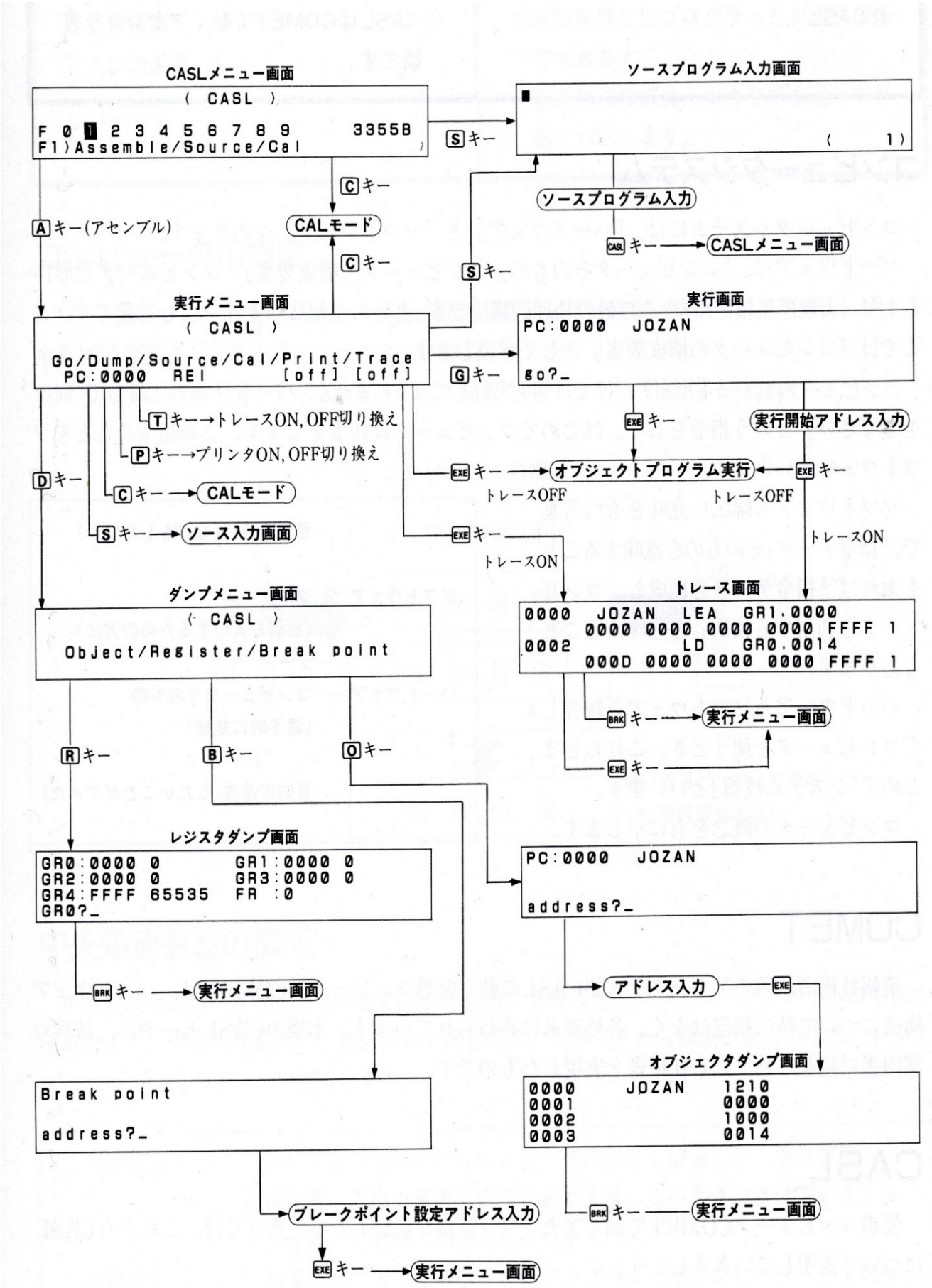
表示の見方

```
PC   ラベル   命令コード  オペランド
↓     ↓        ↓          ↓
0000  JOZAN    LEA    GR1,0000
      0000     0000   0000   0000   FFFF   1
      ↑        ↑      ↑      ↑      ↑      ↑
      GR0      GR1    GR2    GR3    GR4    FR
```

GR0 ～ GR4 } レジスタの内容

表　示	本機	仕様書
FR　フラグの値	0	00
	1	01
	2	10

仕様書とは試験センターのCOMET仕様書のことです。


CASLメニュー画面
(CASL)
F 0 1 2 3 4 5 6 7 8 9 3355B
F1)Assemble/Source/Cal
Sキー
ソースプログラム入力画面
(1)
Cキー
CALモード
ソースプログラム入力
Aキー(アセンブル)
CASLキー
CASLメニュー画面
Cキー
Sキー
実行メニュー画面
(CASL)
Go/Dump/Source/Cal/Print/Trace
PC:0000 REI [off] [off]
実行画面
PC:0000 JOZAN
go?_
Gキー
Tキー→トレースON,OFF切り換え
Pキー→プリンタON,OFF切り換え
Dキー
Cキー
CALモード
Sキー
ソース入力画面
EXEキー
実行開始アドレス入力
EXEキー
オブジェクトプログラム実行
EXEキー
トレースOFF
トレースOFF
EXEキー
トレースON
トレースON
ダンプメニュー画面
(CASL)
Object/Register/Break point
トレース画面
0000 JOZAN LEA GR1,0000
0000 0000 0000 0000 FFFF 1
0002 LD GR0,0014
000D 0000 0000 0000 FFFF 1
BRKキー
実行メニュー画面
EXEキー
Rキー
Bキー
Oキー
レジスタダンプ画面
GR0:0000 0 GR1:0000 0
GR2:0000 0 GR3:0000 0
GR4:FFFF 65535 FR :0
GR0?_
PC:0000 JOZAN
address?_
BRKキー
実行メニュー画面
アドレス入力
EXE
オブジェクタダンプ画面
0000 JOZAN 1210
0001 0000
0002 1000
0003 0014
Break point
address?_
ブレークポイント設定アドレス入力
BRKキー
実行メニュー画面
EXEキー
実行メニュー画面

The CASL Code in the Original Manual

START、END、DC、DS 擬似命令

命令名	開始　START	書　　式 LABEL START〔adr〕
機能	プログラムの先頭に必ず書かなければなりません。adrのラベル名の番地から実行され、省略されるとプログラムの先頭から実行されます。	
命令名	終了　END	書　　式 END
機能	プログラムの終了を表わします。プログラムの最後やサブルーチンの最後には必ず書かなければなりません。ラベル、オペランドはありません。	

メインプログラムの記述例

```
MAIN START
    プログラム

    EXIT
    END
```

サブルーチンの記述例

```
SUBR START
    プログラム

    RET
    END
```

メインプログラムを記述するときは、先頭にSTART命令を書き、次にプログラムの内容を書き最後に処理を終了させるEXITとENDを書きます。

サブルーチンも同様に、先頭にSTARTを最後にENDを書きます。

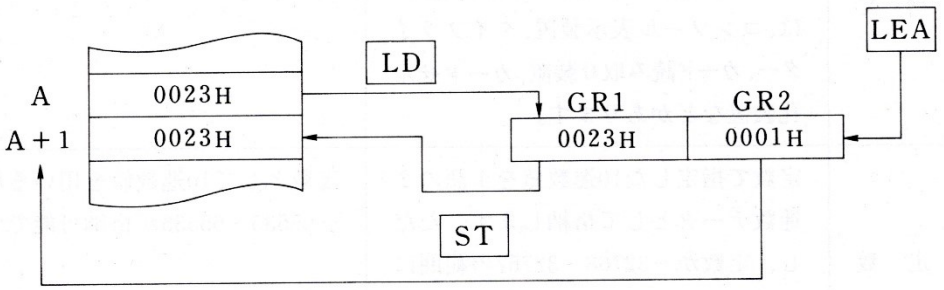
命令名	定数定義　Define Constant	書　　式 〔LABEL〕DC 定数
機能	メモリーに定数データを格納します。定数には、10進(-65535≦n≦65535)、16進(4桁)、文字定数、アドレス定数の四種類が使用できます。	
命令名	領域定義　Define Storage	書　　式 〔LABEL〕DS n
機能	n(≧0)によって指定した語数だけ連続領域を確保します。nが0のときは領域は確保されませんが、ラベルは有効です。	

LD、ST、LEA 機械語命令:ロードストア命令、ロードアドレス命令

命令名	ロード Load	書　　式 〔LABEL〕 LD GR, adr 〔, XR〕
機能	実効アドレスによって示された番地の内容を、レジスタ(GR) に設定します。	
命令名	ストア STore	書　　式 〔LABEL〕 ST GR, adr 〔, XR〕
機能	レジスタ(GR)の内容を、実効アドレスによって示された番地に格納します。	
命令名	ロードアドレス Load Effective Address	書　　式 〔LABEL〕 LEA GR, adr 〔, XR〕
機能	実効アドレスによって示される番地の内容または10進定数を、レジスタに設定します。 また、このときの値によってFRの値を設定します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、A番地に格納されている数値がGR1に格納されます(LD)。次に、GR 2 に定数 1 が格納されます(LEA)。最後に、A+GR2の内容の番地、すなわちA + 1 番地にGR1の内容35が格納されます(ST)。
	LD	GR1, A	
	LEA	GR2, 1	
	ST	GR1, A, GR2	
	EXIT		
A	DC	35	
	DS	1	
	END		



ADD、SUB 機械語命令：算術演算命令

命令名	算術加算 ADD arithmetic	書　　式 〔LABEL〕ADD GR, adr〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容を算術加算してその結果をGRに設定します。演算結果によって、FRを設定します。	
命令名	算術減算 SUBtract arithmetic	書　　式 〔LABEL〕SUB GR, adr〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容を算術減算してその結果をGRに設定します。演算結果によって、FRを設定します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、A番地に格納されている16 進数値＃0029がGR1に設定されます。次に、この値にB番地の内容である＃000Eを算術加算した値＃0037が、GR 1 に格納されます(ADD)。同様に、C番地の内容である#001AをGR1から算術減算した値＃001Dが、GR1に格納されます。(SUB) 最後に、GR1の内容を ANS 番地に格納します。
	LD	GR1, A	
	ADD	GR1, B	
	SUB	GR1, C	
	ST	GR1, ANS	
	EXIT		
A	DC	＃0029	
B	DC	＃000E	
C	DC	＃001A	
ANS	DS	1	
	END		

```
   0029
+) 000E   (ADD)
   0037
−) 001A   (SUB)
   001D
```

AND、OR、EOR 機械語命令： 論理演算命令

命令名	論理積 AND	書　　式 〔LABEL〕 AND GR, adr 〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容のビットごとの論理積を、GRに設定します。演算結果によって、FRを設定します。	
命令名	論理和 OR	書　　式 〔LABEL〕 OR GR, adr 〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容のビットごとの論理和を、GRに設定します。演算結果によって、FRを設定します。	
命令名	排他的論理和 Exclusive OR	書　　式 〔LABEL〕 EOR GR, adr 〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容のビットごとの排他的論理和を、GRに設定します。演算結果によって、FRを設定します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、GR1に16進定数＃5555が格納されます。次に、それとA番地の内容である＃137Fと論理積を取った結果＃1155がGR1に格納されます。 ＃5555 ＝ 0101 0101 0101 0101 AND) ＃137F ＝ 0001 0011 0111 1111 ＃1155 ＝ 0001 0001 0101 0101 (このプログラム例でのANDと同様にORとEORも使用できます。)
	LD	GR1, A	
	AND	GR1, B	
	EXIT		
A	DC	＃5555	
B	DC	＃137F	
	END		

コラム●論理演算●

論理積(AND)では2つの値とも1のときのみ1になり、論理和(OR)ではどちらか一方が1ならば1になり、排他的論理和(EOR)では2つの値が異なるときのみ1になります。これを図で表わすと右のようになります。

演算値		AND	OR	EOR
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

CPA、CPL　機械語命令：比較演算命令

命令名	算術比較 ComPare Arithmetic	書　　式 〔LABEL〕CPA GR, adr〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容を算術比較し、その結果により、FRの値を設定します。	
命令名	論理比較 ComPare Logical	書　　式 〔LABEL〕CPL GR, adr〔, XR〕
機能	GRの内容と実効アドレスによって示される番地の内容を論理比較し、その結果により、FRの値を設定します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、GR1に定数－32が格納されます。次に、CPA命令によりA番地の内容#20＝$(32)_{10}$と算術比較されます。この場合は、 GR1$(-32)_{10}$＜A番地の内容$(32)_{10}$ なのでFRのビット値は$(10)_2$になります。 同様に、次のCPL命令でも比較が行われますが、ここでは以下に示したようなビット構成による論理比較が行なわれます。この場合は、 GR1$(FFE0)_{16}$＞A番地の内容$(0020)_{16}$ なのでFRのビット値は$(00)_2$になります。 GR1＝$(-32)_{10}$の ビット構成　1111 1111 1110 0000 A番地の内容#20＝$(32)_{10}$の ビット構成　0000 0000 0010 0000
	LEA	GR1, －32	
	CPA	GR1, A	
	CPL	GR1, A	
	EXIT		
A	DC	#0020	
	END		

SLA、SRA、SLL、SRL 機械語命令:シフト演算命令

命令名	算術左シフト Shift Left Arithmetic	書　　式 〔LABEL〕SLA GR, adr〔, XR〕
機能	GRの内容を実効アドレスの数だけ左にシフトします。ただし、最上位ビット（符号ビット)はシフト前の値が保存され、空きビットには0が入ります。	
命令名	算術右シフト Shift Right Arithmetic	書　　式 〔LABEL〕SRA GR, adr〔, XR〕
機能	GRの内容を実効アドレスの数だけ右にシフトします。ただし、最上位ビットはシフト前の値が保存され、その値で空きビットが埋められます。	
命令名	論理左シフト Shift Left Logical	書　　式 〔LABEL〕SLL GR, adr〔XR〕〔, XR〕
機能	GRの内容を実効アドレスで示された数だけ左にシフトします。 シフトの結果による空きビットには0が入ります。	
命令名	論理右シフト Shift Right Logical	書　　式 〔LABEL〕SRL GR, adr〔, XR〕
機能	GRの内容を実効アドレスで示された数だけ右にシフトします。 シフトの結果による空きビットには0が入ります。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、GR1にA番地の内容である7FFF(16進)が格納されます。次にGR1の内容が左に5桁算術シフトされます。その結果GR1には7FE0(16進)が残ります。フラグは正ですので$(00)_2$になります。 7FFF = 0111 1111 1111 1111 0111 1111 1110 0000 =7FE0
	LD	GR1, A	
	SLA	GR1, 5	
	EXIT		
A	DC	#7FFF	
	END		

JPZ、JMI、JNZ、JZE、JMP 機械語命令:分岐命令

命令名	正分岐 Jump on Plus or Zero	書　　式 〔LABEL〕JPZ adr〔,XR〕
機能	FRのビット値が$(00)_2$「正」か$(01)_2$「零」のとき、実効アドレスに分岐します。それ以外のときは、次の命令に進みます。	
命令名	負分岐 Jump on MInus	書　　式 〔LABEL〕JMI adr〔,XR〕
機能	FRのビット値が$(10)_2$「負」のとき、実効アドレスに分岐します。それ以外のときは、次の命令に進みます。	
命令名	非零分岐 Jump on Non Zero	書　　式 〔LABEL〕JNZ adr〔,XR〕
機能	FRのビット値が$(00)_2$「正」か$(10)_2$「負」のとき、実効アドレスに分岐します。それ以外のときは、次の命令に進みます。	
命令名	零分岐 Jump on ZEro	書　　式 〔LABEL〕JZE adr〔,XR〕
機能	FRのビット値が$(01)_2$「零」のとき、実効アドレスに分岐します。それ以外のときは、次の命令に進みます。	
命令名	無条件分岐 unconditional JuMP	書　　式 〔LABEL〕JMP adr〔,XR〕
機能	FRのビット値に関わらず、無条件に実効アドレスに分岐します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		このプログラムは、複数のデータを読み込み、そのデータの中から正の最小値を探し、そのデータを保存するというものです。ただし、「3」というデータがある場合は、そのデータだけを除外して処理を行ないます。 はじめに、CASLであつかえる符号つきの最大値7FFFHをMIN番地に格納しておきます。DATA番地からのデータをGR0に読み込み、それが、 ・A番地の内容1より小さい（0や負のデータは切り捨てる）。 ・B番地の内容3に等しい（3は除外する）。 ・MIN番地の内容より大きいか等しい。 とき、L2にジャンプします。そうでないときはGR0の内容をMINに格納します。L2以下では、GR1を1だけ増加して次のデータを読み込めるようにします。GR1がC番地の内容（5＝データ数）より小さいときはL1にジャンプして再び比較を行ないます。 最後に3以外の正の最小値4がMIN番地に格納されてプログラムが終了します。
	LD	GR0, MAX	
	ST	GR0, MIN	
	LEA	GR1, 0	
L1	LD	GR0,DATA,GR1	
	CPA	GR0, A	
	JMI	L2	
	CPA	GR0, B	
	JZE	L2	
	CPA	GR0,MIN	
	JPZ	L2	
	ST	GR0,MIN	
L2	LEA	GR1,1,GR1	
	CPA	GR1,C	
	JMI	L1	
	EXIT		
A	DC	1	
B	DC	3	
C	DC	5	
MAX	DC	#7FFF	
MIN	DS	1	
DATA	DC	5	
	DC	−1	
	DC	0	
	DC	3	
	DC	4	
	END		

PUSH、POP　機械語命令:スタック操作命令
CALL、RET　機械語命令:コールリターン命令

命令名	プッシュ PUSH effective address	書　式 〔LABEL〕PUSH adr〔,XR〕
機能	スタックポインタ(SP)から1をアドレス減算したあと、実効アドレスによって示された番地をSPが示す番地に格納します。	
命令名	ポップ POP up	書　式 〔LABEL〕POP GR
機能	スタックポインタ(SP)の示す番地の内容をGRに設定し、SPに1をアドレス加算します。	
命令名	コール CALL subroutine	書　式 〔LABEL〕CALL adr〔,XR〕
機能	実効アドレスに示される番地に分岐し、処理の流れがサブルーチンに移されます。このとき、戻り番地がスタックに保存されます。	
命令名	リターン RETurn from subroutine	書　式 〔LABEL〕RET
機能	スタックに保存されていた戻り番地がプログラムカウンタにセットされ、サブルーチンからメインプログラムに処理の流れが戻されます。	

プログラム例

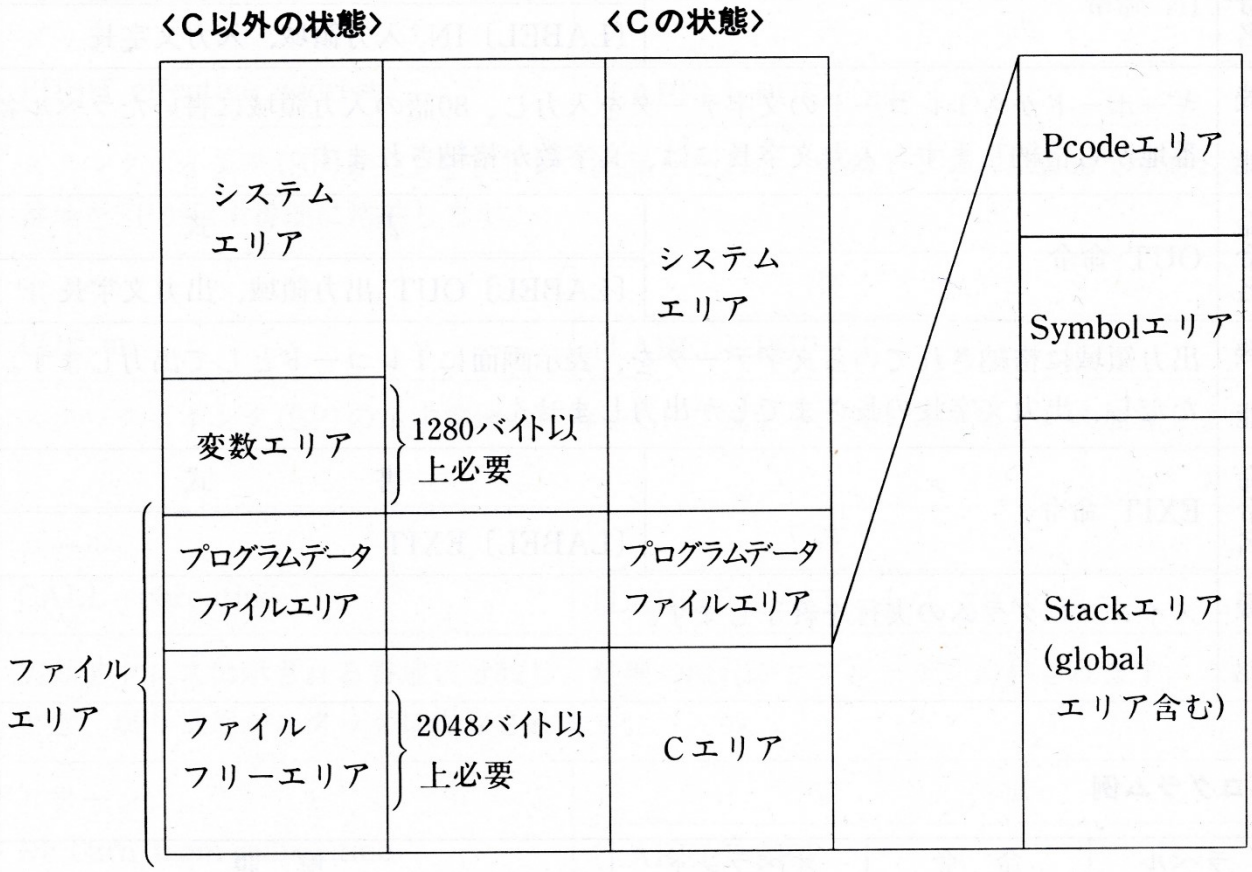
ラベル	命　令	オペランド	解　説
MAIN	START		まず、メインプログラムはサブルーチンを呼び出すだけの構成になっています。サブルーチンではPUSH、POP命令を行ない、スタックを通してGR1に100を入れます。そして、RET命令によってメインプログラムに戻ります。
	CALL	SUBR	
	EXIT		
	END		
SUBR	START		
	PUSH	100	
	POP	GR1	
	RET		
	END		

IN、OUT、EXIT　マクロ命令

命令名	IN 命令	書　　式
		〔LABEL〕IN 入力領域、入力文字長
機能	キーボードから1レコードの文字データを入力し、80語の入力領域に書いたラベル名の番地から格納します。入力文字長には、文字数が格納されます。	
命令名	OUT 命令	書　　式
		〔LABEL〕OUT 出力領域、出力文字長
機能	出力領域に格納されている文字データを、表示画面に1レコードとして出力します。ただし、出力文字長の長さまでしか出力しません。	
命令名	EXIT 命令	書　　式
		〔LABEL〕EXIT
機能	メインプログラムの実行を終了します。	

プログラム例

ラベル	命　令	オペランド	解　説
BGN	START		まず、A番地に80語の領域をとり、IN命令によりキーボードから文字列がA番地から一文字ずつ順番に格納され、B番地には文字数が格納されます。次に、OUT命令によりA番地から格納されている文字列を、B番地に示す文字数だけ読み出し、画面に表示します。 最後にEXIT命令でプログラムが終了します。
	IN	A, B	
	OUT	A, B	
	EXIT		
A	DS	80	
B	DS	1	
	END		



● C使用時のユーザーズエリア(プログラム・データエリア＋Cエリア)

最大　8KB RAM時　3611バイト

40KB RAM時　36379バイト

※最大とは、CLEAR，1280バイトの時です。

デフォルトは 8～16KB時　CLEAR，1536

40KB時　CLEAR，8192　です。

● Cを動作するには、変数エリアが最低1280バイトファイルフリーエリアが最低2048バイト必要です。

8-4 CASL from Inet-Site: http://www5a.biglobe.ne.jp …

The next sides are tranlated from the Inet-side: http://www5a.biglobe.ne.jp/~teamcasl/caslkozatop.htm
The TeamCASL pages präsent the CASL II instruction.

The CASL introduction corner – Table Contents

1. Basic structure of CASL II Program

Basic rules of CASLII program

- Write in labels, instructions, and operands (arguments)
- Command words are written in uppercase letters
- Start with START command, end program with END command
- Label the START instruction line

Examples 1 and 2 show the basic structure of the program.

Example 1

label	order	operand
PROG1	START	GO
DATA1	DC	1
DATA2	DC	Two
ANS	DS	1
GO	LD	GR0, DATA1
	ADDA	GR0, DATA2
	ST	GR0, ANS
	END	

Example 2.

PROG2	START	
	LD	GR0, DATA1
	ADDA	GR0, DATA2
	ST	GR0, ANS
	RET	
DATA1	DC	1
DATA2	DC	Two
ANS	DS	1
	END	

2. Load / store instruction

Assembler languages such as CASL first read data from memory into a storage device called a register, and then perform calculations.

This section describes the instructions for exchanging data between memory and registers.

(1) LD instruction Instruction to read data from memory to register

Description method

label	LD	GRx, address [, GRx]

The contents of the address are stored in GRx.
The register described after the address specifies the index register. (Optional)
The relative position of the address can be specified by using the index register.
Omit the index address specification.

When trying to process data in a program, you must use the LD instruction.

Program example.

PROG_LD ADR GO	START DC LD END	GO; Start processing from label GO 10; Define constant 10 GR0, ADR; 10 is stored in GR0

(2) ST instruction Instruction to write data in the register to memory

Description method

label	ST	GRx, address [, GRx]

This is an instruction to write the data in the register to the memory.

Program example.

PROG_ST ANS GO	START DS ST END	GO 1; Secures one word length for data storage area GR1, ANS; GR1 content in ANS

(3) LAD instruction Instruction to store address directly

Description method

label	LAD	GRx, address [, GRx]

Store the address in a register.

Difference from LD instruction The LD instruction reads the contents of the specified address .
The LAD instruction reads the specified address .

Program example.

PROG_LAD ADR GO	START DC LAD END	GO 1 GR2, ADR; GR2 contains ADR address instead of 1

3. Operation instruction

CASL provides arithmetic and logic instructions.

(1) ADDA arithmetic addition instruction

Description method

label	ADDA	r, address [, x]

Adds the contents of the address to the value stored in r and stores it in r.
In the expression, r = r + the contents of the address.

Program example.

PRG_ADDA	START	
	LD	GR0, DATA1; Read data to register
	ADDA	GR0, DATA2; Add contents of DATA2 to GR0
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program
DATA1	DC	1; Define data
DATA2	DC	2; Define data
ANS	DS	1; Secure data storage area
	END	

(2) ADDL instruction Logical addition instruction

Description method

label	ADDL	r, address [, x]

Adds the contents of the address to the value stored in r and stores it in r.
Works the same as r = r + contents of address.

The difference from the ADDA instruction is handled as if there is no sign (+,-).
In other words, we don't think about minus.

Program example.

PRG_ADDL	START	
	LD	GR0, DATA1; Read data to register
	ADDL	GR0, DATA2; Add contents of DATA2 to GR0
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program
DATA1	DC	1; Define data
DATA2	DC	2; Define data
ANS	DS	1; Secure data storage area
	END	

(3) SUBA instruction Arithmetic subtraction instruction

Description method

label	SUBA	r, address [, x]

This is equivalent to the expression r = r – address content.

Program example.

PRG_SUBA	START	Start processing from GO
DATA1	DC	3; Data definition
DATA2	DC	1; data definition
ANS	DS	1; data definition
GO	LD	GR2, DATA1; Load the contents of DATA1 into GR2

	SUBA	GR2, DATA2; Subtract the contents of DATA2 from GR2
	ST	GR2, ANS; Store result in ANS
	END	

(4) SUBL instruction Logical subtraction instruction

Description method

label	SUBL	r, address [, x]

This is equivalent to the expression r = r – address contents.

Program example.

PRG_SUBL	START	Start processing from GO
DATA1	DC	3; Data definition
DATA2	DC	1; data definition
ANS	DS	1; data definition
GO	LD	GR2, DATA1; Load the contents of DATA1 into GR2
	SUBL	GR2, DATA2; Subtract the contents of DATA2 from GR2
	ST	GR2, ANS; Store result in ANS
	END	

(5) AND instruction Logical product instruction

Description method

label	AND	r, address [, x]

Performs a logical AND with r and the contents of the address, and stores the result in r.

Program example (Example of a program that retrieves the first bit information of DATA)

PRG_AND	START	Start processing from GO
DATA	DC	#FFFF; Data definition
MASK	DC	# 0001; Data definition
ANS	DS	1; data definition
GO	LD	GR2, DATA1; Load the contents of DATA into GR2
	AND	GR2, MASK; Perform logical AND with the contents of GR2 and MASK
	ST	GR2, ANS; Store result in ANS
	END	

(6) OR instruction OR instruction

Description method

label	OR	r, address [, x]

Perform a logical sum of r and the contents of the address, and store the result in r.

Program example (Example of overlapping the contents of DATA and MASK)

PRG_AND	START	Start processing from GO
DATA	DC	# 0FF0; Data definition
MASK	DC	# 3001; Data definition
ANS	DS	1; data definition
GO	LD	GR2, DATA1; Load the contents of DATA into GR2
	OR	GR2, MASK; Perform a logical OR operation on the contents of GR2 and MASK
	ST	GR2, ANS; Store result in ANS
	END	

(7) XOR instruction Exclusive OR instruction

Description method

label	XOR	r, address [, x]

Performs an exclusive OR operation on r and the contents of the address, and stores the result in r.

Program example (Program example for bit-reversing the contents of DATA)

PRG_AND	START	Start processing from GO
DATA	DC	# 1010; Data definition
MASK	DC	#FFFF; Data definition
ANS	DS	1; data definition
GO	LD	GR2, DATA1; Load the contents of DATA into GR2
	XOR	GR2, MASK; Perform exclusive OR on the contents of GR2 and MASK
	ST	GR2, ANS; Store result in ANS
	END	

4. Comparison operation instruction

In CASL, the comparison instruction only performs a comparison operation.
Performs the same operation as IF in combination with a branch instruction.
Here, only the comparison operation instruction is described.
See the branch instruction for the specific selection syntax. (⑤ branch instruction)

(1) CPA instruction Arithmetic comparison instruction

Description method

label	CPA	r, address [, x]

This instruction internally subtracts (r-the contents of the address) and stores the result in the flag register.

The difference from the subtraction instruction is that the result is not the value of the subtraction, and whether the result of the subtraction is positive, negative, or zero is stored in the flag register.

Program example.

```
PRG_CPA   START
          LD      GR0, DATA1; Read data to register
          CPA     GR0, DATA2; Compare the contents of DATA2 to GR0
          RET     ;The end of the program
DATA1     DC      1; Define data
DATA2     DC      2; Define data
          END
```

(2) CPL instruction Logical comparison instruction

Description method

label	CPL	r, address [, x]

This instruction internally subtracts (r-the contents of the address) and stores the result in the flag register.

The difference from the subtraction instruction is that the result is not the value of the subtraction, and whether the result of the subtraction is positive, negative, or zero is stored in the flag register.

A logical operation is an operation in which the contents of an address are treated as numbers that do not handle signs (positive numbers).

Program example.

```
PRG_CPL   START
          LD      GR0, DATA1; Read data to register
          CPA     GR0, DATA2; Compare the contents of DATA2 to GR0
          RET     ;The end of the program
DATA1     DC      1; Define data
DATA2     DC      2; Define data
          END
PRG_CPA   START
          LD      GR0, DATA1; Read data to register
```

5. Branch Instruction

In CASL, a branch instruction is combined with a comparison instruction to create an IF structure. In addition to unconditional branch instructions, there are conditional branches that branch depending on the value of the flag register.

(1) JPL instruction Instruction to branch if the flag register is positive

Description method

label	JPL	Address [, x]

Branches to the address when the value of the flag register is positive.

Program example.
(Compares the contents of DATA1 and DATA2, ends if DATA1> DATA2, adds if DATA1 ≦ DATA2)

```
PRG_JPL   START
          LD     GR0, DATA1; Read data to register
          CPA    GR0, DATA2; Compare the contents of DATA2 to GR0
          JPL    JMP; If CPA result is positive, go to JMP
          ADDA   GR0, DATA2; Addition
          ST     GR0, ANS; Store addition result in ANS
JMP       RET    ; End of program * Here is the jump destination
DATA1     DC     1; Define data
DATA2     DC     2; Define data
ANS       DS     1
          END
```

(2) JMI instruction Instruction to branch if the flag register is negative

Description method

label	JMI	Address [, x]

Branches to the address when the value of the flag register is negative.

Program example.
(Compares the contents of DATA1 and DATA2, ends if DATA1 <DATA2, subtracts if DATA1 ≧ DATA2)

```
PRG_JMI   START
          LD     GR0, DATA1; Read data to register
          CPA    GR0, DATA2; Compare the contents of DATA2 to GR0
          JMI    JMP; If CPA result is negative, go to JMP
          SUBA   GR0, DATA2; Subtraction
          ST     GR0, ANS; Store addition result in ANS
JMP       RET    ; End of program * Here is the jump destination
DATA1     DC     1; Define data
DATA2     DC     2; Define data
ANS       DS     1
          END
```

(3) JNZ instruction Instruction to branch if the flag register is not zero

Description method

label	JNZ	Address [, x]

Branches to the address when the value of the flag register is not zero.

Program example.
(Compares the contents of DATA1 and DATA2, ends if DATA1 <> DATA2, and adds if DATA1 = DATA2)

```
PRG_JNZ   START
          LD     GR0, DATA1; Read data to register
          CPA    GR0, DATA2; Compare the contents of DATA2 to GR0
          JNZ    JMP; If CPA result is not zero, go to JMP
          ADDA   GR0, DATA2; Addition
          ST     GR0, ANS; Store addition result in ANS
JMP       RET    ; End of program * Here is the jump destination
DATA1     DC     1; Define data
DATA2     DC     2; Define data
ANS       DS     1
          END
```

(4) JZE instruction Instruction to branch if the flag register is positive

Description method

label	JZE	Address [, x]

Branch to the address when the value of the flag register is zero.

Program example.
(Compares the contents of DATA1 and DATA2, ends if DATA1 = DATA2, adds if DATA1 <> DATA2)

```
PRG_JZE   START
          LD     GR0, DATA1; Read data to register
          CPA    GR0, DATA2; Compare the contents of DATA2 to GR0
          JZE    JMP; If CPA result is zero, go to JMP
          ADDA   GR0, DATA2; Addition
          ST     GR0, ANS; Store addition result in ANS
JMP       RET    ; End of program * Here is the jump destination
DATA1     DC     1; Define data
DATA2     DC     2; Define data
ANS       DS     1
          END
```

(5) JOV instruction Instruction to branch if the flag register overflows

Description method

label	JPL	Address [, x]

Branches to the address when the value of the flag register is positive.

Program example.

```
PRG_JOV   START
          LD     GR0, DATA1; Read data to register
          ADDA   GR0, DATA2; Add contents of DATA2 to GR0
          JOV    JMP; If the addition result overflows, go to JMP
          ST     GR0, ANS; Store addition result in ANS
JMP       RET    ; End of program * Here is the jump destination
DATA1     DC     #FFFF; Define data
DATA2     DC     1; Define data
ANS       DS     1
          END
```

(6) JUMP instruction Instruction that branches unconditionally

Description method

label	JUMP	Address [, x]

Branch to address unconditionally.

Program example. (3x2 calculation program)

PRG_JUMP	START	
	LAD	GR1,0; Set GR1 to 0
LOOP	CPA	GR1, LIMIT; Compare the contents of GR1 and LIMIT
	JPL	OWARI; to OWARI
	JZE	OWARI; to OWARI
	ADDA	GR0, DATA
	LAD	GR1,1, GR1; Count up
	JUMP	LOOP
JMP	ST	GR0, ANS
	RET	; End of program * Here is the jump destination
DATA	DC	3; Define data
LIMIT	DC	2; Define data
ANS	DS	1
	END	

6. Shift operation instruction

CASL provides an operation instruction to perform a bit shift.

Multiplication and division can be performed by combining shift operations.

(1) SLA instruction Instruction to perform arithmetic left shift.

Description method

label	SLA	r, address [, x]

The data in r is shifted to the left by the number of bits specified by the address , leaving the sign bit unchanged . Empty bits are filled with 0.

Program example.
(The contents of DATA are shifted left by 2 bits. Perform 8 × 4.)

PRG_SLA	START	
	LD	GR0, DATA; Read data into register
	SLA	GR0,2; Shift left by 2 bits
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program
DATA	DC	# 0008; Define data
ANS	DS	1
	END	

(2) SRA instruction This instruction performs an arithmetic right shift.

Description method

label	SRA	r, address [, x]

The data in r is shifted right by the number of bits specified by the address , leaving the sign bit as it is . The vacant bits are the same as the sign bits.

Program example.
(The contents of DATA are shifted right by 2 bits. Perform 8 ÷ 4.)

PRG_SRA	START	
	LD	GR0, DATA; Read data into register
	SRA	GR0,2; shift right by 2 bits
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program

DATA	DC	# 0008; Define data
ANS	DS	1
	END	

(3) SLL instruction Instruction to perform logical left shift.

Description method

label	SLL	r, address [, x]

The data in r is shifted to the left by the number of bits specified by the address without regard to the sign bit . Empty bits are filled with 0.

Program example.
(The contents of DATA are shifted left by 2 bits.)

PRG_SLL	START	
	LD	GR0, DATA; Read data into register
	SLL	GR0,2; Shift left by 2 bits
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program
DATA	DC	# 0008; Define data
ANS	DS	1
	END	

(4) SRL instruction Instruction to perform logical right shift

Description method

label	SRL	r, address [, x]

The data in r is shifted to the left by the number of bits specified by the address without regard to the sign bit . Empty bits are filled with 0.

Program example.
(The contents of DATA are shifted right by 2 bits.)

PRG_SRL	START	
	LD	GR0, DATA; Read data into register
	SRL	GR0,2; shift right by 2 bits
	ST	GR0, ANS; Store result in ANS
	RET	;The end of the program
DATA	DC	# 0008; Define data
ANS	DS	1
	END	

7. Stack operation instructions

COMET has a memory area called a stack.
The stack has a special way of remembering that the data stored later is retrieved first.
By using the stack, you can reverse the order of the data and use it in various ways.

(1) PUSH instruction An instruction to store data on the stack.

Description method

label	PUSH	Address [, x]

Store the address on the stack and store the address in the stack pointer.

(2) POP instruction An instruction to retrieve data from the stack.

Description method

label	POP	r

Reads the data stored in the stack into r.

Program example (Change the order of DATA1 and DATA2)

```
PUSHPOP   START
          LD      GR1, DATA1
          LD      GR2, DATA2
          PUSH    0, GR1
          PUSH    0, GR2
          POP     GR1
          POP     GR2
          RET
DATA1     DC      1
DATA2     DC      Two
          END
```

8. Call return instruction

A call return instruction is an instruction that calls a subroutine.

(1) CALL instruction Instruction to call a subroutine. (Jump to subroutine)

Description method

label	CALL	Address [, x]

Processing is passed to the subroutine at the address.

(2) RET instruction Instruction to return processing to main processing.

Description method

label	RET	

Processing returns to the caller.

Program example (Data is read in main processing and sub processing)

```
CALL_RET  START
          LD      GR0, DATA1
```

	CALL	TEST; The processing moves to the subroutine of the TEST label.
	RET	; Process moves to OS. That means the end of the program
DATA1	DC	1
	END	
;		
TEST	START	
	LD	GR1, DATA2
	RET	; Process returns to CALL_RET side.
DATA2	DC	Two
	END	

9. Other instructions

Introduces SVC and NOP instructions that call OS functions.

(1) SVC instruction An instruction that calls the OS function. (Jump to subroutine defined by OS)

Description method

label	SVC	Address [, x]

Used to call OS functions.

* Note : The operation is determined by the CASL processing system (simulator, etc.). Check the specifications of the simulator used.

CASL2000 allows input, output and decimal output.
For details, refer to the help included with CASL2000.

(2) NOP instruction An instruction that does nothing.

Description method

label	NOP	

As the name implies, it is an instruction that does nothing.
Only the count up of the program register is performed.

10. Macro instruction

Predefined instructions combining machine language instructions are called macro instructions.
In CASL, input / output instructions do not exist as machine language. Defined as a macro instruction combining SVC instructions. There are some other macro instructions.

(1) IN instruction Input instruction

Description method

label	IN	Input data storage address , input character number storage address

Instruction to enter. In CASL2000, input from the keyboard.
Note that the input method differs depending on the simulator used.

Input characters are stored from the first address.
The number of characters entered is stored in the second address.

Note that if you enter a number, it will be treated as a number (character).
If you want to perform calculations such as addition on the "number" you have entered, you need to convert it to a number.

Example. Converts the entered single digit to a numeric value.

```
PROG_IN   START
          IN      DATA, SUU; Enter characters
          LD      GR0, DATA
          SUBA    GR0, HENKAN; Convert numbers to numbers
          ST      GR0, ANS
          RET
DATA      DS      1
SUU       DS      1
ANS       DS      1
HENKAN    DC      # 0030; Data for conversion
          END
```

(2) OUT instruction Output instruction.

Description method

label	OUT	Output data storage address, number of output characters

Instruction to output.

Outputs the data stored from the output data storage address for the number specified by the number of output characters.

Example. Outputs the input character string.

```
PROG_OUT  START
          IN      DATA, SUU; Input
          OUT     DATA, SUU; Output the input data as it is
          RET
DATA      DS      20
SUU       DS      1
          END
```

(3) RPUSH instruction An instruction to store the contents of GR on the stack.

Description method

label	RPUSH	

This instruction stores the contents of GR on the stack in the order of GR1, GR2, ..., GR7.

(4) RPOP instruction This instruction stores the contents of the stack in GR.

Description method

label	RPOP	

This instruction stores the contents of the stack in the order of GR7, GR6, ..., GR1.

Example. Temporarily save the contents of the register and restore it.

RPUSHPOP	START RPUSH RPOP RET END	

11. Assembler instructions

Assembler instructions are instructions for controlling the assembler. It is not converted directly to machine language.

(1) START command Command that indicates the start of a program

Description method

label	START	address

Indicates the start of a program.
This line must be labeled.

If an address is described in the operand, the program starts from that address.

(2) END instruction Instruction indicating the end of the program

Description method

	END	

Indicates the end of the program.

(3) DS instruction Instruction to secure area

Description method

label	DS	Number of words

Allocates a memory area for the number of words specified.

(4) DC instruction Instruction for defining constants

Description method

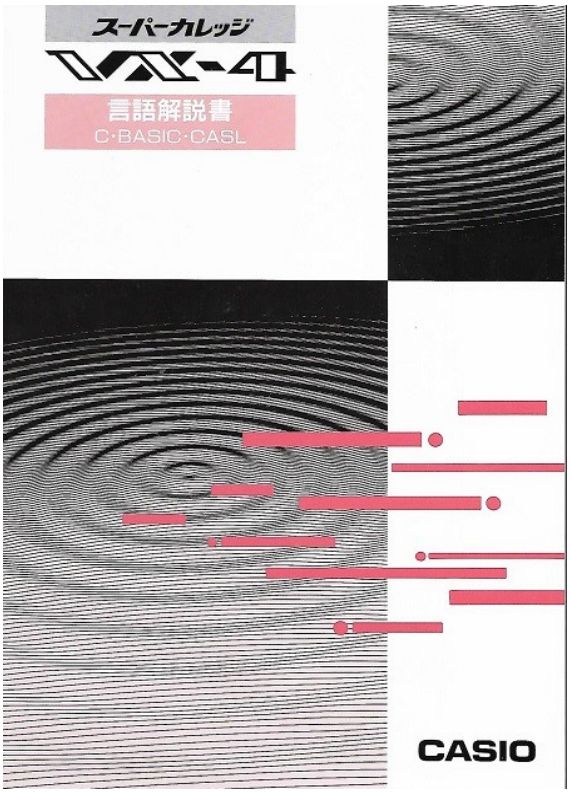
label	DC	Constant [, constant] ・・・

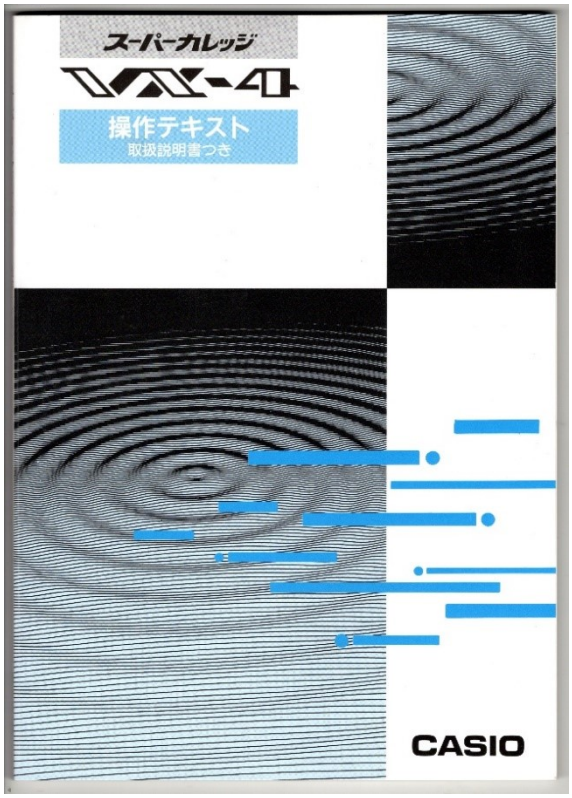
Define a constant.

The constant is

Decimal number: Number between -32768 and 32767
Hexadecimal: #hhhh 4-digit hexadecimal number starting with a sharp (0 to 9, A to F)
Character string: " Enclose in single quotation Address: Write the label

IX. Manuals


スーパーカレッジ
言語解説書
C·BASIC·CASL
CASIO


スーパーカレッジ
操作テキスト
取扱説明書つき
CASIO


スーパーカレッジ
問題集
CASIO


Introduction to the C programming language
PERSONAL COMPUTER
PB-2000C
CASIO

© 2020 created and translated from P.Rost