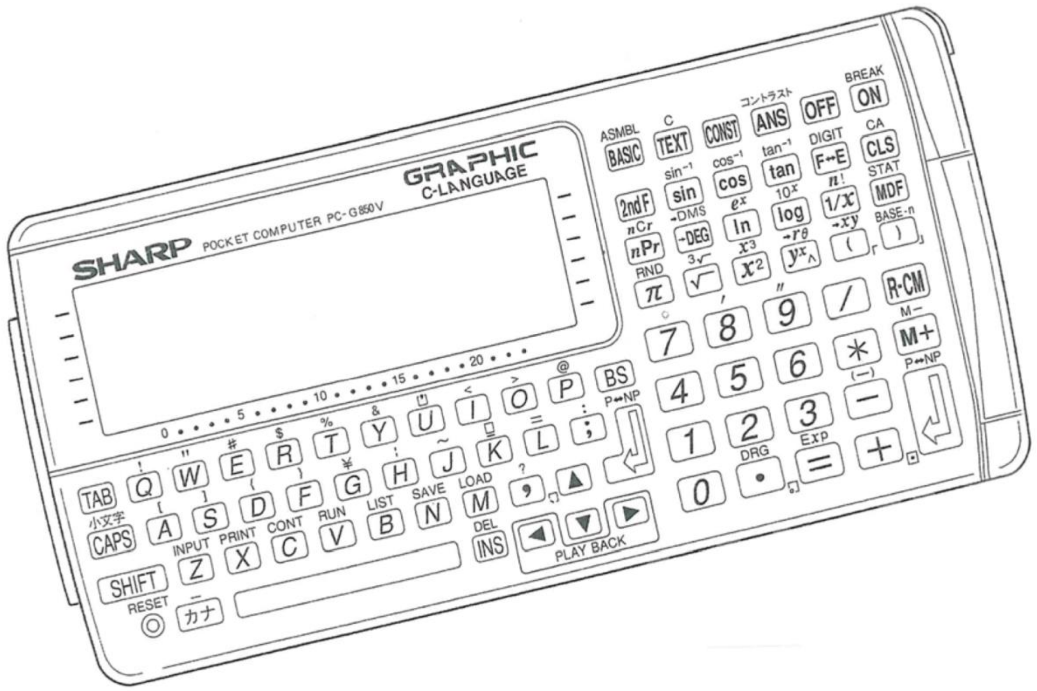
SHARP

POCKET COMPUTER

MODEL

PC-G850V(S)

USERS GUIDE



INTRODUCTION

The Pocket PC Sharp PC-G850V(S) is the latest model of a long line of pocket computers that originated in the late 1970s. At the same time, it stands out from the other models because of its special features. As there are no direct ancestors to the G850 series, the basic functionality of the PC-E200 / PC-E220 / PC-G815 was used as a foundation. Functions from the PC-1600 and PC-E500S were added. Additionally, some mathematical functions from the PC-14xx models were introduced.

A C compiler / interpreter has been integrated in the PC-G850V(S). This is most likely to be compared with the C-interpreter of the Casio Z-1GR. In addition, an integrated CASL assembler and COMET environment have been included.

This guide is made for both the Sharp G850V and Sharp G850VS. The difference between the two models is the slightly lower weight (10g) and the location of the operating system of the VS in flash memory. However, since there never was an update for the operating system, this property is irrelevant. This manual should also apply for the G850 and G850S, however, it has not been verified.

Unfortunately, Sharp's G models were only distributed within Japan, so they are hard to find in the rest of the world and no official non-Japanese documentation is available.

This manual was developed to address the lack of English documentation for this computer. It is based on the German translation of the official Japanese version of the Sharp PC-G850V(S) manual by Jörg Wrabetz. Additional information from the official Japanese version of the manual, the description of the 11-pin interface by Ton Stahl (Appendix A), and the Sharp PC-E500 manual was added for clarification. Thanks to hpmuseum.org forum members SMP, toml_12953, and rprosperi for catching errors and making suggestions for clarifying. Thanks to forplus for additional programming examples. This manual was NOT created by the SHARP CORPORATION and should not be considered official. Distribution of this manual is subject to the friendly (not yet available) permission from Sharp.

For errors in the text, in the technical descriptions, etc., as well as their consequences, no liability can be accepted.

Jack W. Hsu
jwhsu01@yahoo.com

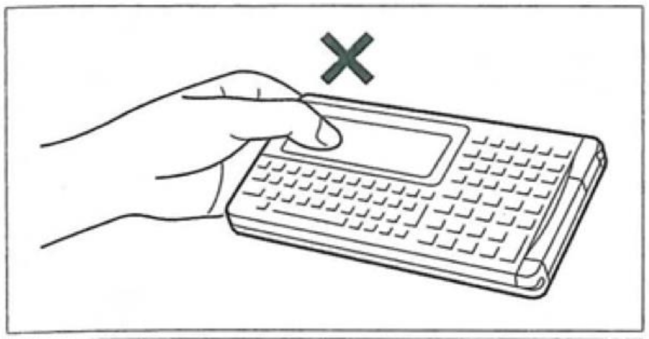
BASIC FEATURES OF THE SHARP PC-G850V(S):

1. *Built-in assembler:* The calculator is equipped with a built-in assembler which allows you to write programs in Z80 machine language.
2. *Programming in Basic:* The G850V(S) has a powerful basic language similar to the PC-1600 and has been enhanced with elements of the PC-E500S.
3. *Programming in C:* To do this, the computer has a built-in compiler to execute simple C programs.
4. *Programming in CASL:* CASL is an assembler language for a COMET virtual machine. This virtual machine and assembler system was created by the Japanese Ministry of Education to provide students and students with consistent training without the need for special hardware.
5. *Scientific calculations:* Simple and easy execution of scientific calculations.
6. *RAM disk:* A part of the internal memory can be used like a RAM disk to store programs and data.
7. *Serial interface:* This makes it possible to exchange programs and data between different pocket computers or even a PC.
8. Connecting programmable PIC microcontrollers.

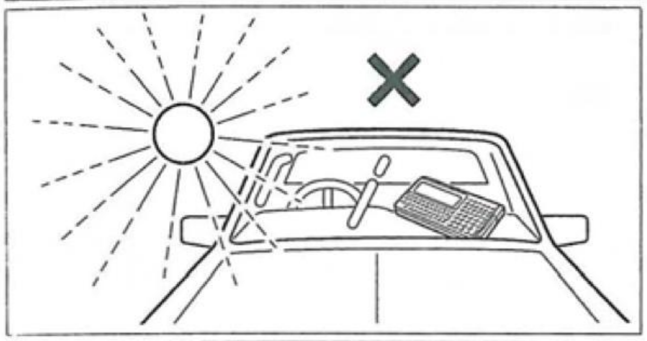
TABLE OF CONTENTS

1. OVERVIEW

1.1. Precautions



Please do not press the liquid crystal display. The display may break.



Please do not store near heaters or expose to direct sunlight (for example in a car). Due to the high temperatures, deformations can occur.

Do not drop, press or expose to any other force - The device may break.

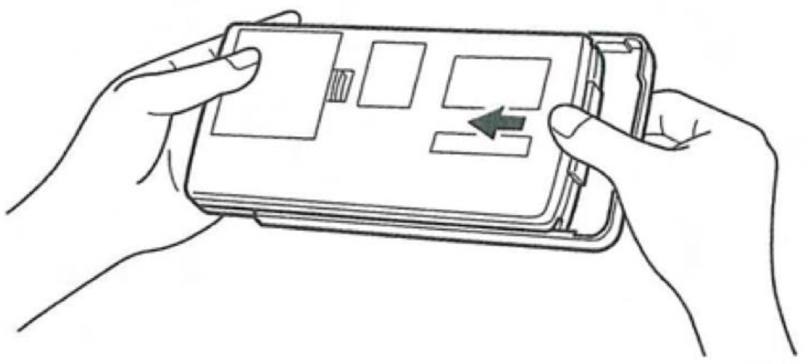
乾いたやわらかい布
揮発性

Clean the surface with a soft, dry cloth. Do not use solvents such as thinner, gasoline or a wet cloth. Color changes or surface damage may occur.

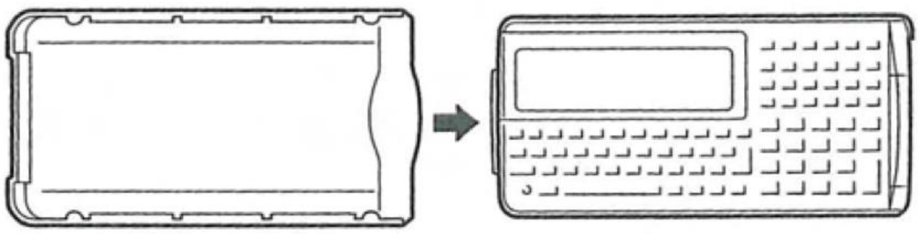
Please do not store with hard or sharp objects in your pocket. The device can be scratched. Always use the cover. The product is not waterproof.

The hardcover serves to protect the computer against damage. Whenever you are not using the pocket computer, please install the hardcover. For example, if you put the calculator in your pocket.

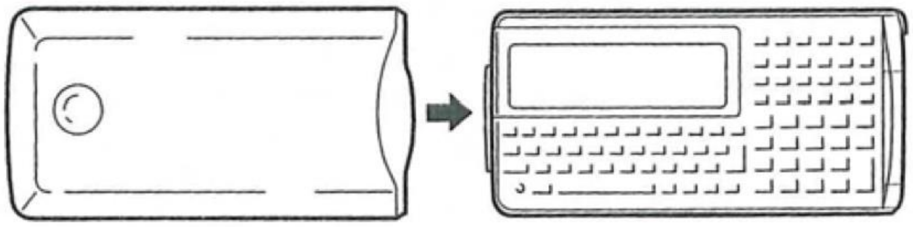
Removing the protective sleeve:



Using



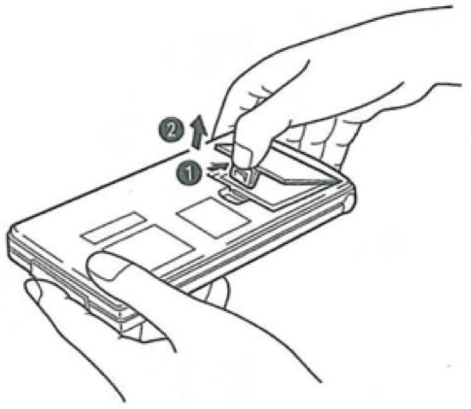
When not in use:



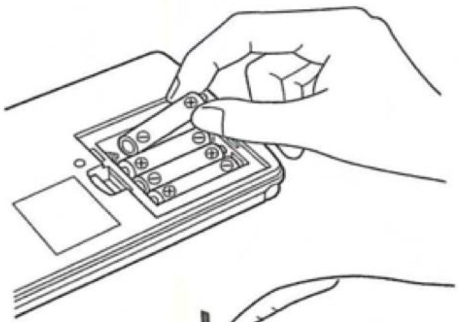
1.2. Using the PC-G850V(S) for the First Time

(1) Insert batteries

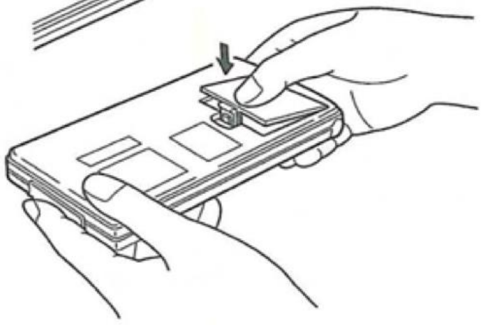
Please insert the batteries. To do so, remove the cover of the battery compartment on the back as shown in the illustration.



Insert the batteries in the correct direction. Follow the pictograms in the battery compartment.



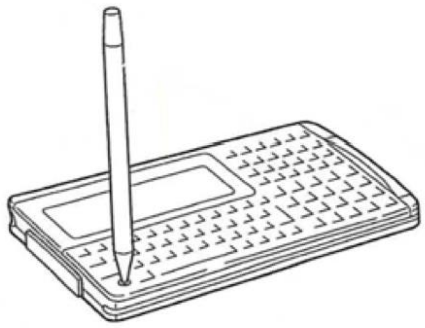
Close the battery lid again.



(2) Reset

Immediately after inserting the batteries into the computer, the internal status of the PC-G850V is not set yet. To do this, the computer must first be initialized.

- Press the **ON** button and then press the reset button under the **SHIFT** button with a ballpoint pen or similar device. Then release the reset button again.



Immediately after pressing the $\boxed{\text{RESET}}$ button, the PC-G850V displays the following screen. If any other indication appears, the above procedure must be repeated.

```

MEMORY CLEAR O.K.? (Y/N)

```

The PC-G850V asks for confirmation to clear the memory:

- Press the $\boxed{\text{Y}}$ key. The following message flashes, indicating that the computer has been initialized and all memory contents have been cleared.

```
************************
*                      *
*       ALL RESET      *
*                      *
************************
```

- Press any key. The following display appears:

```
RUN MODE
>
```

(3) Check computer function

To ensure normal computer function, press the following keys:

$\boxed{\text{F}}\ \boxed{\text{R}}\ \boxed{\text{E}}\ \boxed{↵}$

```
RUN MODE
FRE
                   30179
```

When the above screen appears, the computer is functioning normally and ready for input. The number 30179 represents the storage capacity for programs and data.

Note: If the PC-G850V(S) does not show the appropriate display after the above steps, the corresponding step should be tried again with the correct input for the step.

1.2.1. Replacing the Batteries

If **BATT** is displayed, the batteries must be replaced.

The computer uses four AAA batteries for operation. If the batteries are too weak while the CE-126P is being used simultaneously with the computer, it can also be powered by the CE-126P. This reduces the load on the internal battery.

1.2.2. When to Change the Batteries

If the **BATT** warning light appears in the lower left corner of the display, it means that the batteries are too low. They should be replaced with new ones immediately. If the computer continues to be used, even though **BATT** is displayed, the computer will turn off after some time. After that, it cannot be turned on by pressing the **ON** button again.

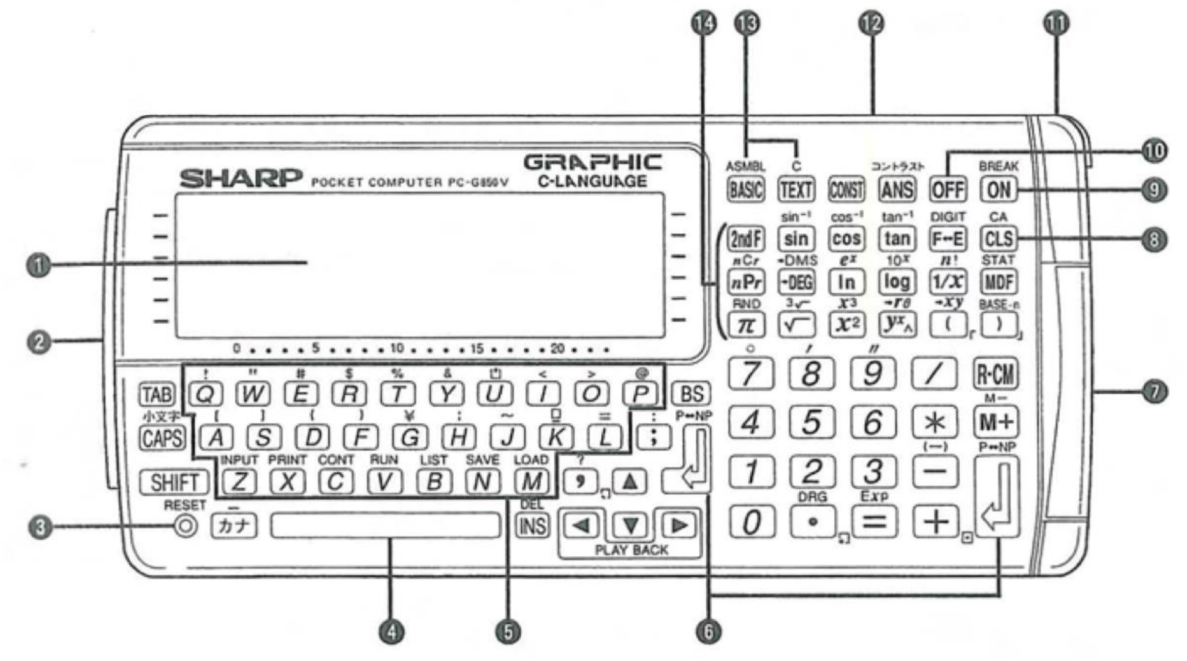
Note: The Pocket Computer retains its programs and files for a long period without batteries. To be on the safe side, do not remove the batteries from the computer for more than 5 minutes.

Caution: NEVER remove the batteries when the pocket computer is switched on, because after reinserting the batteries the computer must always be reset and thus all data is lost. Also, you may want to backup or print all programs and data to a PC first.

If an additional peripheral device is connected, the computer can be powered by this device. In this case, the **BATT** warning indicator does not appear even though the batteries of the computer are too weak. Before use, the peripheral should be disconnected at short notice to check if the **BATT** warning light appears on the display or not. Furthermore, there is a connection on the rear right side to power the computer with an external power supply (6V, 0.2W).

1.3. Device Overview

The SHARP computer consists of a QWERTY keyboard similar to that of a conventional typewriter and an LCD display with adjustable contrast. On the left side is the SHARP 11-pin interface and on the right side the interface to the PIC microcontroller. Top right is a connector for an external power supply with 6V and 0.2W (e.g. power supply Sharp EA-23E).



1 Display (6 lines, 24 characters / line) 144x48 pixels)
2 SHARP 11-pin interface for printer, serial interface, etc.
3 Reset button (recessed)
4 Space bar
5 Typewriter keyboard
6 Enter key(s)
7 Interface for PIC microcontroller
8 Delete key
9 Power-on/Wake-up button
10 Power-off button
11 Connection for power supply (6V, 0.2W, e.g. power supply Sharp EA-23E).
12 Battery compartment cover (on the back)
13 Mode toggle keys (Basic RUN / PRO, Assembler, C, CASL, Text Editor)
14 Function key

2. BASIC FUNCTIONS AND MODES

There are a number of important letters, numbers and symbols on the PC-G850V.

2.1. Switching on the Computer

Press the **ON** button on the right side of the computer keyboard. The computer is in **RUN** mode after being turned on.

2.2. Automatic Shutdown

To protect the batteries, the computer automatically shuts itself off if no buttons are pressed after approximately 11 minutes. Press the **ON** button to turn it back on after the computer turned itself off.

If the computer is executing an `INKEY$` command, the auto power off function is disabled. It is active while the computer is executing an `INPUT` command.

If the computer is not used for a long time while the automatic shutdown is disabled, battery power will be consumed. This can lead to the loss of stored programs or data.

2.3. Setting the Contrast

The menu for setting the contrast is called by pressing **SHIFT** and **ANS**. Adjust the contrast so that you can see the display clearly.

```
*** LCD CONTRAST ***

         ▲    DARK
         ▼    LIGHT
```

Pressing the cursor keys ▲ (Increase Contrast) and ▼ (Decrease Contrast) adjusts the contrast.

If the display is set correctly, the setting can be set by pressing the **BASIC**, **TEXT** or **CLS** key.

2.4. The SHARP PC-G850 Modes

The Sharp PC-G850V has 7 different modes:

RUN mode	execute BASIC programs or BASIC commands, input of mathematical functions
PRO mode	writing or correcting BASIC programs
TEXT mode	entering, editing, deleting and saving (ram disk, SIO), loading (ram disk, SIO) text programs in ASCII format, conversion to BASIC or vice versa, creating and deleting data files
ASMBL mode (assembler mode)	assemble an assembler program (generation of Z80 machine code)
CASL mode	translate and execute CASL programs (accessible via **ASMBL**)
PIC mode	translate source programs and transmit them to the PIC. (accessible via **ASMBL**)
C mode	compile and run C programs.

Mode Switching

Mode	Keys
RUN mode	BASIC
PRO mode	BASIC or BASIC BASIC (to enter **PRO** mode from outside the **RUN** mode, press BASIC twice)
TEXT mode	TEXT
ASMBL mode	SHIFT + BASIC (**ASMBL**), then A
CASL mode	SHIFT + BASIC (**ASMBL**), then C
PIC mode	SHIFT + BASIC (**ASMBL**), then P
C mode	SHIFT + TEXT (**C**)

Instead of SHIFT you can press 2ndF beforehand.

2.5. Basic Operation

Turn on the computer. Pressing **CLS** will clear the screen and start typing in the upper left corner.

Character input

A **B** **C** **D** **E** **F** **G**

```
ABCDEFG
```

In this way you will get capital letters.

Lowercase input

H **I** **J** **K** **L** **M** **N**

```
ABCDEFGhijklmn
```

Pressing **CAPS** will quit the **CAPS** mode (always on after power-on) and will allow entering of lowercase letters.

Special characters

By pressing the **SHIFT** key and the associated key simultaneously, the corresponding special character is written. Alternatively, press the **2ndF** key BEFORE pressing the corresponding key. The following keypress will enter the corresponding special character or mathematical function.

E **R** **T**

```
ABCDEFGhijklmn#$%
```

2ndF **I** **2ndF** **O** **2ndF** **P**

```
ABCDEFGhijklmn#$%<>@
```

This is how numbers are entered

1 **2** **3** **4** **5** **6**
7 **8** **9**

```
ABCDEFGhijklmn#$%<>@1234
56789
```

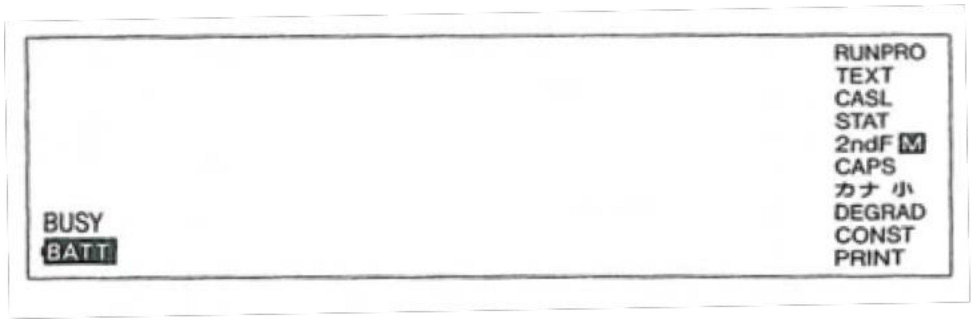
Cursor control

To change entered characters, use the four cursor keys (◀▶▲▼).

If the cursor is at the end of a line, an underscore appears to allow continuation of the line. If the cursor is in the middle of the text field, the corresponding character flashes in black. The entry of characters overwrites existing characters from this position. When you hold down the cursor key, the cursor moves quickly over the screen.

2.6. The Display

The computer has a 6-line liquid crystal display with 24 characters per line and a status bar at the top and bottom. Each character occupies a 5x7 dot matrix. The display shows key names and calculations. The display examples in this manual only reproduce the symbols required for the respective explanation of the function.



In BASIC mode, the display shows:

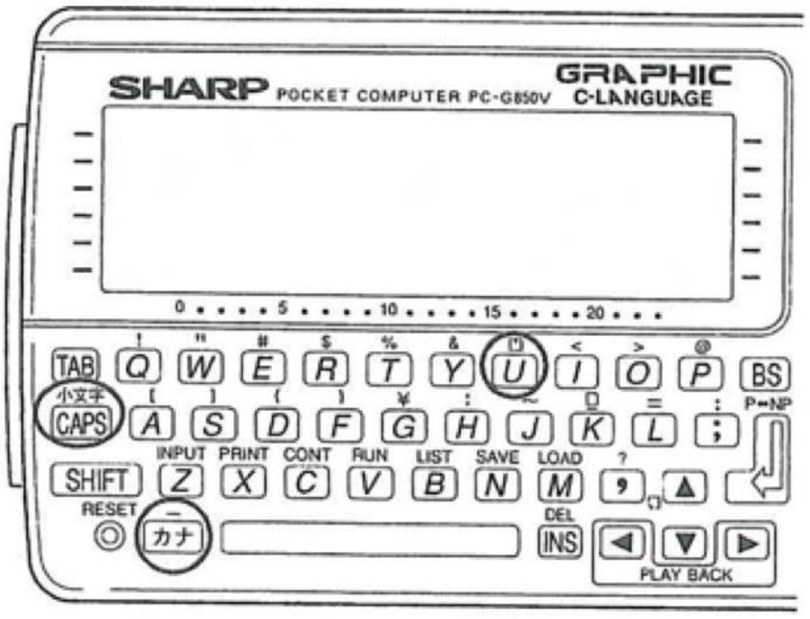
>	Standby symbol. This icon appears when the computer is ready to accept an input in **BASIC** mode. When typing. As you type, the ready icon disappears and is replaced by the cursor
_ ▩	The cursor. This symbol marks the location of the next character to be entered. When you start typing, the cursor replaces the standby icon. As a marker symbol, the cursor is also used in conjunction with the INSert and DELete functions. The underscore cursor changes to the block cursor when it is not on a character.

The status lines reflect the following:

BUSY	This word appears on the display when the computer executes a program or command.
BATT	This symbol indicates that the batteries are weak and need to be replaced.
RUN	This icon indicates the **RUN** mode for the computer.
PRO	This symbol indicates the BASIC programming mode for the computer.
TEXT	Indicates that the computer is in **TEXT** mode.
CASL	This icon indicates the **CASL** programming mode for the computer. Enter this mode by pressing the ASMBL key (SHIFT + BASIC followed by C).
STAT	This icon indicates that the computer is in statistics mode. Enter this mode by pressing the STAT key (SHIFT + MDF)
2ndF	The display appears when the 2ndF button has been pressed and disappears with the following key command. Remember that the 2ndF key must be released before pressing another key when the second

M	Indicates that another number other than zero is stored for manual calculations.
CAPS	Indicates that the computer is in **CAPS** mode. If this indicator does not appear on the display, all letters of the alphabet are entered as lowercase letters. The CAPS key can be used to toggle **CAPS** mode on and off again.
カナ	If you press the カナ key, you can enter katakana syllables with Latin letters. (See page 11) By pressing this key, you can toggle this feature on and off.
小	Indicates that the computer is in katakana mode and you can enter lowercase letters by turning off the **CAPS** mode.
DEG **RAD** **GRAD**	Shows the current angle mode of the computer: **DEG** (Degree mode) **RAD** (Radian mode) **GRAD** (Gradian mode)
CONST	Indicates that the computer has a constant set for calculations. When this icon is displayed, the computer performs a calculation with this constant each time the button is pressed. If the constant is no longer needed, it can be deleted with **SHIFT** + CA.
PRINT	This icon indicates that the computer is ready in **RUN** mode to send data to the printer. Press **SHIFT** + P↔NP to toggle on and off. (Only possible with an optional printer connected.)

The PC-G850V offers you the possibility of Japanese words in Kanji format entry. This function is switched off by pressing the カナ button.


SHARP POCKET COMPUTER PC-G850V
GRAPHIC C-LANGUAGE
0 5 10 15 20
TAB Q W E R T Y U I O P BS
小文字 CAPS A S D F G H J K L ;
P↔NP
INPUT PRINT CONT RUN LIST SAVE LOAD
SHIFT Z X C V B N M
RESET
カナ
DEL INS
PLAY BACK

カナ	Toggle kanji mode
小文字 CAPS	Switch to lowercase letters
SHIFT + U	Switch to entry of consonants

Examples:

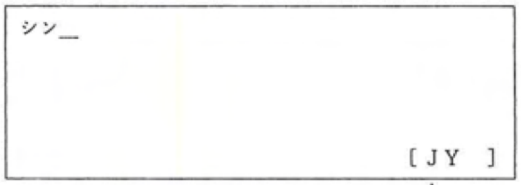
Input Character		Keyboard Input
カタカナ	→	KATAKANA
ガッコウ	→	GAKKOU
ヘンカン	→	HENNKAN $\boxed{\text{SHIFT}}$ + $\boxed{\text{U}}$
ハンイ	→	DHISUKU
ディスク	→	HAN $\boxed{\text{SHIFT}}$ + $\boxed{\text{U}}$ I
ウォッチ	→	I $\overset{\text{小文字}}{\boxed{\text{CAPS}}}$ OTTI

When entering 'n', if consonants excluding Y come after Shift+U (there is no need to press SHIFT + U.

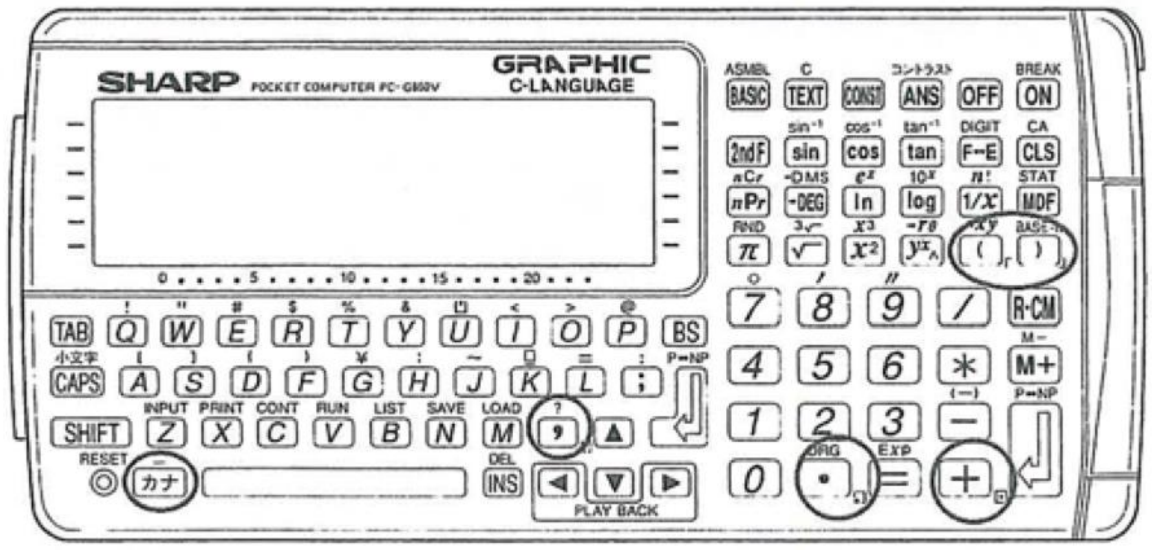
For details on how to write Romani, see page 273.

Caution: It is not always possible to display all characters of the entry in the bottom line become:

S I N J Y



Special kanji symbols:



ー（長音符）	:	$\boxed{\text{SHIFT}}$ + $\boxed{\text{カナ}}$
、（読点）	:	$\boxed{,}$ 、
。（句点）	:	$\boxed{\cdot}$ 。
・（中点）	:	$\boxed{+}$ ・
「（カギカッコ）	:	$\boxed{(}$ 「
」（カギカッコ）	:	$\boxed{)}$ 」

3. MANUAL CALCULATIONS

The computer can be used as a 10-digit calculator. To do this, the computer must be set to **BASIC RUN** mode. The **RUN** indicator will appear on the top right of the display.

	Math Function	Math Operator	Input
(1)	Addition	+	+
(2)	Subtraction	-	-
(3)	Multiplication	×	*
(4)	Division	÷	/
(5)	Integer Division		¥
(6)	Modulo division		MOD
(7)	Sign	+ or -	+, -
(8)	Perform operation	=	↵

Examples:

51 ¥ 5 ↵	→ 10	51 MOD 5 ↵	→ 1	(51 ÷ 5 = 10… 1)
51 ¥ –5.7 ↵	→ –8	51 MOD –5.7 ↵	→ 3	(51 ÷ –6 = –8… 3)
87.57 ¥ 5.4 ↵	→ 17	87.57 MOD 5.4 ↵	→ 3	(88 ÷ 5 = 17… 3)
30º36´ ¥ 14º36´ ↵	→ 2	30º36´ MOD 14º36´ ↵	→ 1	(31 ÷ 15 = 2… 1)
30º36´ ¥ 4.4 ↵	→ 7	30º36´ MOD 4.4 ↵	→ 3	(31 ÷ 4 = 7… 3)
87.57 ¥ 14º36´ ↵	→ 5	87.57 MOD 14º36´ ↵	→ 13	(88 ÷ 15 = 5… 13)
5 + 15 * 2 / 4 ↵	→ 12.5			

3.1. Keypad Operation

The cursor key, ◄, can be used to edit and change the input. To edit, move the cursor to the appropriate position using the cursor keys and overwrite the character with a new one. If characters are to be inserted, press the **INS** key for insert mode. Insert mode remains active until the **INS** key is pressed again. If the character under the cursor is to be deleted, use **DEL** (**SHIFT** + **INS**). If, on the other hand, the character in front of the cursor is to be deleted, use the **BS** key.

3.1.1. Keys for Mathematical Operations

ANS

If **ANS** is used when entering a mathematical function, the result of the last calculation is inserted at the current cursor position.

Example:

5 + 15 * 2 / 4 ↵ → 12.5
4 * **ANS** (inserts 12.5) ↵ → 50

EXP, 10^x, and e^x

Entering the exponent is performed by SHIFT + EXP. Instead of EXP, the letter E can also be used.

Example:

36 EXP 3 → 36E3 ↵ → 36000
52 10^x ↵ → 1.E 52
5 e^x ↵ → 148.4131591

DIGIT

The DIGIT key, in conjunction with a number key, is used to specify the number of decimal places. If a dot (.) is used instead of the number key, the display is reset to the default number of decimal places.

Use the F↔E key to switch between fixed and scientific mode.

Example:

2ndF DIGIT 2 ↵
5 / 8 ↵ → 0.63
2ndF DIGIT 5 ↵ → 0.62500
2ndF DIGIT . ↵↵ → 0.625

USING

Format: `USING [`*`format-string`*`]`

`USING` allows formatted data output. The format is determined by a *`format-string`*, which consists of a series of characters enclosed in quotation marks. The *`format-string`* is composed of the following characters:

#	right-justified character of a numeric field
.	delimiter between the integer and fractional part of a number
,	comma as separator after 3 digits in numeric fields
^	display the number in scientific notation

For each # contained in *`format-string`*, a digit of a numerical value or the sign can be displayed. All other format symbols are used to describe numeric formats in more detail. Both positive and negative values can be represented, however, the sign is only displayed for negative values.

The format for `USING` corresponds to the **BASIC** command `USING` (see page 234). `USING` without parameters resets the output format to default.

Example:

`USING` "###. ##" ↵
8 / 3 ↵ → 2.66
17 / 3 ↵ → 5.66

If the result is greater than that permitted by the output format, an error (`ERROR 70`) is displayed.

```
USING "###. ##" ↵
17865 / 3 ↵        → ERROR 70
```

MDF (Modification Function)

With **DIGIT**, the computer displays only the specified number of decimal places, but internally it always stores all digits. Therefore, the displayed data may differ from the internal data. To match the internal and the displayed data, the modification function is used.

Example:

Without MDF

Keystrokes	Output
(**2ndF** **CA**) ↵	
55.4 **/** 9 ↵	6.155555556
***** 9	6.155555556*9_
↵	55.4

with MDF

Keystrokes	Output
(**2ndF** **DIGIT** 3) ↵	
55.4 **/** 9 ↵	6.156
MDF	6.156
***** 9	6.156*9_
↵	55.404

Sign Change

The sign (–) (**2ndF** **–**) reverses the sign of the displayed result (from plus to minus and vice versa).

3.2. Memory Operations

Independent memory can be selected with the keys **M+**, **M-** and **R-CM**.

R-CM	Shows the contents of the memory and inserts it into the calculation. If **R-CM** is pressed twice, the contents of the memory will be erased. The **M** symbol clears.
M+	Adds the displayed result or result of a mathematical function to the memory. If the memory was empty before (i.e. 0) the **M** symbol appears
M-	Subtracts the displayed result or result of a mathematical function from the memory. If the memory was empty before (i.e. 0) the **M** symbol appears

3.3. Calculations with Constants

The CONST key can be used to apply constants to basic arithmetic operations as described below.

Using constants

Addition:	+	a	CONST	or	a	+	CONST
Subtraction:	–	a	CONST	or	a	–	CONST
Multiplication:	*	a	CONST	or	a	*	CONST
Division:	/	a	CONST	or	a	/	CONST

where "a" means the constant. After pressing CONST, **CONST** appears on the bottom right of the display.

> **Note:** If the constant function is not used, make sure that the **CONST** indicator does not appear on the display.

Viewing constants

To view the last entered constant, press 2ndF + CONST (SHIFT + CONST) while **CONST** is displayed.

Delete the last constant

To delete the last entered constant, press 2ndF + CA (SHIFT + CA). It can also be deleted by switching off the device. The **CONST** indicator should be off.

Example:

Store "+ (4.8 + 3.6)" as a constant and calculate "24 –18.5 + (4.8 + 3.6)" and "8.2 x 6 + (4.8 + 3.6)"

Enter:	+ 4.8 + 3.6 CONST	(Parenthesis not required)
Enter:	24 – 18.5 ↵	Result: 13.9
Enter:	8.2 * 6 ↵	Result: 57.6

3.4. Priority in Direct Input Calculations

The PC-G850V(S) uses the following operator precedence when evaluating an expression:

1. Variable recall or π
2. Functions (such as sin, cos, etc.)
3. Power, roots
4. Sign
5. *, / (operators at the same priority level are executed sequentially from left to right)
6. +, –
7. Relational expressions
8. Logical expressions
9. =, M+

If parentheses are used in a formula, the operation given within the parentheses has the highest priority.

Note:

- Composite functions are evaluated from right to left
- Chained power (3^{4^2} or 3^4^2) is evaluated from right to left
- For items 3 and 4 above, the last entry has the highest priority

  -2^4 $\rightarrow$ -(2^4)

  3^-2 $\rightarrow$ 3^{-2}

3.5. Base Conversion (BASE-n)

In the ROM of the PC-G850V(S) is a BASIC program which allows for numerical conversion among base2, base10, and base16 systems. It also computes the 2 compliment allows basic arithmetic and logical operations. To access this program, press **SHIFT** + **BASE-n**. You will then receive the following prompt asking to clear the BASIC program memory space (effectively deleting all your basic programs).

```
BASIC DELETE OK? (Y)
```

Pressing **Y** will load the `BASE-n` program into memory and execute the program. Pressing any other key will return you to **RUN** mode and leave any BASIC programs in memory intact.

Note: The `BASE-n` program requires 2794 bytes of free memory plus an additional 167 bytes for variable storage. If not enough memory is available to run the program, the process is aborted and BASIC program memory is left intact. This is true even if you see the above prompt to clear memory and press **Y**. The computer will return to **RUN** mode.

Attempting to run the program with **SHIFT** + **BASE-n** with insufficient memory will return you to **RUN** mode.

If variables cannot be allocated at runtime, you will receive an `ERROR 60` message. In this case, clear the variable memory or delete a `TEXT` program to free up more memory space. Please make sure that at least 2961 (2794 + 167) bytes are available prior to starting `BASE-n`.

To exit the program, press **BREAK**. To restart the program, you can `RUN` in **RUN** mode provided no alterations were made in the BASIC program memory space. Using **SHIFT** + **BASE-n** to restart the program will reload `BASE-n` from ROM and execute the program.

Upon starting `BASE-n`, you will be presented with the following screen:

```
 ***** n ツン エンサ’ ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ         4:ケイサン

(1,2,3,4)?
```

The four available options are:

1) Input: enter numeric value for conversion.
2) Convert: convert numbers from one base to another. It also toggles between input and calculation.
3) Complement: finds the two's complement of the number
4) Calculation: allows calculation with basic arithmetic operators (+, –, *, /) and logical operators (AND, OR, NOT, XOR).

3.5.1. Value Range

The program is limited to the following range limits

Binary	: 16 bits. The most significant (leftmost) bit is the sign bit. If more that 16 bits are entered, the first 16 bits are used as the input value. If the result of a calculation exceeds 16 bits, the results is truncated to the least significant 16 bits.
Hexadecimal	: 0000-FFFF (8000–7FFF) If more than 4 hexadecimal numbers are entered, the value is truncated to the 4 least significant 4 hexadecimal digits.
Decimal	: -32768 – 32767 Decimal values are converted to the corresponding hexadecimal number then truncated to the least significant 4 hexadecimal digits if it has more than 4 digits. This means that any decimal value entered will always fall within the above decimal range.

3.5.2. Input Number

Pressing $\boxed{1}$ allows entry of a number for conversion or calculation. The program prompts for numeric entry. For example, to enter the decimal number 1230 for conversion:

$\boxed{1}$

```
 ***** n ツン エンサ゛ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

(1,2,3,4)?
```

$\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{0}$ $\boxed{↵}$

```
 ***** n ツン エンサ゛ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[10ツン]=                1230
```

Note that the program is expecting a decimal number. This is indicated by the value in brackets. If an incorrect digit is entered (i.e. a hexadecimal digit is entered when the program is expecting a binary number), `ERROR` is temporarily displayed and the program waits for another input.

3.5.3. Base Conversion

After a number is entered, pressing $\boxed{2}$ allows conversion of the number from one base to another. The sequence of the conversion is:

Decimal $\rightarrow$ Hexadecimal $\rightarrow$ Binary - …

For example, given the decimal number 1230:

$\boxed{2}$

```
 ***** n ツン エンサ゛ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[16ツン]=                04CE
```

$\boxed{2}$

```
 ***** n ツン エンサ゛ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[ 2ツン]= 0000010011001110
```

2

```
***** n ツン エンサ' ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[10ツン]=                1230
```

The converted number is available for further calculations.

3.5.4. Two's Complement

The two's complement is the inverse of a binary number. The sum of a number and its two's complement is equivalent to the sum of the number and its inverse, which is zero. You can calculate the two's complement of an entered number by pressing 3. The operation works regardless of the base of the number. For example, to calculate the two's complement of 12C7:

2 (to switch to hexadecimal numbers)

1

12C7↵

```
***** n ツン エンサ' ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[16ツン]=                12C7
```

3

```
***** n ツン エンサ' ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[16ツン]=                ED39
```

Press 3 again and you have 12C7. The two numbers are complementary to each other.

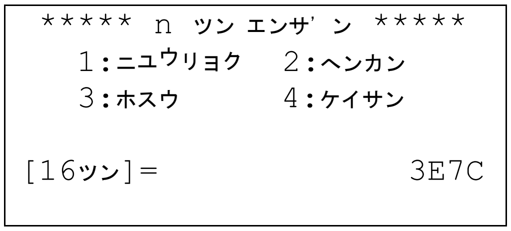
3.5.5. Calcuations

This program allows calculations directly on hexadecimal and binary numbers. Normally in **RUN** mode, hexadecimal/binary values must be converted to decimal to allow mathematical or logical operations. In this program, the operations can be performed on the values directly, without the need for conversion. Pressing 4 brings up a list of the available operations:

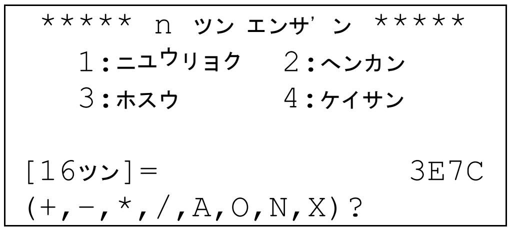
```
***** n ツン エンサ' ン *****
   1:ニユウリヨク   2:ヘンカン
   3:ホスウ        4:ケイサン

[16ツン]=                ED39
(+,-,*,/,A,O,N,X)?
```

In addition to the standard arithmetic operations (+, –, *, and /), the logical operators (A)ND, (O)R, (N)OT, and (X)OR are also available. For example, to calculate the result of 3E7C AND 0FF0:

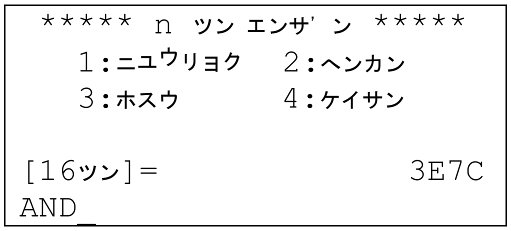
$\boxed{1}$
3E7C ↵


```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[16ﾂﾝ]=              3E7C
```

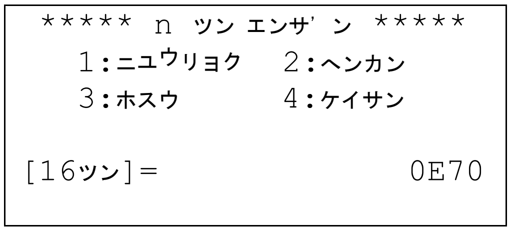
$\boxed{4}$


```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[16ﾂﾝ]=              3E7C
(+,-,*,/,A,O,N,X)?
```

$\boxed{A}$

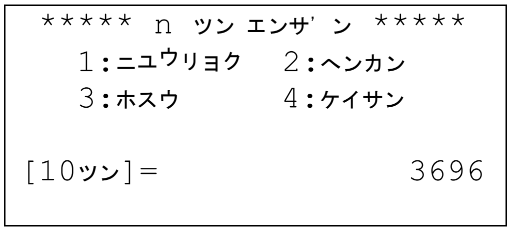

```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[16ﾂﾝ]=              3E7C
AND_
```

0FF0 ↵

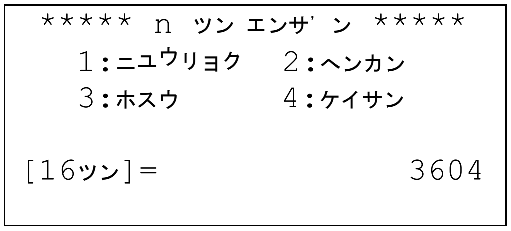

```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[16ﾂﾝ]=              0E70
```

You can perform further calculations on the result. For instance, to find the result of 92 (decimal) subtracted from the result of the prior calculation:

$\boxed{2}$ $\boxed{2}$ (switch to decimal)


```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[10ﾂﾝ]=              3696
```

$\boxed{4}$ $\boxed{–}$ 92 ↵


```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆﾕｳﾘﾖｸ   2:ﾍﾝｶﾝ
   3:ﾎｽｳ      4:ｹｲｻﾝ

[16ﾂﾝ]=              3604
```

2 (switch to hexadecimal)

```
 ***** n ﾂﾝ ｴﾝｻﾞﾝ *****
   1:ﾆｭｳﾘｮｸ    2:ﾍﾝｶﾝ
   3:ﾎｽｳ       4:ｹｲｻﾝ

[16ﾂﾝ]=                 0E14
```

Note: The the program executed by **SHIFT** + **BASE-n** is named `BASE_N.BAS`. If there is a BASIC program of the same name in the File area, the `BASE_N.BAS` program in ROM will not be executed. Therefore, to use the ROM program, the program in the File area with the same name must be either renamed or removed. Otherwise, pressing **SHIFT** + **BASE-n** will execute the program with the same name in the File area.

Additionally, pressing **SHIFT** + **BASE-n** automatically executes a `GOTO 100`. This may result in an error in the execution of the program with the same name in the File area.

4. STATISTICS MODE

The PC-G850V(S) can perform statistical and regression calculations on one or two variables. With statistical calculations, you can obtain mean values, standard deviations and other statistical quantities from sample data. Regression calculation determines the coefficients of linear regression formulas or estimate values from sample data. Sample data is stored in fixed registers U-Z.

4.1. Entering STAT Mode

You can enter **STAT** mode by pressing the **STAT** key (**SHIFT** or **2ndF** + **MDR**). The **STAT** indicator will be displayed on the right side of the screen. The following screen will then appear:

```
***** トウケイ ブ’ンセ *****

  1: 1~ンスワ トウケイ (x)
  2: 2~ンスワ トウケイ (x,y)

バ’ンコ’ウ ヲ エランデ’クダ’サイ.
```

From this screen, you can then select between one-variable or two-variable statistics. To exit from the main **STAT** screen, press **BREAK**.

Caution: When you enter **STAT** mode, all previous statistical data and calculations are erased.

4.2. One-Variable (Univariate) Statistics

To calculate one-variable statistics, press **1** from the main **STAT** screen. You will then see the following screen:

```
*** ショリ ***        (x)

1:ニュウリョク     2:サクソ’ョ/クリア
3:ブ’ンセキ       4:ブ’リンタ

バ’ンコ’ウ ヲ エランデ’クダ’サイ.
```

The menu options are:

1: Enter data
2: Delete/Clear Data
3: Calculate statistics
4: Print statistics

To return to the main **STAT** screen, press **BREAK**.

4.2.1. Data Entry

From the univariate statistics menu, press **1** to enter data for statistical analysis. You will be presented with the following screen:

```
** ﾃﾞｰﾀ ﾆｭｳﾘｮｸ **
1:x=_
```

The number represents the total number of items entered for analysis while the underscore tells you that the computer is waiting for an input. To enter data for analysis, type the value and press ↵ to accept.

To enter a negative value, press the sign (**2ndF** **–**) key then the value. Then press ↵ to accept.

If you need to enter several data points that have the same value, you can enter the value followed by a comma (**,**), then the number of times the value is repeated. Press ↵ to accept.

Press **BREAK** to accept the entered data and return to the univariate statistics menu.

Example: Enter the values from the following table for analysis:

Value	# Entries	Value	# Entries
30	1	70	8
40	1	80	9
50	4	90	5
60	5	100	2

Keystrokes	Output
SHIFT + **STAT** (Enter **STAT** mode)	
1 (Select 1-variable statistics)	
1 (Select Data Entry mode)	1:x=_
3 **0** ↵	2:x=_
4 **0** ↵	3:x=_
5 **0** **,** **4** ↵	7:x=_
6 **0** **,** **5** ↵	12:x=_
7 **0** **,** **8** ↵	20:x=_
8 **0** **,** **9** ↵	29:x=_
9 **0** **,** **5** ↵	34:x=_
1 **0** **0** **,** **2** ↵	36:x=_
BREAK (Exit data entry)	

4.2.2. Univariate Statistical Calculations

Press $\boxed{3}$ in the univariate statistics menu to access the available univariate statistical calculations. The following screen is displayed:

```
** フ゛ンセキ **      (x)

 1:n    2:∑x   3:∑x²  4:x̄
 5:s    6:σ

ハ゛ンコ゛ウ ヲ エランテ゛クタ゛サイ.
```

The following calculations are available:

n	:Sample size of x
$\bar{x}$	:Sample mean of x
∑x	:Sum of samples x
∑x^2	:Sum of squares of samples x
s	:Sample standard deviation. The formula for the sample standard deviation is:

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n-1}}.$$

σ :Population standard deviation. The formula for the population standard deviation is:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \bar{x})^2}{n}}.$$

Press the appropriate number to obtain the corresponding result.

Example: *Continuing from the prior example.*

Keystrokes	Output
$\boxed{3}$ (Select univariate calculations)	
$\boxed{4}$ ↵	x̄= 71.42857143
$\boxed{5}$ ↵	s= 16.47508942
$\boxed{6}$ ↵	σ= 16.23802542
SHIFT + $\boxed{\textbf{STAT}}$ (Exit **STAT** mode)	

4.2.3. Correcting Errors

If an error is made during data entry, return to the univariate statistics menu by pressing $\boxed{\textbf{BREAK}}$ and press $\boxed{2}$ to enter Delete/Clear mode. The following screen is displayed:

```
** サクシ゛ョ/クリア **


 1: テ゛ータ サクシ゛ョ
 2: オールレ クリア

ハ゛ンコ゛ウ ヲ エランテ゛クタ゛サイ.
```

Selecting $\boxed{1}$ will allow you delete incorrect entries in the dataset. The following screen is displayed:

```
*  ﾃﾞｰﾀｻｸｼﾞｮ  *
x=_
```

Use the same procedure as in Data Entry (section 4.2.1) to delete erroneous values. You can use $\boxed{,}$ to delete multiple identical data values. When complete, press $\boxed{\textbf{BREAK}}$ to accept changes and return to univariate statistics menu. You can now return to Data Entry (Option 1) to enter the correct values.

Selecting $\boxed{2}$ will delete the entire dataset allowing you to start over with data entry. When pressed, the following screen is displayed:

```
* ｵｰﾙ ｸﾘｱ *

 1:YES
 2:NO

ﾊﾞﾝｺﾞｳ ｦ ｴﾗﾝﾃﾞ ｸﾀﾞｻｲ.
```

If `YES` is selected, the statistical registers are cleared and the univareiate statistics menu is displayed. If `NO` is selected, the statistical registers are untouched and the univariate statistics menu is displayed.

4.2.4. Printing

After data entry is complete, the calculated statistics can be printed with an optional CE-126P printer. Connect the printer to the computer and turn on the power to the printer. After data entry is complete, press $\boxed{4}$ to print the results of the statistical calculations. You will see the following on the screen:

```
**  ｲﾝｻﾂﾁｭｳ **
```

The printer will print the results of all the statistical calculations. A sample printout is shown below:

```
n=                        35
∑x=                     2500
∑x²=                  187800
MEAN(x̄)=         71.42857143
s=               16.47508942
σ=               16.23802542
```

4.3. Two-Variable (Bivariate) Statistics

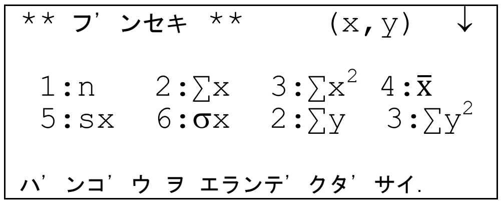
The basic operation of two-variable (bivariate) statistics is identical to univariate statistics. Enter **STAT** mode and press $\boxed{2}$ to enter the bivariate statistics menu. Press $\boxed{1}$ to enter data. Data entry is identical to univariate statistics mode except you are now prompted to enter two values, x and y. As in univariate data entry, the sign key ($\boxed{\text{2ndF}}$ $\boxed{-}$) is used to enter negative values and comma ($\boxed{,}$) is used to enter duplicate value pairs. When data entry is complete, press $\boxed{\text{BREAK}}$ to accept values and return to the bivariate statistics menu.

When performing exponential, logarithmic, power, or inverse regression calculations, use the following table to transform the respective regression calculation formula into a linear regression. Each statistic can then be determined using the transformed values for x and y.

Type	X	Y	Transformation Formula
Linear	x	y	none
Exponential	x	ln y	Y = ln a + bx
Logarithmic	ln x	y	y = a + bX
Power	ln x	ln y	Y = ln a + bX
Inverse	1/x	y	y = a + bX

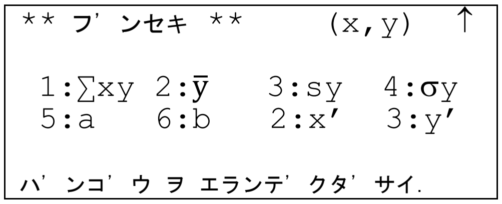
4.3.1. Bivariate Statistical Calculations

Press $\boxed{3}$ in the bivariate statistics menu to access the available bivariate statistical calculations. The following screen is displayed:



```
** フ’ ンセキ **      (x,y)  ↓

 1:n    2:∑x   3:∑x²  4:x̄
 5:sx   6:σx   2:∑y   3:∑y²

ハ’ ンコ’ ウ ヲ エランテ’ クタ’ サイ.
```

Note there is an arrow in the upper right corner of the screen to indicate there are additional statistical calculations available. To access the additional screen of calculations, press $\boxed{▼}$. The following screen is shown:



```
** フ’ ンセキ **      (x,y)  ↑

 1:∑xy  2:ȳ    3:sy   4:σy
 5:a    6:b    2:x’   3:y’

ハ’ ンコ’ ウ ヲ エランテ’ クタ’ サイ.
```

To return to the previous screen, press $\boxed{▲}$. Press the corresponding number to display the result of the calculation. To return to the bivariate statistics menu, press $\boxed{\text{BREAK}}$.

The following calculations are available:

n	:	Sample size
$\bar{x}, \bar{y}$	:	Mean of x, y
$\sum x$, $\sum y$	:	Sum of x, y
$\sum x^2$, $\sum y^2$	:	Sum of squares of x, y
$\sum xy$	:	Sum of products of x and y
sx, sy	:	Sample standard deviation of x, y. The formula for the sample standard deviation is: $sx = \sqrt{\frac{\sum x^2 - n\bar{x}^2}{n-1}}, sy = \sqrt{\frac{\sum y^2 - n\bar{y}^2}{n-1}}.$
σx, σy	:	Population standard deviation of x, y. The formula for the population standard deviation is: $\sigma y = \sqrt{\frac{\sum x^2 - n\bar{x}^2}{n}}, \sigma y = \sqrt{\frac{\sum y^2 - n\bar{y}^2}{n}}.$
a	:	Intercept of the linear regression line. The formula for the intercept is: $a = \bar{y} - b\bar{x}$
b	:	Slope of the linear regression line. The formula for the slope is: $b = \frac{Sxy}{Sxx}.$
r	:	Correlation coefficient. The formula for the correlation coefficient is: $r = \frac{Sxy}{\sqrt{Sxx \cdot Syy}}.$
x'	:	Estimated value of x. The formula is $x' = \frac{y-a}{b}.$
y'	:	Estimated value of y. The formula is $y' = ax + b.$
Where	:	$Sxx = \sum x^2 - \frac{(\sum x)^2}{n}, Syy = \sum y^2 - \frac{(\sum y)^2}{n}, Sxy = \sum xy - \frac{\sum x \cdot \sum y}{n}$

For x' and y', the computer will prompt for a value. Press $\boxed{\text{BREAK}}$ after the value is displayed to return to the bivariate calculation menu.

Example:

The following table contains the average temperature (in °C) for the month of April and the number of flowering cherry trees on the same day. Perform a linear regression analysis and calculate the slope, intercept, and correlation coefficient of this dataset. Also estimate how many flowering cherry trees do you expect to see at a temperature of 9.1°C and what would you expect the temperature to be if there are 10 flowering cherry trees.

Day	2	3	4	5	6	7	8	9
Temperature (x)	6.2	7.0	6.8	8.7	7.9	6.5	6.1	8.2
Cherry Blooms (y)	13	9	1	5	7	12	15	7

Keystrokes	Output
SHIFT + MDR ↵ (Enter **STAT** mode)	
2 (Select bivariable statistics)	
1 (Select Data Entry mode)	1:x=_
6 . 2 ↵ 1 3 ↵	1:x=6.2 y=13 2:x=_
7 . 0 ↵ 9 ↵	2:x=7.0 y=9 3:x=_
6 . 8 ↵ 1 ↵	3:x=6.8 y=1 4:x=_
8 . 7 ↵ 5 ↵	4:x=8.7 y=5 5:x=_
7 . 9 ↵ 7 ↵	5:x=7.9 y=7 6:x=_
6 . 5 ↵ 1 2 ↵	6:x=6.5 y=12 7:x=_
6 . 1 ↵ 1 5 ↵	7:x=6.1 y=15 8:x=_
8 . 2 ↵ 7 ↵	8:x=8.2 y=7 9:x=_
BREAK (Exit data entry)	
3 ↵ (Enter Bivariable cacluations)	
▼ (show 2nd calculation menu)	
5 ↵	a= 34.44951017
6 ↵	b= -3.425-18839
7 ↵	r= -9.691068372E-01
9 ↵	X=_
9 . 1 ↵	Y= 3.281838734
8 ↵	Y=_
1 0 ↵	X= 7.13850386
SHIFT + MDR ↵ (Exit **STAT** mode)	

4.4. Calculation Storage

Results of statistical calculations are stored in fixed variables U-Z (see section 5.2.2 Fixed Variables). The following table shows what calculations are stored in each variable. When you exit **STAT** mode, these values are retained. This allows access of the results statistical calculations in **RUN** mode. Be aware if you return to **STAT** mode, the contents of these variables are cleared and the values will have to be re-entered for further calculations.

Variable	U	V	W	X	Y	Z
Univariate	–	–	–	∑x2	∑x	n
Bivariate	∑y2	∑y	∑xy	∑x2	∑x	n

5. PROGRAM OPERATION

5.1. Constants

5.1.1. String Constants

The computer is able to process letters and special symbols in many ways besides numerically. These letters, numbers and special symbols are called characters.

In BASIC, a collection of characters is called a string. In order for the computer to understand the difference between a string and other parts of the program, e.g. If you want to recognize commands or variable names, you must enclose the string in quotes ("). To use quotation marks as characters, use "CHR$ &H22".

Here are some examples of string constants:

"HELLO"
"Goodbye"
"SHARP COMPUTER"

The following examples are not accepted as string constants:

"COMPUTER	quotes are missing at the end.
"VALUE OF "A" IS"	quotes may not be used within a string.

5.1.2. Hexadecimal

The hexadecimal system is based on the number 16 instead of the number 10. To write hexadecimal digits, use the digits 0 to 9, and six additional "digits" A, B, C, D, E and F. These correspond to the numbers 10, 11, 12, 13, 14, and 15. To use a hexadecimal number, place an ampersand (&) and "H" in front of the number:

&HA	=	10
&H10	=	16
&H100	=	256
&HFFFF	=	65535

5.2. Variables

Computers are made up of a number of very small memory units, called bytes. Each byte can be thought of as a single character. For example, the word "byte" requires four bytes of memory because it contains four letters. To find out how many bytes are available to use, simply enter the `FRE` command in **RUN** mode. The number displayed indicates how many bytes are free for programming.

This method works well for strings, but is inefficient for storing numbers. For this reason, numbers are stored in coded form. This allows the computer to store very long numbers in 8 bytes. The largest number that can be stored is +9.999999999E+99. The smallest number is 1E–99, a fairly large number range. However, if the result of the calculation exceeds these limits, the computer issues an error message.

Example:

```
R = 556
```

For string variables, the principle is the same. The computer recognizes the difference between string vs. numeric variables by the addition of a $ at the end of the variable name. For example, the word "BYTE" can be stored in the variable B$. Note the $ sign after B. This tells the computer that the contents of the variable B$ are alphanumeric or string data. More explicitly:

```
B$ = "BYTE"
```

5.2.1. Types of Variables

The variables that the computer uses are divided as follows:

Numeric variables:	Fixed numeric variables (A to Z)
	Simple numeric variables (AB, C1, etc.)
	Numeric array variables
String variables:	Fixed character variables (A$ to Z$)
	Simple string variables (BB$, C2$, etc.)
	String array variables

5.2.2. Fixed Variables

Fixed variables, are variables with pre-allocated memory. In other words, no matter how much memory the program uses, you will always have at least 26 variables available to store data. Each fixed variable is seven bytes long. There are two types of fixed variables: numeric and string variables (alphanumeric characters). For a specific fixed variable, both variable types share the same memory area. Once a variable has been declared a specific type (numeric vs. string), it cannot be used as the other type.

Example:

```
A = 123↵
A$↵
```

The following message is displayed:

```
ERROR 91
```

This means that numeric data has been allocated to a memory area called "A" and then the computer has been instructed to use this information as a string. The computer is confused and gives an error message. Press **CLS**/**CA** to clear the error message. Now the following is entered:

```
A$ = "ABC"↵
A↵
```

Again the computer is confused and gives the error message `ERROR 91`. The variable A corresponds in memory to the same area as the variable A$, furthermore B corresponds to the same memory area as B$ and so on for all letters of the alphabet.

Each fixed character variable can contain up to 7 characters and symbols.

5.2.3. Simple Variables

Simple variable names are specified by alphanumeric characters, e.g. AB or C8$. Unlike fixed variables, the simple variables do not have a declared storage area in memory. The area for simple variables is automatically allocated (within the program and data area) when the simple variable is first used. Separate memory areas are provided for simple numeric and simple string variables, even if they have the same name, e.g. AB and AB$ can be used simultaneously.

While alphanumeric characters are used to name simple variables, the first character must always be a capital letter. Two or more characters can be used to specify a variable name, but the computer only reads the first two.

Note: Computer-resident names for functions and BASIC commands, e.g. Pl, IF, TO, ON, SIN and others, cannot be used for variable names.

Each simple character variable can contain up to 16 characters and symbols. Each fixed character variable can contain up to 7 characters and symbols.

5.2.4. Array Variables

In some cases, it is useful to process numbers in organized groups, e.g. a table of football results or a tax table. In BASIC, these groups are called arrays. An array can be one-dimensional, e.g. a list, or two-dimensional, e.g. a table.

To define an array, use the `DIM` command (short for dimension). Arrays must always be defined before use (unlike the single-valued variables we used so far). The format for dimensioning numeric arrays is:

```
DIM array-name(size)
```

where:

`array-name` is the name of the array according to the above-mentioned naming rules for numeric or string variables.

`size` is the number of storage locations in the array and should be a number in the range of 0 to 255. When you specify a number, you get one more location than you specified.

Examples of allowed commands for numeric or string sizing:

```
DIM X(5)     → X(0), X(1), X(2), X(3), X(4), X(5)
DIM AA(24)
DIM Q5(0)
```

The first command creates an array `X` with 6 storage locations. The second command creates an array `AA` with 25 storage locations, the third is an array with one storage location, which is illogical (at least for numbers), because one could just as well define a numeric variable.

It is important to know that an array-variable `X` and a variable `X` are separate in the computer. The first `X` denotes a series of numeric memory locations, the second a single and different memory location.

Now that you know how to create arrays, you might be wondering how do you refer to each storage location? Since the entire group has only one name, the way we refer to a single storage location (called "element") is to follow the group name with a number in parenthesis. This number is called "subscript". For example, to store the number 8 at the fifth position in our (previously defined) array `X`, we would write:

```
X(4) = 8
```

If you are puzzled by the use of the number 4, remember that the numbering of array elements starts with zero and continues until the number of elements declared in the `DIM` statement.

The real power of arrays is the ability use an expression or a variable as an subscript.

The definition of a string array uses a slightly different form of the DIM statement:

```
DIM string-variable-name(size)[*length]
```

where:

`string-variable name` is the name for the string array that follows the aforementioned rules for variable names.

`Size` is the number of storage locations and should be a number in the range of 0 to 255. When you specify a number, you get one more location than you specified.

`*Length` is optional. If used, it will set the length of each string in the array. The length must be specified by a number from 1 to 255. If no entry is given, the default setting of 16 characters is used for the string.

Examples of legal string array definitions:

```
DIM X$(4)
DIM NM$(10)*10
DIM IN$(1)*255
DIM R$(0)*26
```

The first example creates an array of five strings of 16 characters each. In the second example, the `DIM` statement creates an array `NM` that contains 11 strings of 10 characters each. This assignment is unusual as the length of the string is shorter than the default of 16. The third example is a two-element array with a string length of 255 characters and in the last example a single element string with 26 characters.

The computer can also handle "two-dimensional" arrays. A one-dimensional array lists a sequence of data in a single column. A two-dimensional array corresponds to a table with rows and columns. Two-dimensional arrays are determined by the following statement:

```
DIM numeric-array-name(rows,columns)
```

or

```
DIM string-array-name(rows,columns)[*length]
```

where:

`Rows` is the number of lines in the array. It must be a number in the range of 0 to 255. Assigning a number to *rows* will provide one row more than specified.

`Columns` is the number of columns in the array. It must be a number in the range of 0 to 255. Assigning a number to *rows* will provide one row more than specified.

The following table illustrates the storage locations that result from the `DIM T(2,3)` instruction and the indexes associated with each storage location (two numbers in this example):

	Column 0	Column 1	Column 2	Column 3
Line 0	T (0,0)	T (0,1)	T (0,2)	T (0,3)
Line 1	T (1.0)	T (1,1)	T (1,2)	T (1,3)
Line 2	T (2,0)	T (2,1)	T (2,2)	T (2,3)

Note: Two-dimensional arrays take up a lot of space. For example, an array with 25 rows and 35 columns require 875 storage locations.

The following table shows the number of bytes needed to define each variable and the number of bytes required for each individual program instruction.

Variable type	Number of bytes used	
	Variable Name	Array Element
Numeric variable Numeric array variable	7 bytes	8 bytes
String variable	7 bytes	16 bytes
String array variable	7 bytes	Assigned number

Element	Line number	Command & Function	ENTER and others
Number of bytes used	3 bytes	2 bytes	1 byte

For example: for `DIM Z$(2,3)*10`, 12 variables are provided with a storage space of 10 characters each. 127 bytes are required: 7 bytes (variable name) + 10 bytes (number of characters) x 12.

5.3. Program and Data Files

Program and data files are fundamental in the use of your computer. Part of the computer's internal memory can be used as a RAM disk. Programs stored on the RAM disk must be loaded into the program data area (user area) before execution (See BASIC COMMAND LEXICON for instructions on the commands `SAVE`, `LOAD`, `KILL` and `FILES`). Programs can save data files to the RAM disk. Files to be stored on the RAM disc must be created in TEXT mode under Rfile before they are used. (See **TEXT** mode under Rfile).

5.3.1. File Names

Before saving a file to the RAM disk, the file must have a name. This name is used to identify the file to load into computer memory or to open with the `OPEN` command. The file name is arbitrary and can consist of up to 8 of the following characters:

A…Z, a…z, 0…9, 1, $, %, 8, ', (,), {, }, ", @

5.3.2. File Name Extensions

A file extension is an additional way to identify file types (such as BASIC program files, data files, or text files). The extension consists of three characters at the end of a file name which is separated from it by a period. The extension is specified when a file is saved.

BASIC programs automatically receive the `.BAS` extension if they are saved with the `SAVE` command. When reloaded into memory using the `LOAD` command, the `.BAS` extension does not need to be specified.

When the `FILES` or `LFILES` commands are used to list a directory of the RAM disk files, BASIC programs will appear with the `.BAS` extension, unless some other extension was specified by the user when the file was saved.

5.4. Expressions

An expression is a combination of variables, constants, and operators that can be evaluated into a single value. The calculation examples you entered before were examples of expressions. Expressions are an integral part of BASIC programs. For example, an expression may be a formula that computes the result of an equation, or a test to determine the relationship between two sizes, or a means to format a series of strings.

5.4.1. Numeric Operators

The computer has five numeric operators. These are the arithmetic operators that you used when exploring the use of the computer as a calculator:

+	Addition
–	Subtraction
*	Multiplication
/	Division
^	Exponentiation

A numeric expression is constructed in the same way that you entered compound calculator operations. Numeric expressions can contain any meaningful combination of numeric constants, numeric variables, and the numeric operators:

```
(A*B)^2
A(2,3)+A(3,4)+5.0-C
(A/B)*(C+D)
```

5.4.2. String Expressions

String expressions are similar to numeric expressions, except there is only a single string operator: concatenation (+). This is the same symbol used for addition. When used with a pair of strings, the + appends the second string to the end of the first string, creating a longer string. Be careful in making complex string concatenations and other string operations because the maximum work space for string calculations is 255 characters.

Note: String quantities and numeric quantities cannot be defined in the same expression, unless you use one of the functions that converts string values to numeric values, or vice versa:

"15" +10 is not allowed. "15" + "10" is "1510", not "25".

5.4.3. Relational Expressions

A relational expression compares two expressions and indicates whether the established condition is true or false. The relational operators are:

>	greater than
>=	greater than or equal to
=	equal
<>	unequal
<=	less than or equal to
<	less than

The following are valid relational expressions:

```
A < B
C(1,2) >= 5
D(3) <> 8
```

If `A` were equal to 10, `B` equal to 12, `C(1,2)` equal to 6, and `D(3)` equal to 9, all of these expressions would be true.

Character strings can also be compared using relational expressions. The two strings are compared character by character according to their ASCII value starting at the first character (see Appendix H). If one string is shorter than the other, a 0 or NULL will be used for any missing positions. All the following examples are true.

```
"ABCDEF" = "ABCDEF"
"ABCDEF" <> "ABCDE"
"ABCDEF" > "ABCDE"
```

Relational expressions are evaluated as true or false. The computer represents true with -1, false with a 0.

5.4.4. Logical Expressions

Logical operations use the Boolean algebra functions `AND`, `OR`, `XOR`, and `NOT` to build connections between relational expressions. The logical operations in a single expression are evaluated after arithmetic and relational operations.

In this way, logical operators can be used to make program decisions from multiple conditions using the `IF…THEN` statement.

Example:

```
IF A<=32 AND B>=90 THEN 150
```

This statement causes the execution to jump to line 150 if the value of the numerical variable A is less than or equal to 32 and, at the same time, the value of the numerical variable B is greater than or equal to 90.

```
IF X<>13 OR Y=0 THEN 50
```

This statement causes the execution to jump to line 50, unless the variable X has the value 13, or the variable Y is not equal to 0.

In a logical operation involving two numbers in the range -32768 to +32767, the two numbers are converted to 16-bit binary integers (in two's complement form), and the logical connection is then evaluated for each corresponding bit of the two numbers.

The results returned by the logical operators for these bit evaluations, are listed below:

AND

X	Y	X AND Y
1	1	1
1	0	0
0	1	0
0	0	0

OR

X	Y	X OR Y
1	1	1
1	0	1
0	1	1
0	0	0

XOR

X	Y	X XOR Y
1	1	0
1	0	1
0	1	1
0	0	0

NOT

X	NOT X
1	0
0	1

After each bit pair has returned the corresponding result (a 1 or 0) according to the above tables, the resulting 16-bit binary number is converted back to a decimal vale. This number is the results of the logical operation.

Example:

```
41 AND 27 →             41 = 101001
equals                  27 = 011011   AND
9                          ← 001001

41 OR 27 →              41 = 101001
equals                  27 = 011011   OR
59                         ← 111011

41 XOR 27 →             41 = 101001
equals                  27 = 011011   AND
50                         ← 110010

NOT 3 →                 3 =  0000000000000011
equals                                          NOT
-4 (two's complement form) ← 1111111111111100
```

5.4.5. Parenthesis and Operator Precedence

When working on complex expressions, the computer follows a predefined set of priorities that determine the sequence in which the operators are evaluated. This can be quite significant:

5 + 2 * 3 could be

5 + 2 = 7	or	2 * 3 = 6
7 * 3 = 21		6 + 5 = 11

The exact rules of "operator precedence" are on page 19.

To avoid having to remember all the rules and to make your programs more precise, always use parentheses to specify the sequence of evaluation. The above example is clarified by writing either:

(5 + 2) * 3 or 5 + (2 * 3)

6. PROGRAMMING IN BASIC

In the previous chapter, we examined some of the concepts and terms of the BASIC programming language. In this section we now want to use these elements to create programs. However, this is not a manual on how to program in BASIC. This chapter will familiarize you with the use of BASIC on your computer.

6.1. Programs

A program consists of a series of instructions to the computer. Remember that the computer is only a machine. It will perform the exact operations that you specify. You, the programmer, are responsible for issuing the correct instructions.

6.1.1. BASIC Statements

The computer interprets instructions according to a predetermined format. This format is called statement. You must always enter the BASIC statements in the same pattern. Statements must start with a line number:

Example:

```
10: INPUT A
20: PRINT A*A
30: END
```

6.1.2. Line Numbers

Each line of a program must have a unique line number – any integer between 1 and 65279. Line numbers are the reference for the computer. They tell the computer the order in which to run a program. You do not need to enter lines in sequential order (although if you are a beginning programmer, it is probably less confusing for you to do so). The computer always begins execution with the lowest line number and moves sequentially through the lines in ascending order.

You can use the `AUTO` command to automatically insert line numbers. Each time you press the ↵ key, a new line number, with the correct increment, will automatically be inserted. See the BASIC COMMAND Glossary for a full description of this useful function.

It is wise to allow increments of several numbers in your line numbering. (10, 20, 30, ... 10, 30, 50, etc.). This enables you to insert additional lines, if necessary. If you use the same line number, the older line with that number is deleted when you enter the new one.

6.1.3. Labeled Programs

Often you will want to store several different programs in memory at the same time. (Remember that each must have unique line numbers.) Normally, to start a program with a `RUN` or `GOTO` command, you will normally need to remember the beginning line number of each program. However, there is an easier way. You can label each program with alphanumeric characters and run the program.

Label the first line of each program you want to reference. The label consists of a letter and up to 19 alphanumeric characters with a * in front or in quotes, followed by a colon.

Example:

```
10: *A: PRINT "FIRST"
20: END
80: "B": PRINT "SECOND"
90: END
```

Although both *Label and "Label" forms may be used, *label is recommended because it executes more quickly and more visible in a program listing.

6.2. BASIC Commands

All BASIC statements must contain commands. These commands tell the computer what action to perform. A command is contained within a program, and as such is not acted upon immediately.

Some instructions require or allow an operand:

Example:

```
10: DATA "HELLO"
20: READ B$
30: PRINT B$
40: END
```

Operands provide information to the computer telling it what data the command will act upon. Some commands require operands, while with other commands, they are optional. Certain commands do not allow operands. (See the **BASIC COMMAND GLOSSARY** for BASIC commands and their uses.)

Note: Commands, functions and variables entered in lower case characters will be converted to uppercase characters.

6.2.1. Direct Commands

Direct commands are instructions to the computer that are entered outside of a program. They instruct the computer to perform some immediate action or set modes that affect how your programs are executed.

Direct commands have immediate effect – as soon as you complete entering direct commands (by pressing the ↵ key), the command will be executed. Direct commands are not preceded by a line number.

```
RUN
NEW
RADIAN
```

6.2.2. Modes (Operating Modes)

When you used the computer as a calculator, you were working in **RUN** mode. **RUN** mode is also needed to execute the program you have entered. Use **PRO** mode if you want to enter or edit programs.

6.3. Beginning to Program

To enter program statements into the computer, the computer must first be placed in PRO (program) mode using the **BASIC** key. The following display will appear:

```
>
```

Enter the `NEW` command.

```
NEW
>
```

The `NEW` command clears the memory of all existing programs and data. The prompt appears after you press the ↵ key, indicating that the computer is awaiting input.

6.3.1. Entering and Running a Program

Make sure the computer is in **PRO** mode and enter the following program:

10PRINT **SHIFT** + **"** HELLO **SHIFT** + **"**

```
PROGRAM MODE
NEW
10:PRINT"HELLO"_
```

Notice that the computer automatically inserts the colon between the number and the command when you press the ↵ key.

Check that the statement is in the correct format and then change the mode to **RUN** by pressing the **BASIC** key.

CLS RUN↵

```
RUN
HELLO
>
```

Since this is the only line of the program, the computer will exit the program and return to the BASIC prompt “>”.

6.3.2. Editing a Program

Suppose you want to change the message that your program was displaying. That is, you wanted to edit your program. With a single line program, you could just retype the entry, but as you develop more complex programs, editing becomes a very important component of your programming. Let’s edit the program you have just written.

Switch to **PRO** mode. You need to recall your program in order to edit it. Use the up arrow key ▲ to recall your program. If your program was completely executed, the ▲ key will recall the last line of the program. If there was an error in the program, or if you used the .BREAK. key to stop execution, the ▲ key will recall the line in which the error or break occurred. To make changes in your program, use the ▲ key to move up in your program (recall the previous line) and the ▼ key to move down in your program, display the next line). If held down, the ▲ or ▼ key will scroll vertically (up or down) through your program.

Remember that to move the cursor within the program line, you use the ◀ (right arrow) and ▶ (left arrow) keys. Using the ▶ key, position the cursor over the first character you wish to change:

▲

```
10:PRINT "HELLO"
```

▶▶▶▶▶▶▶▶

```
10 PRINT "HELLO"
```

Notice that the cursor is now in the flashing block form, indicating that it is on top of an existing character. Enter:

GOODBYE SHIFT + " SHIFT + !

```
10 PRINT "GOODBYE"!
```

Remember to press the ↵ key at the end of the line. Change to **RUN** mode.

RUN↵

```
RUN MODE
RUN
ERROR 10 IN 10
```

The error message indicates the type of error, and the line number in which the error occurred. Press the **CLS** key to clear the error condition and return to **PRO** mode. You must be in **PRO** mode to make changes in a program. Using ▲ (or ▼), recall the line in which the error occurred.

▲ (or ▼)

```
10 PRINT "GOODBYE"!
```

The flashing cursor is positioned over the error. You learned that when entering string constants in BASIC, all characters must be contained within quotation marks. Use the **DEL** key to eliminate the "!".

DEL

```
10 PRINT "GOODBYE"_
```

Now let's put the ! in the correct location. When editing programs, **DEL** and **INS** are used in exactly the same way as they are in editing calculations. Using ◀, position the cursor on top of the character that will be the first character following the insertion.

◀

```
10 PRINT "GOODBYE"
```

Press the **INS** key. A ◀ will indicate where the new data will be entered.

INS

```
10 PRINT "GOODBYE"
```

Enter the !. The display looks like this:

SHIFT + **!**

```
10 PRINT "GOODBYE!"
```

Remember to press the ↵ key so the correction will be entered into the program.

Note: If you wish to delete an entire line from your program, just enter the line number and the original line will be eliminated. The `DELETE` command can be used to delete more than one line at a time.

6.3.3. Using Variables in Programming

Using variables in programming allows more sophisticated use of the computer's abilities. The values assigned to a variable can change during the execution of a program, taking on the value entered or computed during the program. One way to assign a variable is to use the `INPUT` command. In the following program, the value of `A$` will change in response to the data typed in answer to the inquiry "`WORD?`".

Enter the following program:

```
10:INPUT ”WORD?”;A$
20:B=LEN(A$)
30:PRINT “THE WORD (”;A$;”) HAS”
40:PRINT “HAS “;B;” LETTERS”
50:END
```

The second new element in this program is the use of the `END` statement to signal the completion of the program. `END` tells the computer that the program is completed. It is always good programming practice to use and `END` statement.

As your program become more complex, you may wish to review them before you begin execution. To look at your program, use the `LIST` command. `LIST`, which can only be used in **PRO** mode, displays programs beginning with the lowest number. Try listing this program:

LIST↵

```
10:INPUT ”WORD?”;A$
20:B=LEN(A$)
30:PRINT “THE WORD (”;A$
  ;”)”
40:PRINT “HAS “;B;” LETT
  ERS”
50:END
```

Use the ▲ and ▼ keys to move through your program until you have reviewed the entire program. After checking your program, change to **RUN** mode and run it.

CLS RUN↵

```
RUN
WORD?
```

HELP

```
RUN
WORD?HELP
```

↵

```
RUN
WORD?HELP
THE WORD (HELP)
HAS 4. LETTERS
```

This is the end of your program. Of course you may begin it again by entering `RUN`. However, this program would be a bit more entertaining if it presented more than one opportunity for input. We will now modify the program so it will keep running without entering `RUN` after each answer.

Return to PRO mode and use the ▲ or ▼ keys (or `LIST`) to reach line 50, or enter:

LIST50↵

```
50:END
```

You may enter 50 to delete the entire line or use the key to position the cursor over the E in `END`. Change line 50 so that it reads:

```
50:GOTO 10
```

Now `RUN` the modified program.

The `GOTO` statement causes the program to loop (keep repeating the same operation). Since you put no limit on the loop, it will keep going forever (an "infinite" loop). To stop this program, press the **BREAK** key.

When you have stopped a program using the **BREAK** key, you can restart it using the `CONT` command. The program will restart on the line that was being executed when the **BREAK** key was pressed.

6.3.4. More Complex Programming

The following program computes N factorial (N!). The program begins with 1 and computes N! up to the limit that you enter. Enter this program:

```
100:F=1: WAIT 118
110:INPUT"LIMIT?";L
120 FOR N=1 TO L
130:F=F*N
140:PRINT N,F
150:NEXT N
160:END
```

Several new features are contained in this program. The `WAIT` command in 100 controls the time that displays are held before the program continues. The numbers and their factorials are displayed as they are computed. The time they appear on the display is set by the `WAIT` statement to approximately 2 seconds.

Notice that there are two statements in line 100 separated by a colon (:). You may put as many statements as you wish on one line (separating each by a colon) up to a maximum of 254 characters including the ↵ key. Multiple-statement lines can make a program hard to read and modify, so it is good programming practice to use them only where the statements are very simple or there is some special reason to want the statements on one line.

In this program, we have used the `FOR` command in line 120 and the `NEXT` command in line 150 to create a loop. In the previous example, you created an "infinite" loop that kept repeating the statements inside the loop until you pressed the **BREAK** key. With this `FOR…NEXT` loop, the computer adds 1 to N each time execution reaches the `NEXT` command. It then tests to see if N is larger than the limit L. If N is less than or equal to L, execution returns to the top of the loop and the statements are executed again. If N is greater than L, execution continues at line 160 and the program stops.

You may use an fixed numeric variable or simple numeric variable in a `FOR…NEXT` loop. You do not have to start counting at 1 and can increment by any amount at each step. See the **BASIC COMMAND GLOSSARY** for details.

We have labeled this program with line numbers starting with 100. Labeling programs with different line numbers allows you to have several programs in memory at one time. To `RUN` this program instead of the one at line 10, change to **RUN** mode and enter:

CLS
RUN100

You could also give the program a name using a label and start the program with `RUN *LABEL`.

If more than six lines must be displayed, the first lines will scroll up off the display and cannot be recalled. Use the `WAIT` command in the program to display data more slowly, or use the printer. The `WAIT` command applies to every `PRINT` command. Break long `PRINT` commands into a number of shorter commands if the display scrolls too quickly.

6.4. Debugging

After entering a new BASIC program, it will often not work the first time. Even if you are entering a program that you know is correct, such as those provided in this manual, it is common to make at least one typing error. It may also contain at least one logical error as well.

Here are some general hints on how to find and correct your errors.

You run the program and receive an error message:

1. Switch back to **PRO** mode and use the cursor keys ▲ or ▼ key to review the line with the error on display. The cursor is at the location where the error occurred.
2. If you cannot fine an obvious syntax error, the problem may be with the values that are being used. For example, `CHR$(A)` generates a space if `A` is 1. Check the values of the variables you are using in either **RUN** or **PRO** mode by entering the name of the variable and pressing the ↵ key.

You run the program with `RUN` and don't get an error message, but the program doesn't do what you expect:

1. Check the program line by line using `LIST` and the ▲ and ▼ keys see if you entered the program correctly. It is surprising how many errors can be corrected when you take another look at the program.
2. Think about each line as you go through the program as if you were the computer. Take simple values and try to apply the operation in each line to see if you get the result you expected.
3. Insert one or more extra `PRINT` statements in the program to display key values and key locations. Use these to isolate the parts of the program that are working correctly and the location of the error. This approach is also useful for determining which parts of a program have executed. You can also use `STOP` to temporarily halt execution at critical points so that several variables can be examined.
4. Use `TRON` (TRace ON) and `TROFF` (TRace OFF), either as direct commands or within the program to trace the flow of the program through individual lines. Stop to examine the contents of critical variables at crucial points. This is a very slow way to find a problem, but it is sometimes the only way.

6.4.1. Trace Mode

No matter how careful you are, eventually you will create a program that does not do what you expect it to do. To isolate the problem, BASIC has a special method of executing programs known as "Trace" mode.

`TRON` (TRace ON) starts **Trace** mode. The `TRON` instruction can be used as a direct command (in **RUN** mode) or can be embedded within a program. Used as a direct command, `TRON` informs the computer that tracing will be required during the execution of all subsequent programs. The programs to be traced are then started in the usual fashion, using the `GOTO` or `RUN` command.

If `TRON` is used within a program, it will initiate **Trace** mode only when the line it is executed. If, for some reason, the line is never reached, **Trace** mode will remain inactive.

6.4.2. Debugging Procedures

1. Set the computer to `RUN` mode.
2. Enter `TRON`↵ to specify trace mode.
3. Enter `RUN`↵ to run the program. After executing each line, the computer stops execution and displays the current line number.
4. Use the ▼ key to examine the current line. If the ▼ key is held down, the program continues execution line by line. Releasing the ▼ key stops program execution. To examine the contents of the last executed line, press and hold the ▲ key. When the ▼ key is released, the BASIC command prompt ">" appears. To resume execution, press the ▼ key.
5. If execution of the program is interrupted during data entry with the INPUT statement, press ↵ to continue program execution.
6. Continue the trace procedure and check if the program is executing properly by confirming program execution order and variable contents after each line is executed. If the program is not executing properly, correct the logic.
7. After debugging, enter `TROFF`↵ to exit trace mode.

Example:

```
10 INPUT "A =";A,"B =";B
20 C = A*2
30 D = B*3
40 PRINT "C =";C;"D =";D
50 END
```

Run the program.

RUN mode

```
TRON↵          >
RUN↵           A =_
8↵             B =_
9↵             10:
▼              20:
▼              30:
▼              C = 16.    D = 27.
               40:
```

If the execution is interrupted by the **BREAK** key, review the variables manually and check that the values are as expected. Pressing the ▼ key will execute one statement at a time and entering `CONT`↵ will execute the statements continuously.

Note:

- If a result or other information is displayed at the location specified by `LOCATE`, the line number appears on the line after this. (See **BASIC COMMAND GLOSSARY** for instructions on the `LOCATE` command)

- If a variable is accessed manually or a manual calculation is performed after `LOCATE` was assigned, it assignment is lost.
- Trace mode remains active until `TROFF` is entered, the **SHIFT** + **CA** keys are pressed, or power is interrupted.
- When executing a comment line in trace mode, no line number is displayed. In this case, the number of the last executed line remains on the display.

To troubleshoot by interrupting a running program, do the following:

1. Press the **BREAK** key while running the program
2. Enter the `STOP` command at the appropriate location.

The computer indicates that the program has stopped and execution is interrupted. Afterwards, manually check the contents of the variables. Press the ▼ key to execute the instructions line by line, `CONT`↵ for continuous execution.

7. TEXT MODE

In **TEXT** mode (text editor) you can write and edit programs (BASIC, C, assembler or CASL) in ASCII format. In the same way, data files can be created, edited or deleted. Programs as well as data can be stored on the RAM disk or sent/received via the serial I/O interface.

BASIC instructions for the computer are stored in a 2-byte format called "intermediate code". Since this code differs depending on the hardware or BASIC interpreter used, it cannot be used for communication between personal computers or other devices. ASCII code is commonly used for communication between personal computers because representation of alphanumeric characters and primitive symbols is the same, regardless of the hardware used. With **TEXT** mode, you can write, edit and save programs in ASCII. Programs can also be converted from intermediate code (BASIC) to ASCII and vice versa. This section describes the functions of **TEXT** mode.

When you press the **TEXT** key, you will see a screen like the one on the right.

```
    *** TEXT EDITOR ***

  Edit Del  Print
  Sio  File Basic Rfile
```

TEXT mode can be exited at any time by switching to another mode (**RUN**, **PRO**, **ASMBL**, **CASL**, **C**). Data already entered is not lost and can be further edited by pressing **TEXT** + **E** (for Edit).

To get to the main menu from any submenu of **TEXT** mode, press the **TEXT** key. To go up one menu level, press the **BREAK** key.

7.1. Functions in TEXT mode

In **TEXT** mode, the following functions are available:

Mode	Operation
`Edit`	Creating and editing programs or files
`Del`	Delete programs or files in the editor
`Print`	Send a program listing or data to the printer
`Sio`	Serial I/O port
`  Save`	Send program or data via the serial interface
`  Load`	Load program or data from the serial interface
`  Format`	Configure the serial interface
`File`	Program file operations on the RAM Disk
`  Save`	Store a program
`  Load`	Load a program
`  Kill`	Delete a program
`  Files`	Retrieve/display all programs on the RAM disk
`Basic`	Convert file between BASIC and TEXT formats

`Basic←text`	Conversion from TEXT to BASIC
`Text←basic`	Conversion from BASIC to TEXT
`Rfile`	Data file operations on the RAM disk
`Init`	Create data file
`Save`	Save data file
`Load`	Load data file
`Kill`	Delete data file
`Files`	Retrieve/display all data files on the RAM disk

7.1.1. Editing Programs and Files

Select the edit function from the main menu, press $\boxed{\textbf{E}}$.

In the edit function, the prompt in the command line is "<" (instead of ">" in BASIC).

```
TEXT EDITOR
<
```

As with a BASIC program, each line of a TEXT program begins with a line number. However, the computer does not automatically add a colon after the line number (:), as with BASIC programs. Also, a space is not automatically inserted between commands. Each line appears exactly as it is typed.

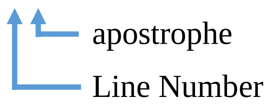
Note: - Line numbers are automatically sorted in ascending order.

- The range of possible line numbers for a program is from 1 to 65279. If this range is exceeded or no line number is entered, an error message (**LINE NO ERROR**) is displayed. Press $\boxed{\textbf{CLS}}$ / $\boxed{\textbf{CA}}$ to clear the error message.

To return to the main menu press $\boxed{\textbf{BREAK}}$.

Note: A TEXT line cannot begin with a number directly after the line number. If the line should necessarily begin with a number, an apostrophe (') must be inserted between the line number and the number.

50 '100 FORMAT (17X, A)



(Example program) Enter the following program:

```
10INPUT A
20B=A*A
30PRINT A,B
40END
```

10INPUT $\boxed{\textbf{SPACE}}$ A↵

```
10INPUT A
```

20B =A*A
30PRINT SPACE A,B
40END

```
20B=A*A
30PRINT A,B
40END
```

7.2. TEXT Editor

A TEXT program is edited just like a BASIC program. (See the explanations for programming in BASIC)

The **TEXT** mode edit commands correspond to BASIC commands. (For details of the commands, see the explanations in the **BASIC COMMAND GLOSSARY**)

Commands:

A	Auto numbering (see also the `AUTO` command on page 177)
L	List the lines (see also the `LIST` command on page 210)
R	Renumber (see also `RENUM` command on page 226)
D	Delete lines (see also `DELETE` command on page 188)
C	Copy lines (see also `LCOPY` command on page 205)
S	Search for string
E	Find and replace string

If the R command is executed in a TEXT program that was converted by a BASIC program, only the line numbers at the beginning of a line are renumbered, while the line numbers within `GOTO`, `THEN`, `GOSUB`, or `RESTORE` statements are not. In this case, the program does not run when it is converted back to BASIC.

A — Auto

Format: `A [[`*`start-line-number`*`][,`*`interval`*`]]`

Description: After starting A, the first line number appears in the display with a trailing cursor. The desired content can now be entered. Pressing the key, generates the next line number and so on.

L — List

Format:
`L`
`L` *`line number`*
`L` *`label`*

Description: Lists the program from the beginning or from the specified line number or labels.

R — Renumber

Format: `R [`*`oldline`*`[,`*`newline`*`][,`*`interval`*`]]`

Description: Re-number all rows or specified rows with specified interval.

D Delete

Format: `D` *`start line number`*`[,[`*`endline number`*`]]`

Description: Deletes the specified line or all lines from the specified start line up to and including the specified end line number. The remaining syntax variants correspond to the `DELETE` command.

C Copy

Format: `C` *`startline`*`,`*`endline`*`,`*`destination`*

Description: Copies the lines *startline* … *endline* to *destination*.

Caution: Jump addresses are not changed in BASIC commands

S Search

Format: `S [0|1,]` *`string`*

Description: Searches for a string in the text file. If the string is found, the cursor is placed on the first character of the matching string. Press ↵ to move the cursor to the next matching string. CLS ends the search. The search string may have a maximum length of 16 characters. Specifying 0 or 1 determines the search direction:

1: Search forwards from the beginning of the file.
0: Search backwards from the end of the file.

If this parameter is not specified, the search will start from the beginning of the file (1).

When searching for a " (double quotation mark), use ¥" as a string for ". For example: S "¥"".

E Replace

Format: `S [0|1,]` *`search string, replacement string`*

Description: Finds and replaces a string in the editor. If the string is found, the cursor is placed on the first digit of the matching string. Press ↵ to move the cursor to replace the characters and jump to the next matched string. SPACE does not replace the string and the cursor is moved to the next matching string. CLS ends the search.

Strings may have a maximum length of 16 characters. Specifying 0 or 1 determines the search direction:

1: Search forwards from the beginning of the file.
0: Search backwards from the end of the file.

When searching for a " (double quotation mark), use ¥" as a string for ". For example: E "¥""

7.2.1. The TAB Key

In **EDIT** mode, pressing the TAB key moves the cursor to the next column. When the TAB key is first pressed, the cursor moves to column 8. At the next press, the cursor moves to column 14 (6 digits after the first tab position). Each subsequent press of the TAB key moves the cursor seven places forward (to 21, 28, etc.…).

7.3. Delete TEXT Memory (Del)

Select the delete function from the TEXT menu, press D.

```
    *** TEXT EDITOR ***

TEXT DELETE OK? (Y)

```

If the Y key is pressed, the entire TEXT memory area is completely deleted, including the TEXT program, and the main menu is displayed.

If any key other than Y is pressed, the computer returns to the main menu without deleting anything.

Note: If no text is stored in the TEXT memory, the computer will not respond to the D key and returns to the main menu.

7.4. Print TEXT Program (Print)

Connect the CE-126P printer to the computer and turn on the computer and the printer. Display the TEXT main menu and press P to print the stored TEXT program.

```
    *** TEXT EDITOR ***


  --- PRINTING ---

```

After printing, the computer displays the main menu.

Note: To cancel printing, press BREAK. If the printer is not turned on or is not connected to the computer, the computer will not respond to P when the main menu is displayed.

7.5. Serial Input/Output (SIO)

Pressing the S key in the TEXT main menu will display the serial input / output menu (SIO menu). Select the corresponding function `Save` (send), `Load` (receive) or `Format` from the SIO menu by entering the first letter of the function (S, L or F).

```
          << SIO >>

 Save   Load   Format

```

7.5.1. Set I/O Parameters (Format)

Serial communication parameters can be set with this menu. The communication parameters must match the device that this computer will communicate with. To display a help menu from the SIO menu, press F.

```
          << SIO >>

 Select   ←,→,↑,↓ key
 Set      ↲ key

  --- Push any key ---
```

Press any key or wait until the computer displays the communication settings.

```
     →baud rate =1200
      data bit  =8
      stop bit  =1
      parity    =none
      end of line =CR LF
      end of file =1A
```

→ indicates the chosen parameter. You can move → to a different setting to change with the ▲ or ▼ keys. There are a total of seven settings that can be set. With the ▼ key, you can scroll through all the settings on the display.

The ◀ and ▶ keys change values of the setting indicated by →. However, the setting for the "`end of file`" must be entered manually. After entering the changes, press ↵ to save the changes. If the new settings are not saved, the computer will use the previously set parameters.

Communication Parameters

`baud rate`	: 300, 600, 1200, 2400, 4800, 9600 Sets the baud rate. The baud rate is the speed that data is transmitted, the higher rate, the faster the speed. Allowable baud rates are 300, 600, 1200, 2400, 4800 and 9600 bps (bits per second).
`data bit`	: 7 or 8 Specifies the number of bits needed to represent a character. It can be set to either 7 or 8 bits.
`stop bit`	: 1 or 2 Specifies the length of the stop bit at the end of character.
`parity`	: none, even or odd Specifies the type of data check (parity check). none : no parity. even : even parity. odd : odd parity.
`end of line`	: CR, LF or CR + LF Specifies the code for the end of each program line. CR : carriage return. LF : line feed. CR + LF : both CR and LF.
`end of file`	: 00 to FF (two-digit hexadecimal number) Specifies the code to indicate the end of a program or other file.
`line number`	: yes or no Specifies whether a TEXT program is sent with or without line numbers. yes : the program is sent with line numbers. no : the program is sent without line numbers. Line number also determines whether a line number (in increments of 10) should automatically be added to each program line upon receipt. yes : no line numbers are added. "yes" is selected if the program already contains line numbers. no : line numbers are automatically added. If the received file does not contain line numbers, even though "yes" was set, an error message (LINE NO, ERROR) is displayed.
`flow`	: RS/CS, Xon/Xoff, or none Specifies how information exchanged through the serial port is controlled. RS / CS : flow control is controlled by the RS/CS signals. Xon/Xoff : flow control is through the Xon/Xoff protocol none : transmission is carried out without any flow control.

The settings apply to all subsequent `FOPEN("stdaux1")`, or `OPEN "COM1"` commands. Once the settings have been changed and saved, these new parameters will apply until the

RESET button is pressed to clear the memory, the battery is replaced, or the settings are changed again.

7.5.2. Send Program (Save)

Pressing the **S** key in the SIO menu will send a program or data stored in TEXT memory to the serial I/O port.

```
          << SIO >>

 ---  SENDING ---

```

After sending is complete, the computer returns to the SIO menu.

Note:

1) To cancel the transmission, press the **BREAK** key. The computer returns to the SIO menu.
2) If there is no program or data stored in TEXT memory, the computer will not respond to the **S** key.

7.5.3. Receive Program (Load)

Pressing the **L** key in the SIO menu will load data from the serial I/O port to TEXT memory.

```
          << SIO >>

 ---  RECEIVING ---

```

After receiving is complete, the computer returns to the SIO menu.

Note:

1. To cancel the reception, press the **BREAK** key. The computer goes back to the SIO menu.
2. If the program was not received correctly or if a parity error occurred, an error message will appear (**I/O DEVICE ERROR**). To clear the error message, press **CLS** / **CA**.

7.5.4. Printing

Pressing the **L** key in the main menu will allow printing via the parallel interface (Centronics protocol).

For more information on using the parallel interface, see also `INP` and `OUT` commands.

7.6. Program File Management (File)

Pressing the F key in the main menu will display the ram disk file menu.

```
  << PROGRAM FILE >>

Save  Load  Kill  Files

```

From this menu, the corresponding function "Save, Load, Delete, or View Files" is selected by entering the first letter of the function (S, L, K or F).

7.6.1. Save TEXT Program (Save)

Pressing the S key in the program file menu prompts for a name for the TEXT file. Enter the file name and press ↵. The computer now saves the contents of TEXT memory in this file.

Saving a file with the file name "TEST".

S

```
  << PROGRAM FILE >>

→Save  Load  Kill  Files


FILE NAME=?
```

T E S T ↵

```
  << PROGRAM FILE >>

→Save  Load  Kill  Files


FILE NAME=?TEST
```

The computer saves the file "TEST" and then returns to the program file menu.

Note:

- You must enter a file name. If the ↵ key is pressed without entering a name, **ILLEGAL FILE NAME** error is displayed. To clear the error, press CLS.
- A file name can consist of up to eight characters and an extension of up to three characters. If no extension is entered, the computer automatically assigns the extension `.TXT`.
- If there is no TEXT program is stored in TEXT memory, saving is aborted.

7.6.2. Load TEXT Program (Load)

Pressing the L key in the program file menu displays a list of stored files. "LOAD →" points to the first file name (if no program has been saved, the computer will not respond to the L key).

Example of a list of files

```
LOAD →ABC        .TXT   456
      PRO        .TXT  1234
      カタカナ      .BAS  1567
      TEST       .TXT   789
```

Use the ▲ and ▼ keys to move "LOAD →" to the name of the file to be loaded; then press ↵. The computer loads the contents of the selected file into the TEXT area and then returns to the file menu.

Note: Only programs and files created in **TEXT** mode can be loaded. Trying to load a program that was saved using the BASIC SAVE command results in a **FILE MODE ERROR**. To clear the error, press CLS.

7.6.3. Delete Program File (Kill)

This function deletes a file. Pressing the K key in the program file menu prompts for the name of the file to be deleted.

```
   << PROGRAM FILE >>

Save  Load →Kill  Files


FILE NAME=?
```

Enter the name of the file to be deleted and press ↵. The computer then asks for confirmation.

```
FILE DELETE OK? (Y)
```

Pressing the Y key confirms the deletion. Any other key will cancel the deletion and the program file menu will be displayed. If the file name does not contain an extension, the suffix .TXT is added by default.

If the specified file does not exist, the computer issues a **FILE NOT FOUND** error.

7.6.4. List File Names (Files)

Pressing the **F** key in the program file menu displays a list of all stored files. A "→" points to the first file name on the list. (If no files are stored, the computer does not respond to the **F** key).

Example of a list of files.

```
LOAD →ABC        .TXT   456
      PRO        .TXT  1234
      カタカナ     .BAS  1567
      TEST       .TXT   789
```

The list can be scrolled by pressing the ▲ and ▼ keys.

To load a program marked with →, press **SHIFT** + **M** (or **2ndF** + **M**).

7.6.5. About TEXT files

The size of the text file is the total number of bytes for each line. The number of bytes in each line is calculated from the line number (3 bytes), the linefeed (1 byte) and the number of characters in the text of the line.

Example:

10_INPUT_A↵
results in 3 + 8 + 1 = 12 bytes for this line.

When converting to BASIC code, the program length becomes shorter because the BASIC keywords require fewer bytes.

7.7. BASIC Converter (Basic)

This function converts a BASIC program in intermediate code into a TEXT file in ASCII code or vice versa. This feature is useful for editing BASIC programs written for the G850V(S) on a personal computer.

Pressing **B** in the main menu will show the BASIC converter menu.

```
 << BASIC CONVERTER >>

 Basic→text  Text←basic
```

From this menu, conversion from TEXT to BASIC or from BASIC to TEXT can be selected. Enter the first letter of the format you want to convert to.

7.7.1. Conversion of TEXT and BASIC Programs

Pressing the **B** key in the BASIC converter menu converts the TEXT program in TEXT memory into a BASIC program and saves it to program memory.

Pressing the **T** key in the BASIC converter menu converts the BASIC program in program memory into a TEXT program and saves it to TEXT memory.

Example:

Convert a TEXT program into BASIC.
B

```
<< BASIC CONVERTER >>

--- CONVERTING ---
```

After conversion, the computer returns to the main menu. (Converting a short program takes very little time to convert.)

If there is a BASIC program loaded while a TEXT program is being converted, or vice versa, the computer asks for confirmation on whether the existing program should be deleted before the conversion.

```
Basic→text  Text←basic

BASIC DELETE OK? (Y)
```

When you press **Y**, the computer deletes the existing BASIC program and begins the conversion. Pressing any other key will cancel the conversion and the computer will return to the main menu.

In general, the computer keeps the original program after it has been converted to another format. However, if there is not enough memory available after a program has been converted, the computer asks for confirmation on whether the original program should be deleted.

```
 --- CONVERTING ---

TEXT DELETE OK? (Y)
```

Pressing the **Y** key will clear the original program during the process of conversion. At the end of the conversion, the original program is deleted. Pressing any other key will cancel the conversion and the computer will return to the main menu.

7.7.2. Out of Memory when Using the TEXT/BASIC Converter

If during a BASIC conversion the computer detects that there is not enough memory to hold both versions, the source version will be deleted line by line during the conversion. If the target program takes too much memory, it may cause the conversion to abort. As a result, part of the program is in source format and the rest is in target format and is therefore no longer usable. If you anticipate such a situation (i.e. in the case of low memory), you should first save the source program via the serial interface or print it out for emergencies.

7.8. Data File Management (RFILE)

With this function, data files are created, deleted, loaded into the TEXT editor or saved by the editor.

When the $\boxed{\text{R}}$ key is pressed while the main menu is displayed, the computer shows the data file menu (RFILE).

```
  << RAM DATA FILE >>

 Init  Save  Load  Kill
 Files

```

Enter the first letter of the function (I, S, K, L, or F) to select the corresponding function (File Create (Init), Save, Load, Delete (Kill) or View (Files)).

7.8.1. Create File (Init)

Pressing the $\boxed{\text{I}}$ key in the data file menu will prompt for the name of the file to be created.

Example:

To create the file TEST.
`TEST.DAT`

```
→Init  Save  Load  Kill
 Files

FILE NAME=TEST.DAT
```

```
→Init  Save  Load  Kill
 Files

FILE SIZE=?
```

The size of the file must be specified in bytes. The size must be chosen so that all the necessary data fits into the file.

Example:

Create a 1024 byte file.
`1024`

```
→Init  Save  Load  Kill
 Files

FILE NAME=1024
```

If the specified file has already been created, the computer asks whether the file should be reinitialized with the prompt **FILE INITIALZE OK? (Y)**.

Pressing the $\boxed{\text{Y}}$ key will reinitialize the file and all previous data is lost. Any other key aborts the re-initialization.

Note:

1. If a file extension is not included in the file name, the extension `.DAT` is automatically appended.
2. The file name can have a maximum length of 8 characters.
3. One file occupies the specified number of bytes in memory plus 34 additional bytes
4. If there is not enough space left in the memory to create the file, the computer displays **MEMORY OVER**

7.8.2. Load Data File (load)

When you press the $\boxed{\text{L}}$ key in the data file menu, the computer displays a list of stored files; where ""LOAD →" points to the first file name (if no files have been saved, the computer will not respond to the $\boxed{\text{L}}$ key).

Here is an example of a list of saved files.

```
LOAD →TEST     .DAT  1024
      ABC      .DAT   512
      SAMPLE   .DAT  2048
```

Use the ▲ and ▼ keys to move "LOAD →" to the name of the file to be loaded; then press ↵. The computer loads the contents of the selected file into the TEXT area and then returns to the program file menu.

7.8.3. Delete Data File (Kill)

This function deletes a specific file.

When you press $\boxed{\text{K}}$ in the data file menu, the computer asks for the name of the file to be deleted.

file menu (RFILE).

```
   << RAM DATA FILE >>

 Init  Save  Load  Kill
 Files

 FILE NAME=?
```

Enter the name of the file to be deleted and press ↵. The computer then asks for confirmation that the file should be deleted.

FILE DELETE OK? (Y)

Pressing $\boxed{\text{Y}}$ confirms the deletion. Any other key will cancel the deletion and the file menu will be displayed. If no file name extension is specified, the suffix .DAT is added by default.

If the specified file does not exist, the computer displays **FILE NOT FOUND**.

7.8.4. List Data Files (Files)

Pressing F in the program file menu will display a list of all stored files. → points to the first file name of the list. (If no files are stored, the computer does not respond to the F key.)

```
LOAD →TEST     .DAT  1024
      ABC      .DAT   512
      SAMPLE   .DAT  2048
```

The hidden parts of the list can be scrolled by pressing the ▲ and ▼ keys. To load a data file marked with the →, press **SHIFT** + M (or **2ndF** + M).

7.8.5. Save Data File (Save)

Pressing F while the data file menu displayed, prompts for the name of the file to be saved.

```
 Init →Save  Load  Kill
 Files

FILE NAME=?
```

Enter the file name of a previously created data file and press ↵. The computer the saves the file.

Example:

Saving a file with the filename "TEST".
`TEST`

```
 Init →Save  Load  Kill
 Files

FILE NAME=TEST.DAT
```

The computer then prompts **FILE OVERWRITE OK? (Y)**. Pressing Y saves the file "`TEST.DAT`" and then returns to the data file menu. Any other key aborts the SAVE function.

If the file was not previously created with `INIT`, the computer issues the error message **FILE NOT FOUND** and cancels the function.

Note:

- If the file to be saved is larger than the size specified in the Init function, the computer aborts the action and displays the error message **MEMORY OVER**.
- After selecting a function from the file menu, it is essential to enter a file name. If the ↵ key is pressed without entering a name, an **ILLEGAL FILE NAME** error will be displayed. To clear the error message, press **CLS**.
- A file name can consist of up to eight characters and an extension of up to three characters. If no extension is entered, the computer automatically assigns the extension "`.DAT`".
- If no data is stored in the TEXT area, saving cannot be performed.

8. THE C PROGRAMMING LANGUAGE

This chapter describes the differences between the C language on large computers (such as UNIX) and on the SHARP PC-G850. This chapter does not teach you how to program in C. Numerous books are available to learn how to program in C.

8.1. Properties of the C Programming Language

C is a very compact language. On one hand, C is a higher programming language, on the other hand, it is possible to use detailed processing notations that are very close to machine language.

While other higher programming languages (such as BASIC, FORTRAN, etc.) restrict access to the underlying hardware with PEEK and POKE, the C language makes it possible to write programs which directly access the hardware and memory, much like an assembly program. Programming in C can be compared to assembly language, however, it is much more efficient.

With its structured programming, C is easy to read and easy to understand. Thus, it is very powerful for program development. In addition, there are a variety of data types for processing data and numeric functions. Therefore, there is a wide range of applications for C, be it professional, private or scientific.

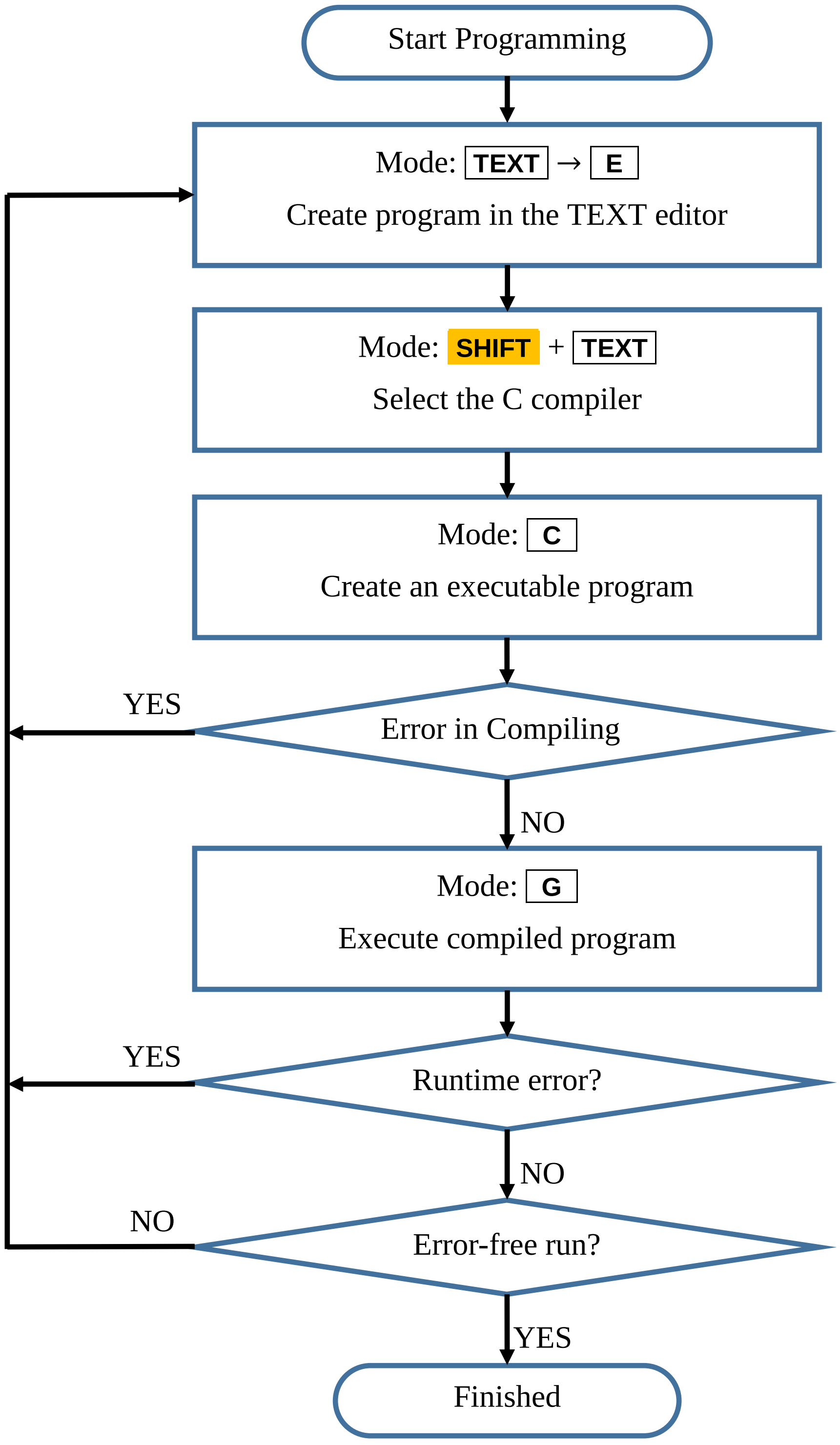
C programs are very compact. In addition, programs are very efficient due to the use of pointers.

Programs written in C are highly portable despite hardware-related programming. C programs can usually be run on other computer systems with few changes.

The C language is very powerful. This also has the disadvantage that programs can be written obscurely, as there are often several methods to solve a task.

8.2. The C Compiler

Since the computer does not understand the C statements directly, the C program must be compiled before execution. This usually requires the following steps:



8.2.1. Call the Text Editor:

TEXT → E

```
TEXT EDITOR
<
```

8.2.2. Enter the C Source Program:

```
10 main ()
20 {
30     printf ("Hello World ¥n");
40 }
```

Switch to **CAPS** mode to enter the commands. Similar to BASIC, each line must be preceded by a line number (without the following colon). By contrast, the C compiler doesn't do anything with the line numbers. They are internally ignored during compilation. They are used only for editing.

The editor functions can be found in the chapter on **TEXT** mode.

8.2.3. Compile the Source Program

Call the C compiler menu:
SHIFT → TEXT

```
        *** C ***

 Compile Trace Go Stdout
```

The following commands are available:

Compile	: Compile the program in the TEXT Editor
Trace	: Run the Program in Trace Mode (Step by Step)
Go	: Run the Program
Stdout	: Switch the standard output to the printer

The respective command is selected by entering the first letter.

8.2.4. Compile

Press the C key. The message "compiling" appears briefly. If the program compiles properly, the message "complete!" appears shortly afterwards

```
        *** C ***

 Compile Trace Go Stdout

complete!
```

If an error message appears instead, the program must be corrected with the editor (**TEXT** → **E**) and recompiled.

If the error **MEMORY FULL** appears, there is insufficient free memory to create the executable program.

8.2.5. Running the Program

Execute the program by pressing the key **G** in the C menu.

```
Hello World
*EXIT (40)
```

"• EXIT" indicates the number of the line where the program finished execution. To return to C menu, press the **CLS** or **BREAK** key.

Below are the descriptions of the runtime errors.

8.3. Trace

In order to locate errors in a program, it may be helpful to execute the program step by step and observe what the program does in detail and examine the contents of the variables. The `TRACE` function can be used in the C compiler menu for this purpose. The trace function is explained using this example.

```
10 main ()
20 {
30 int i, gokei = 0;
40 for (i = 1; i <51; i ++) {
50    gokei += i;
60    printf (" 1 + ... +% d =% d¥n", i, gokei);
70    }
80 }
```

8.3.1. Start TRACE Mode

Trace mode is started by pressing the **T** button in the C menu.

```
?10 main()
```

Each command is shown in the display and executed by pressing ↵. Each subsequent command is run by pressing ↵. Press the **BREAK** button to enter pause mode.

Functions in Pause Mode:

↵	: Exit the pause mode and continue the program.
C	: Exit the pause mode and continue the program.
A	: Cancel the trace mode and return to the compiler menu.
T	: Continue Trace
N	: Resume normal compilation of the program (without further tracing).
D	: Enter variable mode

In variable mode, enter the name of the variable to display its contents.

BREAK

D i ↵

```
1+...+1 = 1
 40   for(i-1;i<51;i++)  {
Break>D
var>i
int : 2(0x0002)
var>_
```

In this example, the variable `i` currently has the value 2. Pressing BREAK will exit variable mode

8.4. Redirecting Screen Output to the Printer

If the CE-126P printer (sold separately) is connected and ready to use, press S on the C compiler menu screen. This will change the display from stdout (screen output) to stdprn (printer output).

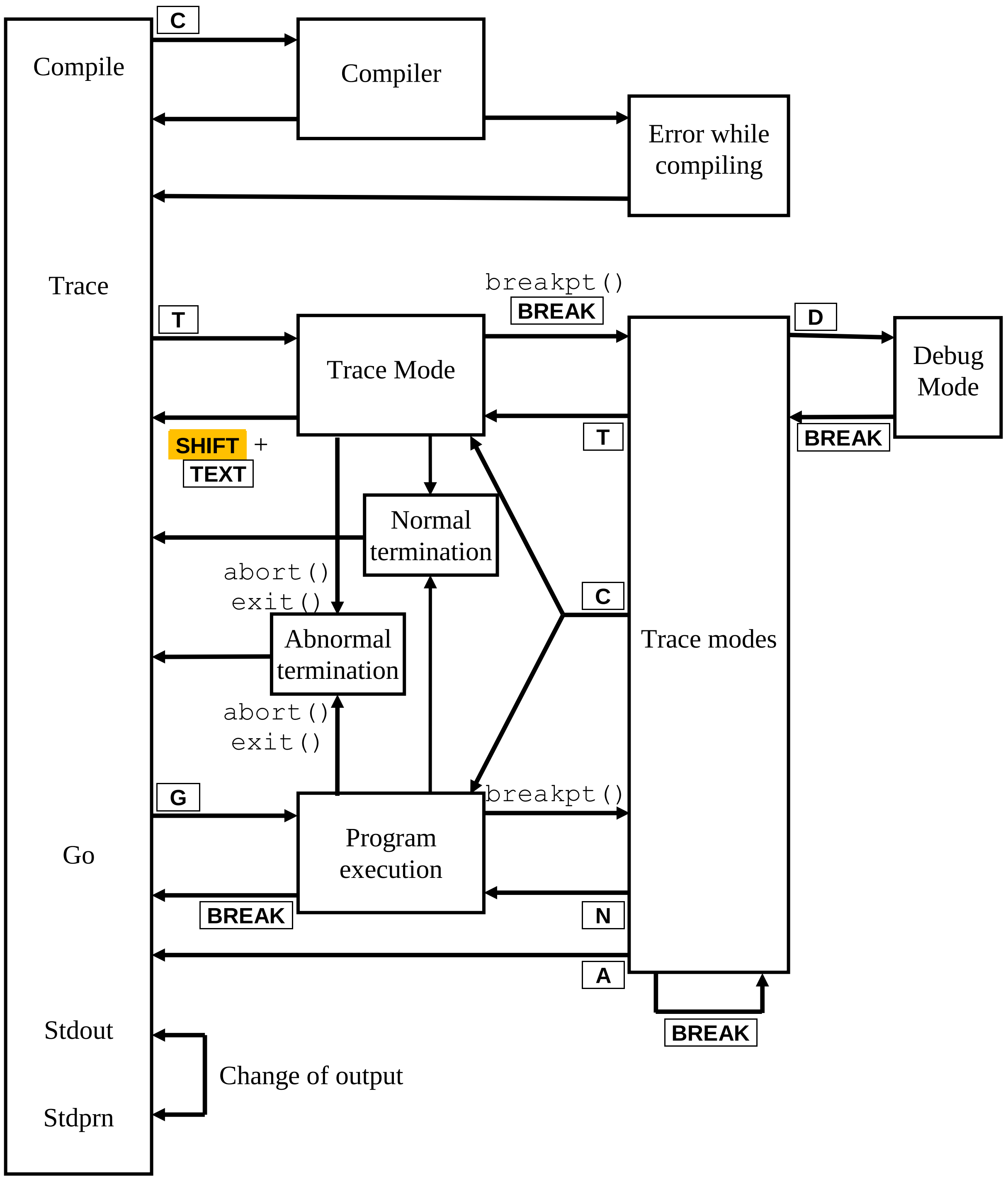
Press the S key again to switch to screen output. If the program explicitly uses stdprn, the output will be directed to the printer, regardless of what is specified in the compiler menu.

`stdout`: output on the screen
`stdprn`: output on the printer

The following C commands depend on the setting in this menu:

```
putc
fputc
fputs
fprintf
```

8.5. Functional Diagram of the C Compiler



8.6. C Programming Basics

This section only deals with the specific features of the C compiler in the SHARP PC-G850.

8.6.1. Formatting Options for Output (i.e. printf)

command	output
%d	Integer decimal number
%x	Integer hexadecimal number
%o	Integer octal number
%f	Floating point
%s	String
%c	Single sign

8.6.2. Variable Types

Type	Subtype	Range	Size
Integer	char	-128 to +127	8-bit
	unsigned char	0 to 255	8-bit
	int	-32768 to +32767	16-bit
	unsigned int	0 to 65535	16-bit
	short	-32768 to +32767	16-bit
	unsigned short	0 to 65535	16-bit
	long	-2147483648 to +2147483647	32-bit
	Unsigned long	0 to 4294967295	32-bit
Real	float	$\pm$ 1x10^{-99} to $\pm$ 9,999x10^{+99}	32-bit
	Double	$\pm$ 1 x 10^{-99} to $\pm$ 9.999999999 x 10^{+99}	64-bit
	long double	$\pm$ 1 x 10^{-99} to $\pm$ 9.999999999 x 10^{+99}	64-bit

Unsigned: Unsigned works without a sign. Thus, the full number of bits is available for the number.

Variable Names

Variable names may consist of lower case, capital letters and numbers (no Kana characters) and must always begin with a letter. Special characters are not allowed.

A variable name has a maximum length of 31 characters. Extra characters are ignored.

A variable cannot have the same name as a keyword.

8.6.3. Operators

Comparison Operators

Expression	Comparison
`a==b`	True if a equals b
`a!=b`	True if a is not equal to b
`a<b`	True if a is less than b
`a>b`	True if a is larger than b
`a<=b`	True if a is less than or equal to b
`a>=b`	True if a is greater than or equal to b

Arithmetic Operators

Operator	Operation	Example
`+`	addition	`a + b`
`-`	subtraction	`a - b`
`*`	multiplication	`a * b`
`/`	division	`a / b`
`%`	modulo	`a % b`

Assignment Operators

Operator	Example	Definition	Arithmetic operation
=	a = b	Replace a with b	
+=	a += b	Add a to the contents of b	a = a + b
–=	a –= b	Subtract b from the contents of a	a = a – b
*=	a *= b	Multiply a by the contents of b	a = a * b
/=	a /= b	Divide a by the content of b	a = a / b
%=	a %= b	a is the remainder of the division of a by b	a = a % b

Increment / Decrement Operators

Operator	Example	Definition	Arithmetic operation
`++`	`++a`	Increment a by 1 then use	a = a + 1
`++`	`a++`	Use a then increment by 1	
`--`	`--a`	Decrement a by 1 then use	a = a – 1
`--`	`a--`	Use a then decrement by 1	

Logical Operators

operator	Example	Definition
`&&`	`a&&b`	Logical AND of a and b (1 if neither a nor b is 0)
`||`	`a||b`	Logical OR of a and b (1 if neither a nor b is 0)
`!`	`!a`	Logical NOT (if a <> 0, then 0, if a = 0 then 1)

Bitwise Operators

Operator	Example	Definition
&	`a&b`	Bitwise AND
|	`a|b`	Bitwise OR
^	`a^b`	Bitwise XOR
~	`~a`	Bitwise NOT

Shift Operators

Operator	Example	Definition
<<	`a<<b`	shift a one bit to the left b times
>>	`a>>b`	Shift a one bit to the right b times

Keywords

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Escape Control Characters

control character	Hex value	description
¥b	0x08	Backspace
¥n	0x0A	Newline
¥r	0x0D	Go to the beginning of the line
¥t	0x09	Tab (jump to the next tab stop)
¥¥	0x5C	The character ¥
¥'	0x2C	The character '
¥"	0x22	The character "
¥?	0x3F	The character ?
¥ddd		The characters as a 3-digit octal number
¥xhh		The characters as a hexadecimal number.

8.7. C SYNTAX

8.7.1. Compound Instructions

Statements surrounded by curly brackets are treated by the computer as a group or as a standalone statement. The only difference is that there is no semicolon behind the closing bracket at the end of the compound statement

```
{
   Instruction 1
   Instruction 2
       ⋮
   Instruction n
}
```

8.7.2. Conditional Jumps

If…else

Format:

`1) if (expression)`
`     statement`
	If the *expression* is true, the *statement* will be executed.

`2) if (expression)`
`     statement1`
	If the *expression* is true, *statement1* is executed,

`   else`
`     statement2`
	otherwise, *statement2* is executed

`3) if (expression1)`
`     statement1`
	If *expression1* is true, execute *statement1*.

`   else if (expression2)`
`     statement2`
	If *expression1* is false and *expression2* is true, execute *statement2*,

`   else`
`     statement3`
	otherwise, *statement3* is executed.

switch…case

Format:

```
switch (expression) {
   case const-expression1: statement1
                  [break;]
   case const-expression2: statement2
                  [break;]
                     ⋮
   case const-expression#: statement#
                  [break;]
   default: statement
}
```

8.7.3. Loops

for

Format:
```
for (expression1, expression2, expression3)
    statement
```

expression1: initialize loop.

expression2: after executing *expression1*, *expression2* is checked and if true, the statement is executed.

expression3: execute *expression3* with every iteration of the loop. Execution continues until *expression2* is false.

while

Format:
```
while (expression)
    statement
```

statement is repeated as long as *expression* is true.

do-while

Format:
```
do
   statement
while (expression)
```

statement is executed and then *expression* is checked. If true, the *statement* is executed again.

8.7.4. Unconditional Jumps

goto

Format:
```
goto label
  ⋮
label: instruction
```

The `goto` statement continues program execution at the specified *label*.

continue

The continue statement aborts the current loop and starts the next iteration of a while, do-while, or for loop.

```
for (i = 0; i<100; i++) {
        ⋮
    if (i%2 == 0)
       continue;
    printf (“%d\n"NN, i);
}
```

break

The break statement immediately aborts the next outer switch, while, do-while, or for statement.

```
for (i = 0; i <100; i ++) {
      ⋮
  if (a[i] <0)
    break;
      ⋮
}
```

return

Return returns to the calling program. A return value can be passed (not null).

Example:

```
return (expression);
```

or

```
return;
```

8.8. Storage Classes

Storage classes are used to define storage areas for variables and to define the extent (area from which the program can read / write).

storage class	range of validity
auto	short-term storage within a program
register	For frequent access. Variables for increasing the execution speed by assigning values to a register (otherwise like auto).
static	Reserves an area during program execution. Value access and corresponding actions throughout the program.
external	File-external or function-external global variables.

8.9. Arrays

The C compiler supports the use of up to eight-dimensional arrays. Example for a two-dimensional array:

`char color[3][6]` (3 rows by 6 columns of characters)

The same statement with assignment of values:

```
char color[3][6] = {"white", "red", "blue"};
```

8.10.Structures

Structures defines a new data type which unites components of different types. With the help of `struct`, data structures can be generated.

The type declaration

```
struct struct-identifier {
  data-declaration
};
```

allows the declaration of variables of this type

```
struct-identifier var-identifier;
```

Example: Declaration of a data type for storing the personal data of a student.

```
// Structure
{
// new structure
struct Student {
  long long int ID;
  int skz;
  char surname[30], first name[20];
};


// create variables of type Student
Student arni, robbi;
// Data input
cout << endl << "First Name:";
cin >> arni.firstname;
         ⋮
robbi = arni;     // copy record
cout << robbi.firstname << endl;
}
```

The assignment `robbi = arni;` copies the entire record from one variable to another. Access to the component first name of the variable arni (of type Student) is made via

```
arni.name
```

The data are stored in the form

```
ID   skz  surname   first name
```

8.11.Compiler Runtime Options

#include “file”

Inserts the contents of the file at the appropriate place in the source file before compilation. Similarly, certain parts of the source code can be included or ignored during compilation, depending on the results of conditional testing.

For example, with the `#include` command, header files can be inserted. This is usually not necessary with this computer.

#define name [value]

This defines symbols, constants or macros for the program (for example, to perform tests, see `#ifdef` and `#ifndef`).

Examples:

```
#define TEST
#define PI 3.141592
#define NULL 0
#define EOF -1
#define FILE int
```

Macros can also be defined:

```
#define SQR(x)((x)*(x))
```

#if ... #elif ... #else ... #endif

With `#if`, similar to `#ifdef`, a conditional expression can be initiated. Constant expressions can be evaluated as well.

```
#if expression1
    statement1
  [#elif expression2
    statement2]
  [#else expression3
    statement3]
#endif
```

#ifdef name ... #endif

The `#ifdef` command can be used to check if a symbol has been defined. If the symbol is defined, the code following the command will be passed to the compiler. An `#ifdef` directive must be completed by a `#endif` directive.

#ifndef name ... #endif

The `#ifndef` command is the counterpart to `#ifdef`. It checks if a symbol is not defined. If the symbol is defined, the code following the command will not be passed to the compiler. An `#ifndef` command must be terminated by an `#endif`.

8.12.Library Functions

This section explains the library functions of the C compiler. In this computer, the standard input and output devices (stream) are defined as follows:

Input	`stream: stdin`	Keyboard
Output	`stream: stdout (or stdprn)`	Screen (or printer)
Serial	`stream: stdaux`	Half duplex over 11-pin
	`stream: stdaux1`	Full duplex over 11-pin

In addition, the following constants are defined by default:

```
#define NULL 0
#define EOF -1
#define FILE int
```

When redirecting to the printer with **SHIFT** (or **2ndF**) + ↵ (**P↔NP**), the input functions return the following:

`getch:`	`0xFF`
all other input functions:	`EOF`

The delimiters when the input is through the serial interface are:

Row separator:	`0x0d, 0x0a or 0x0d + 0x0a`
End of file:	`0x1a`

(The input from `0x0d + 0x0a` is converted to `0x0a`) You should normally use `0x0a` as a row separator.

The delimiters when the output is via the serial interface are:

Separator: `null`

(The output of `0x0a` (row separation) is converted to `0x0d + 0x0a`)

8.12.1.Standard I/O Functions

getc, getchar, fgetc

Format:

```
int getc (FILE* stream);
int getchar (void);
int fgetc (FILE* stream);
```

Description: A character is read. If read by `stdin`, the character is not transmitted until ↵ is pressed.

`getc:`	Reads a character from the given stream.
`getchar:`	Reads a character from stdin.
`fgetc:`	Reads a character from the specified stream.

Return Value: the read character

gets, fgets

Format:

```
char* gets (char* s);
char* fgets (char* s, int n, FILE* stream);
```

Description: Characters are read and stored in the string s.

`gets`: Reads characters from `stdin` to ↵. Before saving the string, carriage returns / line feeds will be replaced by `0x00` (`¥0`).

`fgets`: Reads characters from the specified stream. The characters start from the current position of the data stream to the first carriage return / line feed character encountered, end of the file (`EOF`), or until the number of read characters equals n-1. A `null` character is appended to the end of the passed string.

Return value: Zero is returned when the end of the file (`EOF`) is reached.

scanf, fscanf, sscanf

Format:

```
int scanf (const char* format [, address,…]);
int fscanf (FILE* stream, const char* format [,
  address,…]);
int sscanf (char* s, const char* format [,
  address,…]);
```

Description: The family of `scanf()` functions checks the input for a format as described below. This format may contain conversion specifications. The results of such conversions, if any, are stored at the locations pointed to by the pointer arguments that adhere to the format.

Each pointer argument must have an appropriate type for the return value by the associated conversion specification. If the number of conversion specifications in format exceeds the number of pointer arguments, the results are undefined. If the number of pointer arguments exceeds the number of conversion specifications, then the excess pointer arguments are evaluated but otherwise ignored.

`scanf`: Reads characters from `stdin` to ↵.
`fscanf`: Reads characters from input to carriage return / line feed.
`sscanf`: The characters are read from the specified string *`s`*.

Return value: Number of assigned arguments. `EOF` will be returned when the end of the file is reached.

Format definition
The string format consists of a series of guidelines which describe how the sequence of input characters is processed. If processing a directive fails, no further input is read and `scanf()` returns.

1. Space / Carriage Return
   The entry is read or, if no more characters are present, is read until it encounters a character that is not a space (will not be read anyway). Execution of the function will be terminated if a character is encountered that is not a space.
2. Normal character (other than space and %)
   The next character is read. The execution of the function is terminated if it is not a normal character, and the input character is not read.

Conversion Definitions

Symbol	Expected format	Conversion
%d	String in binary integer format (decimal)	int
%i	String in binary integer format (decimal, octal or hexadecimal)	int
%O	String in binary integer format (octal)	int
%u	String with a whole decimal number without sign.	unsigned int
%x	String with a whole hexadecimal number	int
%f	String with a floating-point number	float
%e	String with a floating-point number	float
%G	String with a floating-point number	float
%c	String with a string (character number 1 or specified field width)	char
%s	String (at the end zero (¥0) is added)	
%p	String of 4 hex characters (e.g., 89ab)	pointer

Format of the conversion statement
`%[*] [`*`fieldwidth`*`] [I] Symbol`

`*`:	(Assignment prevented) It is possible to read into the input field, but it is not possible to assign the conversion result to an argument.
`fieldwidth`:	The maximum field width is defined by an unsigned integer.
`I`:	Integer numbers are converted to long integer. Floating point numbers are converted to double. Long floating point numbers are converted to long double.

putc, putchar, fputc

Format:

```
int putc (int c, FILE* stream);
int putchar (int c);
int fputc (int c, FILE* stream);
```

Description: Outputs a single character.

`putc`:	A character is written to the specified stream.
`putchar`:	A character is written to `stdout`.
`fputc`:	A character is written to the specified stream

Return value: the written character. If an error occurs while writing `EOF` is returned.

puts, fputs

Format:

```
int puts (const char* s);
int fputs (const char* s, int n, FILE* stream);
```

Description: Characters are written from the string `s`.

`puts`:	Writes a string to `stdout`. The end of the string null character is replaced by carriage return/line feed.
`Fputs`:	Writes a string to the specified stream starting at the current position of the output flow. The end of the string null character is not written.

Return value: the last written character. If an error occurs while writing `EOF` is returned.

printf, fprintf, sprintf

Format:

```
int printf (const char* format [, arg,…]);
int fprintf (FILE* stream, const char* format [,
  arg,…]);
int sprintf (char* s, const char* format [, arg,
  …]);
```

Description: The family of `printf()` functions converts the “argument” of the format definition and outputs it to a stream, writes it to `stdout`, or returns the result as a string. The format string is a character string of length greater than 0 and can be composed of normal characters, ESC sequences, and conversion definitions. Normal characters and ESC sequences are output in order of appearance. Conversion definitions, on the other hand, are carried out by sequential extraction, conversion, and output of the arguments. If there are more arguments than definitions, the additional arguments are ignored. If there are too few arguments, the results are undefined.

`printf`:	Write characters to stdout.
`fprintf`:	Write characters to the specified stream from the current position.
`sprintf`:	Write characters to the specified string `s`.

Return value: Number of characters output. `EOF` will be returned if an error occurs.

Format definition
The string format consists of a series of guidelines which describe how the sequence of output characters is generated.

Symbol	Expected format	Argument
%d	Display as signed decimal	int
%i	Display as signed decimal	int
%O	Display as unsigned octal	int
%u	Display as unsigned decimal	int
%x	Display as unsigned hexadecimal (abcdef)	int
%X	Display as unsigned hexadecimal (ABCDEF)	int
%f	Display decimal in the form [-] ddd.ddd, where ddd is a single-digit decimal value or longer.	double
%e	Display decimal in the form [-] d.ddde ± dd, where d is a one-digit decimal value, ddd is one or more digits.	double
%E	Display decimal in the form [-] d.dddE ± dd, where d is a one-digit decimal value, ddd is one or more digits.	double
%g	Converts f or e in a shortened form to Double %G	double
%G	Converts f or E in a shortened form to Double %c	double
%c	Conversion to an unsigned character	int
%s	Characters of the string are output until zero (¥0, is not output) or the specified number of characters is reached.	string (char*)
%p	Output as a pointer argument	pointer

Format of the conversion statement
`%[`*`flag`*`] [`*`fieldwidth`*`] [.`*`precision`*`] [`*`I`*`] Symbol`

`flag`

1.	: left-justified output
+	: sign is always output
#	: for a % conversion, a 0 is prefixed. For a %x and %X conversion, a 0x (or 0X) is prefixed.
0	: fill result with leading zeros (for %d, %i, %O, %u, %x, %X)
(omitted)	: right-justified output

`fieldwidth`

n	: specifies the number of digits to be output. Spaces are used if there are fewer characters than spaces.
0n	: set field to length n. If the result of the conversion is shorter than n, the result is padded with zeroes.
(omitted)	: length is defined by the conversion result.

`.`*`precision`*

n	:%d, %O, %u, %x, %f - defines the smallest number of digits to output (default is 1). %e, %E, %f - defines number of places after the decimal point (default is 6). %g, %G - defines number of characters to output (default: all significant characters).
`I`	:Defines the output as a long argument for %d, %i, %o, %u, %x, %X

fflush

Format: `int fflush (FILE*` *`stream`*`);`

Description: Writes the contents of the buffer to a file in the output stream. For an input stream, the contents of the buffer memory are deleted. This feature does not close the stream. The buffer is automatically flushed when it is full.

Return value: `null`. If an error occurs while writing, `EOF` is returned.

clearerr

Format: `void clearerr (FILE*` *`stream`*`);`

Description: This function clears a data stream `EOF` error condition

8.12.2.Character Functions

isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit

Format:

```
int isalnum (int c);
int isalpha (int c);
int iscntrl (int c);
int isdigit (int c);
int isgraph (int c);
int islower (int c);
int isprint (int c);
int ispunct (int c);
int isspace (int c);
int isupper (int c);
int isxdigit (int c);
```

Description: Characterization of a character

`isalnum`	:check for letters or digits
`isalpha`	:check for letters
`iscntrl`	:check for control characters
`isdigit`	:check for digits
`isgraph`	:check for any printable character except space
`islower`	:check for lowercase letters
`isprint`	:check for any printable characters, including space
`ispunct`	:check for any printable character that is not a space or an alphanumeric character
`isspace`	:check for dial tone (spaces, tabs, line breaks and so on, `0x09` ~ `0x0d`, `0x20`)
`isupper`	:check for capital letters
`isxdigit`	:check for hexadecimal character (`0-F`, `0-f`)

Return value: true (value not equal to zero) or false (zero)

tolower, toupper

Format:

```
int tolower (int c);
int toupper (int c);
```

Description:

`tolower`	: converts the character to lowercase
`toupper`	: converts the character to capital letters

Return value: the converted character

8.12.3.String Functions

strcat

Format: `char* strcat (char* s1, const char* s2);`

Description: Append string *s2* to string *s1*.

Return value: Pointer to string `s1`

strchr

Format: `char* strchr (const char* s, int c);`

Description: Searches a string for the first occurrence of a particular character.

Return value: returns a pointer to the first occurrence of the character `c` in the string `s` or `null` if the string does not contain this character.

strcmp

Format: `int strcmp (const char* s1, const char* s2);`

Description: Compares two strings. Starting with the first character, the two strings are compared character-by-character until two corresponding characters are unequal or the end of the strings are reached.

Return value: returns a value

<0	if *s1* is less than *s2*
0	if *s1* is equal to *s2*
0>	if *s1* is greater than *s2*

strcpy

Format: `char* strcpy (char* s1, const char* s2);`

Description: Copies one string to another. `strcpy` copies the contents of string *s2* to string *s1*. The final `null` character of `s2` is copied as the last character.

Return value: Pointer to string *s1*

strlen

Format: `int strlen (const char* string);`

Description: Determines the length of a string

Return value: Returns the number of characters in the string. The final `null` character is not counted.

8.12.4.Memory Functions

calloc

Format: `void* calloc (unsigned n, unsigned size);`

Description: Allocates memory to a group of `n` elements, where each element is size *`size`*.

Return value: Returns a pointer to the reserved memory area. If memory allocation fails (for example, if the RAM size is exceeded), `NULL` is returned.

malloc

Format: `void* malloc (unsigned size);`

Description: Allocates memory of the size `size`.

Return value: Returns a pointer to the reserved memory area. If memory allocation fails (for example, if the RAM size is exceeded), `NULL` is returned.

free

Format: `void* free (void* ptr);`

Description: Releases allocated memory reserved by `calloc` or `malloc`. The argument `ptr` must be a pointer to a memory area previously allocated by `calloc` or `malloc`.

Return value: none

8.12.5.Mathematical Functions

abs

Format: `int abs (int x);`

Description: The absolute value of integer `x`

Return value: absolute number (0-32767).

asin, acos, atan

Format:

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

Description: These functions return a value equal to their respective inverse trigonometric function. It can be specified in **DEG**, **RAD** or **GRAD**. The calculation range for asin and acos is between -1 and 1

Return value: the respective value of the result, NULL in case of an error.

	Value Range		
Function	DEG	RAD	GRAD
asin	-90° to 90°	-π/2 to π/2	-100 to 100
acos	0° to 180°	0 to π	0 to 200
atan	-90° to 90°	-π/2 to π/2	-100 to 200

asinh, acosh, atanh

Format:

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

Description: These functions return a value equal to their respective inverse hyperbolic function.

Return value: the respective value of the result.

exp

Format:

```
double exp (double x);
```

Description: Calculates e^x

Return value: the expected value of the result

log, log10

Format:

```
double log (double x);
double log10 (double x);
```

Description: `log(x)` calculates the natural logarithm of `x`.
`log10(x)` calculates the common logarithm of `x`.

Return value: the result.

pow

Format:

```
double pow (double x, double y);
```

Description: Raise *x* to the power of *y*.

Return value: the result.

sin, cos, tan

Format:
```
double sin (double x);
double cos (double x);
double tan (double x);
```

Description: These functions return a value equal to their respective trigonometric function. The indication can be made in **DEG**, **RAD** or **GRAD**.

Return value: the respective value for the result.

sinh, cosh, tanh

Format:
```
double sinh (double x);
double cosh (double x);
double tanh (double x);
```

Description: These functions return a value equal to their respective hyperbolic function.

Return value: the respective value for the result.

sqrt

Format:
```
double sqrt (double x);
```

Description: Calculates the square root of `x`

Return value: the result.

8.13.Hardware Interface Functions

This section describes the hardware-specific I/O functions.

8.13.1.Mini I/O Functions

miniget

Format:
```
int miniget (void);
```

Description: Reads a byte from the mini I/O port
Bit 2: Xin, Bit 1: Din, Bit: Ack

Return value: the byte read

miniput

Format:
```
void miniput (char byte);
```

Description: Writes a byte to the mini I/O port
Bit 2: Busy, Bit 1: Dout, Bit0: Xout

Return value: None

8.13.2.8-bit PIO Control via the 11-pin Interface

fclose

Format: `int fclose (FILE* stream);`

Description: close the stream

Return value: if successful, `NULL` is returned. In case of an error EOF will be returned.

fopen

Format: `FILE* fopen (char* path, char* type);`

Description: Opens a stream to the device specified by path with the mode specified by type. For 8-bit PIO, path is "`pio`" and the mode can be set as follows:
r+ : input
w+ : output
a+ : input and output.

Return value: with normal execution, the pointer to the FILE structure is returned. In case of error, `NULL` is returned.

pioget

Format: `int pioget (void);`

Description: reads in a byte from the PIO port.

Return value: the byte read.

pioput

Format: `void pioput (char byte);`

Description: writes a byte to the PIO port.

Return value: none

pioset

Format: `void pioset (char byte);`

Description: set the input and output mode of the PIO port. 1 sets the input mode and 0 sets the output mode

Return value: none

8.13.3.SIO (RS-232C) Control via the 11-pin Interface

fclose

Format: `int fclose (FILE* stream);`

Description: Close the stream. In the output mode, the `EOF` character is written, which was determined by the entry in **TEXT** mode under SIO with the end-of-file parameter.

Return value: if successful, `NULL` is returned. In case of an error, `EOF` will be returned

fopen

Format: `FILE* fopen (char* path, char* type);`

Description: Opens a stream to the device specified by path with the mode specified under type. For the serial interface, path is `stdaux` for half-duplex communication and `stdaux1` for full-duplex communication. The mode can be specified as follows:

r+	: input
w+	: output
a+	: input and output.

Return value: with normal execution, the pointer to the `FILE` structure is returned. In case of error, `NULL` is returned.

8.13.4.Buffer / Communications Controller

feof

Format: `int feof (FILE* stream);`

Description: Checks if the stream has reached the end of the file (`EOF`).

Return value: When the end of the file is reached, the value of -1 is returned. If the end of the file has not yet been reached, `NULL` is returned.

8.13.5.I/O port functions

inport

Format: `unsigned char inport(unsigned charport);`

Description: Reads a byte from the specified I/O port address (`0x20-0x3F`)

Return value: the read byte

outport

Format: `void outport(unsigned` *`charport`*`, unsigned char` *`byte`*`);`

Description: Writes a byte to the specified I/O port address (`0x20-0x3F`)

Return value: none

8.13.6.Memory Functions / Program Call

call

Format: `unsigned call(unsigned` *`adr`*`, void*` *`arg_HL`*`);`

Description: Calls a machine language program starting from the address *`adr`*. The value of the *`arg_HL`* argument is passed to the HL register.

Return value: Contents of the HL register

peek

Format: `unsigned char peek(unsigned` *`adr`*`);`

Description: Reads a byte from the address `adr`.

Return value: Content of memory address `adr`

poke

Format: `void poke(unsigned` *`adr`*`, unsigned char` *`byte`*`);`

Description: Writes a byte to the address *`adr`*.

Return value: none

8.13.7.Datafile Functions

fclose

Format: `int fclose(FILE*` *`stream`*`);`

Description: Close the file stream. If file mode “w” or “a” was specified, a `0x1A` is written on closing the file.

Return value: if successful, `NULL` is returned. In case of an error, `EOF` will be returned

feof

Format: `int feof(FILE*` *`stream`*`);`

Description: Checks if the stream has reached the end of the file (`EOF`).

Return value: When the end of the file is reached, a value of 1 is returned. If the end of the file has not yet been reached, a 0 is returned.

flof

Format: `unsigned long flof(FILE* stream);`

Description: Determines the number of remaining unused bytes in the file.

Return value: the number of remaining unused bytes.

fopen

Format: `FILE* fopen(char* path, char* type);`

Description: Opens a stream to the file specified by path with the mode specified under type. The path definition for a data file corresponding to the file name is defined in **TEXT** mode. The mode is specified as follows:

r	: input
w	: output
a	: input and output.

Return value: with normal execution, the pointer to the `FILE` structure is returned. In case of error, `NULL` is returned.

8.13.8.Graphic Functions

The graphic functions described here correspond to the BASIC functions. For more detailed information, please use the descriptions of the corresponding BASIC commands.

circle

Format:

```
int circle (int x, int y, int r, double s-angle,
  double e-angle, double ratio, int reverse,
  unsigned short fill);
```

Description: draws a circle

`x, y`	: coordinate of the center
`r`	: radius
`s-angle`	: start angle
`e-angle`	: end angle
`ratio`	: ratio for the ellipse
`reverse`	: 0-set point
	: 1-delete point
	: 2-invert point
`fill`	: fill pattern (See the description of the BASIC command)

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

gcursor

Format: `int gcursor (int x, int y);`

Description: Positions the graphic cursor at point x, y.

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

gprint

Format: `void gprint (char* image);`

Description: Draws graphic pattern on the display.

Return value: none

line

Format: `int line (int x, int y, int x2, int y2, int reverse, unsigned short mask, int rectangle);`

Description: Draws a line or a rectangle.

`x,y`	:coordinates of first point (corner)
`x2,y2`	:coordinates of second point (opposite corner)
`reverse`	: 0-Set point
	: 1-Delete point
	: 2-Invert point
`mask`	: Line style (See the description of the Basic command):
`rectangle`	: 0-draws a line
	: 1-draws a rectangle
	: 2-draws a filled rectangle

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

paint

Format: `int paint (int x, int y, unsigned short kind);`

Description: Fills the area with the pattern starting at the coordinate `x,y`.
`kind`: fill pattern (See the description of the BASIC command)

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

point

Format: `int point (int x, int y);`

Description: Provides information about the state of the display point at `x,y`.

Return value: If the point is dark, i.e. set, then 1 is returned. If the item is not set, the value 0 is returned. If the point is outside the screen, -1 is returned.

preset

Format: `int preset (int x, int y);`

Description: Clears the display point.

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

pset

Format: `int pset (int x, int y, int reverse);`

Description: Sets the display point.
`reverse`: 0 sets the point
1 inverts the point

Return value: if successful, `NULL` is returned. In case of an error, -1 is returned

8.14. Other Functions

abort, exit

Format:
```
void abort (void);
void exit (int status);
```

Description: Exits / terminates the program.
`abort` : aborts the program. `A B O R T` will be displayed on screen.
`exit` : normal program termination with return code

Return value: none

angle

Format: `void angle (unsigned n);`

Description: Sets the mode for the trigonometric functions.
`n` = 0 : DEG
`n` = 1 : RAD
`n` = 2 : GRAD

Return value: none

breakpt

Format: `void breakpt (void);`

Description: Interrupts program execution and enters **BREAK** mode.

Return value: none

clrscr

Format: `void clrscr (void);`

Description: Clears the screen

Return value: none

getch

Format: `int getch (void);`

Description: Waits for a character from the keyboard. Does not require ↵.

Return value: returns the read character

gotoxy

Format: `void gotoxy (unsigned x, unsigned y);`

Description: Sets the text cursor to the specified coordinate on the screen. (0,0) is the upper left corner.

Return value: none

kbhit

Format: `int kbhit (void);`

Description: Reads a character from the keyboard without waiting.

Return value: returns the read key. If no key is pressed, 0 is returned.

8.15.Error Messages

8.15.1.Compiler Error Messages

Error message	Description
zero dimension	An array size of NULL in a context where this is illegal.
array of function is illegal	Array of function is not allowed
cannot find include file	Include file cannot be found
case in not switch	case statement is not inside a switch statement
constant expected	- Element count of the array is not an integer - Term of a case label is not a constant expression
default not in switch	Default statement is not within a switch statement
define buffer full	Too many #define statements
different s / u	Different struct/union
division by 0	Division by 0
duplicate #define <name>	Macro double defined
duplicate case	More than one case statement in the switch statement.
duplicate default	More than one default statement in the switch statement.
duplicate label: <name>	Label <name> was defined more than once.
empty character constant	Constant has no content
float overflow	Float point constant outside of range
float underflow	Float point constant outside of range
function illegally s / u	Function in struct/union area not allowed
function returns illegal type	Type of return value is not allowed in this function
if nest too deep	Too many nested #if/#ifdef statements
if nesting error	#if/#ifdef/#endif syntax error
if-less elif	Matching #if/#ifdef to #elif not found

Error message	Description
if-less else	Matching #if/#ifdef to #else not found
if-less endif	Matching #if / #ifdef to #endif not found
illegal #line	Wrong #define syntax
illegal break	Break statement within do, for, while, or switch loop not allowed
illegal character	Illegal character in source code
illegal class	Defined class cannot be used
illegal continue	Continue statement within do, for, or while loop not allowed
illegal digit in octal	Illegal digit in an octal number (8 or 9).
illegal function	Call function that does not match the type
illegal if	Incorrect expression in #if/#ifdef
illegal include	Incorrect #include statement syntax
illegal indirection	Invalid operand for unary operator *
illegal initialization	The right side of the initialization is not a constant expression
illegal main	Argument declared in the function main()
illegal operand of <operator>	Operand of <operator> is wrong type
illegal operand of U+	Operand of unary + operator is wrong type
illegal operand of U-	Operand of unary - operator is wrong type
illegal operand of ARG	Function arguments are wrong type
illegal operand of RET	Expression of return statement is wrong type
illegal s / u	struct / union used incorrectly.
illegal size	Size of the structure / union is too large
illegal switch expression	Invalid expression in switch command
illegal type	Invalid type cast has occurred
illegal void	Use of type void is incorrect
include nest too deep	#include nesting is too deep
macro recursion	Macro is recursive
memory full	Memory is full
missing argument: <name>	No argument <name> in the function call
missing declarator	No declaration
missing function: <name>	Function <name> was not declared
missing label	-No label in goto statement -Undefined label in goto statement
missing main	Main () not defined
missing member	-undefined members in struct / union -unused members in struct/union.
missing member in s / u	Missing member in struct/union.
missing name in prototype	No argument name in prototype definition
missing type	Type not defined.
missing type in prototype	Syntax error in prototype
newline in character constant	Line break in character constant.

Error message	Description
newline in string constant	Line break in string constant
prototype mismatch	Expression of the function call does not match the prototype
redeclaration: <name>	<name> was already defined
reserved: <name>	<name> is reserved
syntax error	Program violated the syntax rules
token buffer full	Macro expansion is too complex
too complicated declarator	Definition is too complex
too complicated declaration	Definition is too complex
too complicated initialize	Initialization is too complex
too deep statement	Nesting too deep
too long initializer	Initialization string constant too long
too long macro	Macro text too long
too many #defines	Number of #defines greater than limit.
too many case	Number of case statements greater than limit.
too many characters in character constant	Number of characters in the constant greater than limit.
too many characters in string constant	Number of characters in the string constant greater than limit.
too many initializers	Declared too many initialization expressions
too many label	Number of goto labels greater than limit
too many prototypes	Number of prototype definitions greater than limit
unacceptable operand of &	Operand of & operator invalid
unexpected EOF	The source program ends in the middle of the syntax
unknown size	The size is indefinite.
void function	Returns a value in the return statement even though it is a void function
zero or negative subscript	Negative or zero number of elements in array

8.15.2.Run-Time Error Messages

Error message	Description
`NO MEMORY`	Memory overflow
`BAD POINTER`	Pointer points outside the permitted range
`DIVISION BY 0`	Division by 0
`UNKNOWN ERROR`	Incorrect pointers destroyed the areas of the C program
`BAD FUNCTION`	Incorrect pointer value in function call.
`BAD STREAM`	Input/output data stream incorrect
`ARITHMETIC ERROR`	Calculation error (i.e. floating-point interruptions)
`FRAME ERROR`	Functional frame destroyed
`I / O ERROR OPEN`	Serial interface opened too often
`I / O ERROR`	Mini I/O unopened

9. CASL

9.1. The CASL assembler

The CASL assembler system was developed to teach assembly language and to understand the internal processes of a computer. The system consists of two parts, the CASL assembler and the COMET virtual machine. When developing the COMET system, great care was taken to ensure that all states of the system could be monitored and tracked. Unfortunately, this system has not established itself beyond the borders of Japan and the Philippines. In order to pass the exam for the Japanese Information Technology Standards Examination (JITSE) in Japan, a test in CASL / COMET had to be taken (in 2013, probably not more!?!). The extension to CASL II and COMET II from 2001 does not support this computer.

This manual explains the CASL II / COMET II specification, which focuses on handling the Sharp PC-G850

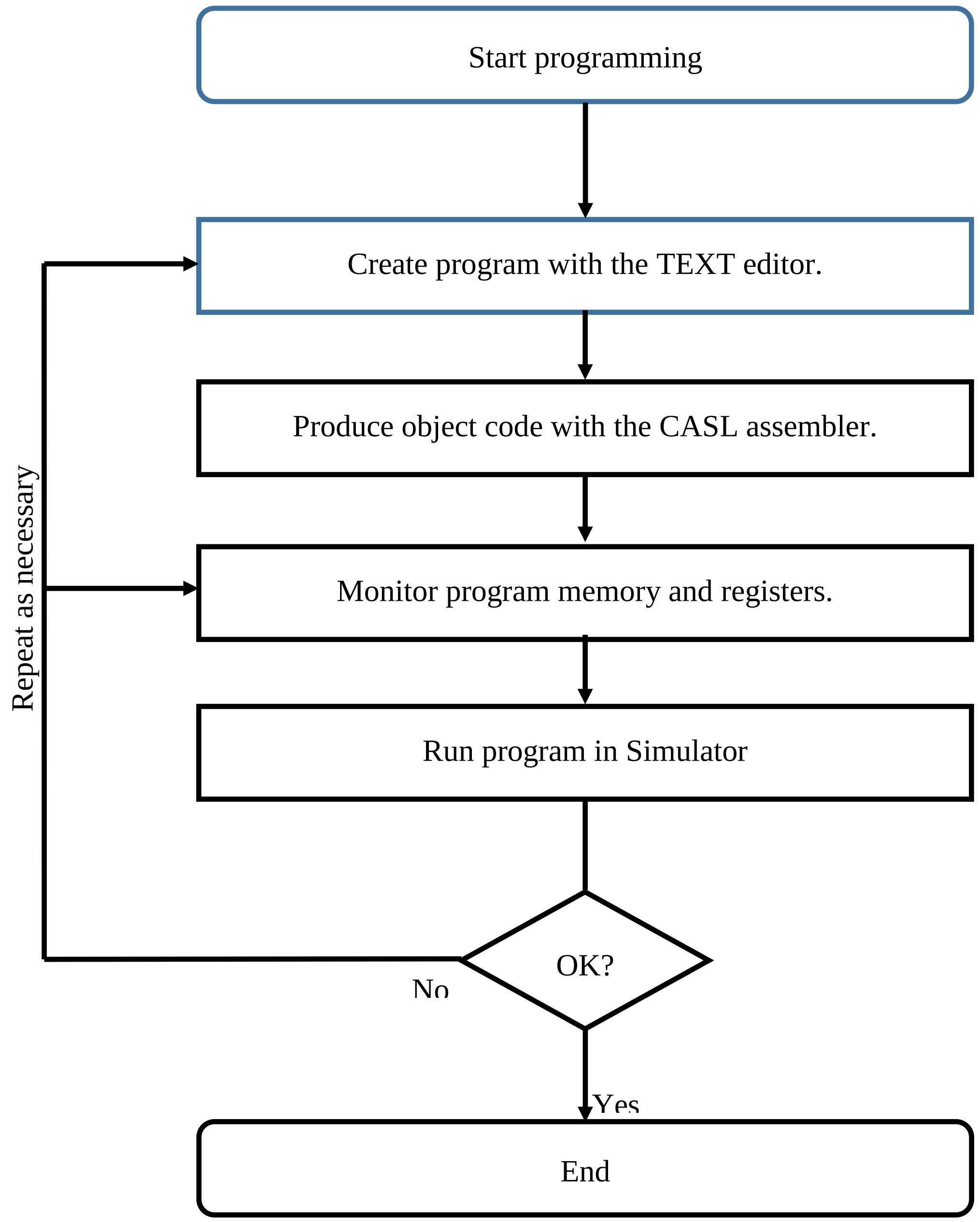
9.2. CASL mode Functions

The CASL mode consists of three functions:

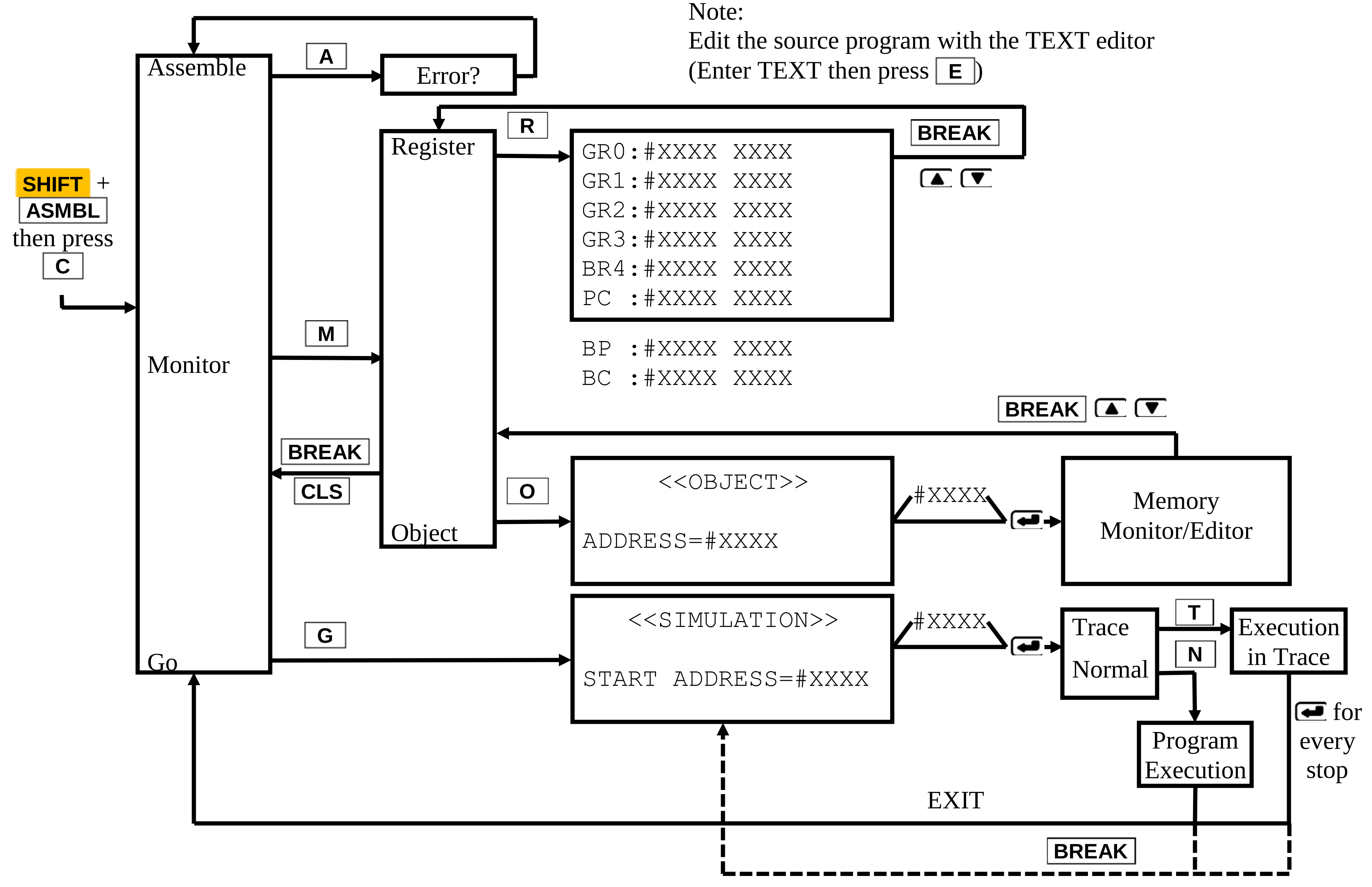
assembler:	Use the TEXT editor to write and save CASL program. The program can then be assembled and executed in CASL mode. If the CE-126P printer is connected, the output can be redirected to the printer.
monitor:	Used to monitor the program and the contents of the registers and change them. Likewise, they can change the data ranges defined with the DS command.
simulated execution:	The program is executed in a simulated environment. The program can be executed normally or in trace mode. The execution of the program can be stopped at defined breakpoints.

9.3. CASL Programming Procedure

Flowchart of the procedure for CASL programming.


Start programming
Create program with the TEXT editor.
Produce object code with the CASL assembler.
Monitor program memory and registers.
Run program in Simulator
OK?
No
Yes
End
Repeat as necessary

Note:
Edit the source program with the TEXT editor
(Enter TEXT then press E)



9.4. Entering / Editing the Source Program

The CASL source program is created and modified in **TEXT** mode using the editor.

$\boxed{\text{TEXT}}$ $\boxed{\text{E}}$

```
TEXT EDITOR
<
```

Detailed information on how to use **TEXT** mode can be found in chapter "TEXT mode".

9.4.1. Line Format

Structure of the source program:

```
32776 BGN    ADD    0, DAT, 1 ;SAMPLE
|<---->|<---->|<---->|<-------->|<------>|
Line number  Label  Command  Operands  Comment
```

Individual operands are separated by either spaces or tabs.

Line number	:	A number between 1 and 65279. If a number outside this range is specified, the message `<LINE NO. ERROR>` is displayed.
Label	:	The label consists of up to 6 alphanumeric characters. All subsequent characters are ignored. The label must start with a letter
Command	:	The command to be executed.
Operands	:	GR register, address operands or XR. Each operand must be separated by a comma. XR can be omitted.
Comment	:	Comments must begin with a semicolon (;) and are used to insert notes in the program.

Including comments, a line may have a maximum length of 254 characters.

Example program:

This program generates the output “♠♥CARDS♦♣”

```
10L1    START L2
20L2    OUT   DSP, N
30      EXIT
40N     DC    9
50DSP   DC    #E8
60      DC    #39
70      DC    'CARDS'
80      DC    #EA
90      DC    #EB
100     END
```

`10L1` TAB `START` TAB `L2` ↵	L2 is start of the program
`20L2` TAB `OUT` TAB `DSP, N` ↵	Output N characters with DSP
`30` TAB `EXIT` ↵	Return from program execution
`40N` TAB `DC` TAB `9`	
`50DSP` TAB `DC` TAB `#E8` ↵	"♠"
`60` TAB `DC` TAB `#39` ↵	"♥"
`70` TAB `DC` TAB `'CARDS'` ↵	"`CARDS`"
`80` TAB `DC` TAB `#EA` ↵	"♦"
`90` TAB `DC` TAB `#EB` ↵	"♣"
`100` TAB `END` ↵	Program end

9.5. The CASL Assembler

After the source code has been created in the text editor, all further steps are carried out in CASL mode.

SHIFT + **ASMBL** after that, press **C** to enter **CASL** mode.

```
          *** CASL ***

    Assemble Monitor Go

```

Press **A** to assemble source code.

```
          *** CASL ***

    Assemble Monitor Go

complete!
```

"`Assembling`" will show in the bottom line of the screen. If the assembler has executed successfully, the message "`complete!`" appears. If an error occurs during assembly, the process is terminated and an error message is displayed.

The finished object program is stored starting from address 1000.

9.5.1. CASL Assembler Log

During the assembly run, a log is generated in the following format:

```
ADD : OBJECT   : LINE NO.
               : 10
1000:7000 100B: 20
1002:7000 100A: 20
1004:8000 0002: 20
1006:1244 0002: 20
1008:6400 0004: 30
100A:0009     : 40
100B:00E8     : 50
100C:00E9     : 60
100D:0043     : 70
100E:0041     : 70
100F:0052     : 70
1010:0044     : 70
1011:0053     : 70
1012:00EA     : 80
1013:00EB     : 90
               : 100
|—1—|————2————|————3————|
LABEL :ADDRESS–
L1      1000   |
L2      1000   4
N       100A   |
DSP     100B   –
```

1) Address (16 bit)
2) Object (16 bit)
3) Line number of source program
4) Label

Note: The output of the protocol can also be redirected to a connected printer CE-126P. (for example, by **SHIFT** + **P↔NP**)

9.5.2. CASL Assembler Error Messages

If the assembler detects errors during execution, these error messages are displayed. Pressing CLS clears the error message on the screen. Afterwards you can correct the program with the text editor.

Type	Error Message	Description
Opcode error	`OP CODE ERROR (line number)`	Incorrect command code in specified line.
	`OP CODE ERROR (0)`	No source program
Operand error	`OPERAND ERROR (line number)`	Incorrect operand at the specified line number.
Label error	`LABEL ERROR (line number)`	Incorrect label at the specified line number
Memory error	`MEMORY ERROR (0)`	- insufficient memory. - insufficient stack space.
General error	`OTHER ERROR`	No START or END command was found or the source program has another syntax error

9.6. Simulation

Press G in the CASL menu to execute the program.

G

```
    << SIMULATION >>

START ADDRESS=#1000
```

The start address is displayed. This can be changed. If nothing else is entered and ↵ is pressed, the program will use address #1000.

You can enter the desired address in either decimal or hexadecimal form (preceded by a # sign).

The following display then appears:

```
    << SIMULATION >>
START ADDRESS=#1000

  Normal   Trace
```

Press N or T.

9.6.1. Normal Execution

Press N to start the program.

N

```
♠♥CARDS♦♣
```

↵

```
      *** CASL ***

  Assemble Monitor Go

_
```

Running programs can be aborted at any time with the BREAK key, e.g. To check or change registers or memory contents. The program can then be continued by pressing G in the **CASL** menu (`Go`).

9.6.2. Trace Mode

Press the T key. The registers (GR0-GR4), the program counter (PC), the flag register (FR) and the current command are displayed. Each time the ↵ button is pressed, the next command is executed.

T

```
1000: GR0:0000 GR4:1B0B
      GR1:0000 PC :1002
      GR2:0000 FR :0000
      GR3:0000 <PUSH>
```

↵

```
1002: GR0:0000 GR4:1B0A
      GR1:0000 PC :1004
      GR2:0000 FR :0000
      GR3:0000 <PUSH>_
```

If the output is redirected by SHIFT + P↔NP, the trace output is directed to the printer as follows:

```
ADD :GR0  GR1  GR2  GR3
1000:0000 0000 0000 0000
1002:0000 0000 0000 0000
1004:0000 0000 0000 0000
♠♥CARDS♦♣
0002:0000 0000 0000 0000
1006:0000 0000 0000 0000
1008:0000 0000 0000 0000
```

9.6.3. Trace Error Messages

Error Message	Description
OBJECT ERROR	No object program found
* MEM * * ERR *	JMP addresses an area outside the address space. The available memory has been exceeded
* OPR * * ERR *	The output of OUT has more than 97 characters

9.7. Monitor

The monitor function is used to check the contents of the registers of the COMET virtual machine. Likewise, you can change the object program or registers. You can also work with breakpoints. The monitor function can be accessed by pressing **M** in the main **CASL** menu.

M

```
      << MONITOR >>

   REGISTER    OBJECT
```

Now you can select:

- R : View and change the registers
- O : View and change the object program and memory

9.7.1. Display Register Contents

Press **R** to display the register contents.

R

```
GR0  #0000  0
GR1:#0000  0
GR2:#0000  0
GR3:#0000  0
GR4:#1BOB  6923
PC  :#1000  4096
```

Use the cursor keys (▲ ▼) or ↵ to scroll the display. The current register is indicated by a missing colon.

⋮
▼

```
GR3:#0000  0
GR4:#1BOB  6923
PC  :#1000  4096
FR  :#0000  0
BP  :#FFFF  65535
BC   #0000  0
```

register	name	description
GR0-GR4	General register	Universal register. GR4 is used as a stack pointer
PC	Program counter	Points to the next command to be executed
FR	Flag register	Result of executing a command (positive, zero, negative)
BP	Break pointer	Used to control the execution of the simulation.
BC	Break counter	

9.7.2. Set Registers

The contents of the selected register (where the colon is off) can be set with values as follows:

Decimal	: Enter a decimal number from -32768-65535	
	Example: 123 ↵	Output: `#007B 123`
Hexadecimal	: Enter a hexadecimal number from 0-FFFF	
	Example: #007B ↵	Output: `#007B 123`
Label	: Label enclosed in double quotes	
	Example: "L1" ↵	Output: `#100A 4106` (L1 points to # 100A)
Character	: Characters enclosed in quotes	
	Example: 'A' ↵	Output: `#0041 65`

Pressing $\boxed{\textbf{CLS}}$ aborts the entry without changing the original value.

Note: The contents of register FR use only 2 bits (values 0,1 and 2) all other bits are ignored.

Contents of the registers after reset / start of the assembler:

GR0-GR3	: 0
GR4	: upper address + 1 of the object area
PC	: Start address of the program (address of the label of the start instruction)
FR	: 0
BP	: FFFF (Hex)/65535 (decimal), No breakpoint set
BC	: 0

Note: GR4 can normally be used freely. The address of the object area can be changed in the monitor or in the program.

9.7.3. Display Object Code

Press $\boxed{\textbf{O}}$ in the **CASL** menu to display object code.

$\boxed{\textbf{O}}$

```
      << OBJECT >>


ADDRESS=#1000
```

Confirm the start address with ↵ or change this address beforehand.

↵
▼
⋮
▼

```
1000:7000 PUSH #100B
1001:100B
1002:7000 PUSH #100A
1003:100A
1004:0000 CALL #0002
1005:0002

1006 1244 LEA  4, #0002, 4
```

Note: If no program was previously loaded into the CASL assembler, the error `<OBJECT ERROR>` is displayed.

Use the cursor keys (▲ ▼) or ↵ to scroll the display. The current address is indicated by a missing colon.

The content of the selected address (where the colon is off) can be set with values as follows:

Decimal	: Enter a decimal number from -32768-65535 Example: 123 ↵ Output: `#007B 123`
Hexadecimal	: Enter a hexadecimal number from 0-FFFF Example: #007B ↵ Output: `#007B 123`
Label	: Label enclosed in double quotes Example: "L1" ↵ Output: `#100A 4106` (L1 points to # 100A)
Character	: Characters enclosed in quotes Example: 'A' ↵ Output: `#0041 65`

Pressing **CLS** aborts the entry without changing the original value.

9.8. Sample CASL Program

Here is an example program that adds 5 numbers:

This program adds the numbers in line 130-170 to DAT (line 120).

```
10EXAM  START
20BGN   LEA    GR0, 0          Write 0 to register 0
30      LEA    GR1, 0          Write 0 to register 1
40      JMP    AGN1            unconditional jump to AGN1
50AGN   ADD    GR0, DAT, GR1   Add the contents of DAT to register 0 (with
                               shift GR1)
60      LEA    GR1,1, GR1      Increase value of register 1 by 1
70AGN1  CPA    GR1, N0         Compares the numbers in GR1 and N
80      JMI    AGN             On negative result (i.e. N > GR1), jump to AGN
90      ST     GR0, TTL        Stores the number in register 0 after TTL
100     EXIT                   End of program / return jump
110N    DC     5
120TTL  DS     1               Define storage space
130DAT  DC     #000C
140     DC     #07F3
150     DC     #0231
160     DC     #0009
170     DC     #000F
180     END                    End of program
```

9.8.1. Operation Example:

Input	Output

	`*** CASL ***`
	`Assemble Monitor Go`

M	`<< MONITOR >>`
	`Register   Object`

O	`<< OBJECT >>`
	`ADDRESS=#1000`

↵
▼
⋮
	`1000 1200 LEA  0, #0000`
	`1001:0000`
	`1002:1212 LEA  1, #0000`
	`1003:0000`
	`1004:6400 JMP  #100A`
	`1006:2001 ADD  0, #1014, 1`
	`1007:1014`
	`1008:1211 LEA  1, #0001, 1`
	`1009:0001`
	`100A:4010 CPA  1, #1012`
	`100B:1012`

▼ ⋮	100C:6100 JMI #!006 100D:1006 100E:1100 ST 0, #1013 100F:1013 1010:6400 JMP #0004 1011:0004 1012:0005 * 1013:0000 1014:000C * 1015:07f3 1016:0231 ***** 1017:0009 1018:000F * 1019:0000
ON	<< MONITOR >> Register Object

Input	Output
	*** CASL *** Assemble Monitor Go
M	<< MONITOR >> Register Object
R ▼ ⋮ ▼	GR0:#0000 0 GR1:#0000 0 GR2:#0000 0 GR3:#0000 0 GR4:#1AA5 6821 PC :#1000 4096 ⋮ BP #FFFF 65535
#100C	BP #FFFF #100C
↵	BP #100C 4108
▼	BC #0000 0
2 ↵	BC #0002 2
BREAK BREAK	*** CASL *** Assemble Monitor Go
G ↵	<< SIMULATION >> START ADDRESS-#1000 Normal Trace
N	100C: GR0:000C GR4:1AA5

	* * GR1:0001 PC :100C
	STP GR2:0000 FR :0002
	* * GR3:0000 <JMI>
↵	100C: GR0:07FF GR4:1AA5
	* * GR1:0002 PC :100C
	STP GR2:0000 FR :0002
	* * GR3:0000 <JMI>
BREAK BREAK	*** CASL ***
	Assemble Monitor Go
M R	GR0:#07FF 2047
	GR1:#0002 2
	GR2:#0000 0
	GR3:#0000 0
	GR4:#1AA5 6821
▼	PC :#100C 4108
⋮	⋮
▼	BC #0000 0
4 ↵	BC #0004 4
BREAK BREAK	*** CASL ***
	Assemble Monitor Go
G ↵	<< SIMULATION >>
	START ADDRESS-#100C
	Normal Trace
N	100C: GR0:0A48 GR4:1AA5
	* * GR1:0005 PC :100C
	STP GR2:0000 FR :0001
	* * GR3:0000 <JMI>
BREAK BREAK	*** CASL ***
	Assemble Monitor Go

9.8.2. Trace Example

Input	Output	Opcode
BREAK BREAK	*** CASL *** Assemble Monitor Go	
G ↵	<< SIMULATION >> START ADDRESS-#100C Normal Trace	
T	1000:GR0:0000 GR4:1AA5 GR1:0000 PC :1002 GR2:0000 FR :0001 GR3:0000 <LEA>	LEA GR0, 0
↵	1002:GR0:0000 GR4:1AA5 GR1:0000 PC :1004 GR2:0000 FR :0001 GR3:0000 <LEA>	LEA GR1, 0
↵	1004:GR0:0000 GR4:1AA5 GR1:0000 PC :100A GR2:0000 FR :0001 GR3:0000 <JMP>	JMP AGN1
↵	100A:GR0:0000 GR4:1AA5 GR1:0000 PC :100C GR2:0000 FR :0002 GR3:0000 <CPA>	CPA GR1, N
↵	100C:GR0:0000 GR4:1AA5 GR1:0000 PC :1006 GR2:0000 FR :0002 GR3:0000 <JMI>	JMI AGN
↵	1006:GR0:000C GR4:1AA5 GR1:0000 PC :1008 GR2:0000 FR :0000 GR3:0000 <ADD>	ADD GR0, DAT, GR1
↵	1008:GR0:000C GR4:1AA5 GR1:0001 PC :100A GR2:0000 FR :0000 GR3:0000 <LEA>	LEA GR1, 1, GR1
↵ ⋮ ⋮ ⋮ ⋮ ↵	100A:GR0:000C GR4:1AA5 GR1:0001 PC :100C GR2:0000 FR :0002 GR3:0000 <CPA> ⋮ 1004:GR0:0A48 GR4:1AA5 GR1:0005 PC :100C GR2:0000 FR :0001	CPA GR1, N

	GR3:0000 <CPA>	
↵	100C:GR0:0A48 GR4:1AA5 GR1:0005 PC :100E GR2:0000 FR :0001 GR3:0000 <JMI>	JMI AGN
↵	100E:GR0:0A48 GR4:1AA5 GR1:0005 PC :1010 GR2:0000 FR :0001 GR3:0000 <ST>	ST GR0, TTL
↵	1010:GR0:0A48 GR4:1AA5 GR1:0005 PC :1004 GR2:0000 FR :0001 GR3:0000 <JMP>	EXIT
↵	*** CASL *** Assemble Monitor Go	

9.9. COMET Specification

Based on the COMET / CASL specification, the Japanese Ministry of Economy, Trade and Industry in 2001, drafted the following specification of COMET II and CASL II:

1. START
   Defines the start of a program. By default this is the address #1000.
2. DC
   Defines a memory area with a decimal value of -32768-65535 (hex # 0000- # FFFF) or a string.
3. IN (CALL # 0000)
   Entering characters from the screen: The input request is a question mark (‘?’). The entry is completed by ENTER. The first operand is the address where the input should be written. The number of read characters is written to the address of the second operand. By pressing the ▼ key, the input is ignored and the number 65636 (#FFFF) is passed.
4. OUT (CALL #0002)
   Output a string: This command corresponds to the Basic command PRINT. The first operand is the address where the characters are to be output. The second operand specifies the address in which the number of characters to be output is stored.
5. WRITE CALL (#0006)
   Outputs the contents of the registers on the screen. Press ↵ to continue the program.
6. END
   Defines the end of the program.

9.10.COMET Architecture

Technical information of the COMET architecture is listed below for a better understanding of the CASL assembler.

Word length:	16 bits? (Each memory address has a length of 16 bits, as opposed to the length of 8 bits of a normal computer)
Architecture:	Von Neumann
Numbers:	16-bit binary numbers. Negative numbers are represented by the two's complement.
Register:	*GR0-GR4 (16bit)*: General Register. GR1-GR4 are also used as index registers. However, register GR4 is mainly used as the stack pointer (SP). The stack starts at the top free address of the comet machine and grows down with each new entry. *PC (16bit)*: Program Counter. This register contains the address of the next instruction to be executed. *FR (2bit)*: Flag Register. This register contains the result of comparison operations 00 = larger (positive), 01 = equal (NULL), 10 = smaller (negative)
Stack:	The stack starts at the top free address of the Comet Machine and grows down with each new entry. Register GR4 points to the most recently stored value. If no value is stored, GR4 points to the last address + 1. If we enter a value with `PUSH`, the address in GR4 decreases by 1. `POP`, on the other hand, increases the address again by 1.

9.11.Command Summary

Commands have a length of 2 16-bit words

Word 1			Word 2			
OP				Syntax		
	GR	XR	adr	Command	Operand	Description
0 0						
1 0				LD	GR, adr, XR	load
1 1				ST	GR, adr, XR	store
1 2				LEA	GR, adr, XR	load effective address
2 0				ADD	GR, adr, XR	add arithmetic
2 1				SUB	GR, adr, XR	subtract arithmetic
3 0				AND	GR, adr, XR	and
3 1				OR	GR, adr, XR	or
3 2				EOR	GR, adr, XR	exclusive or
4 0				CPA	GR, adr, XR	compare arithmetic
4 1				CPL	GR, adr, XR	compare logical
5 0				SLA	GR, adr, XR	shift left arithmetic
5 1				SRA	GR, adr, XR	shift right arithmetic
5 2				SLL	GR, adr, XR	shift left logical
5 3				SRL	GR, adr, XR	shift right logical
6 0	0			JPZ	adr, XR	jump on plus or zero
6 1	0			JMI	adr, XR	jump on minus
6 2	0			JNZ	adr, XR	jump on non-zero
6 3	0			JZE	adr, XR	jump on zero
6 4	0			JMP	adr, XR	unconditional jump
7 0	0			PUSH	adr, XR	push effective address
7 1		0	0000	POP	GR	pop up
8 0	0			CALL	adr, XR	call subroutine
8 1	0	0	0000	RET		return from subroutine
9 ⋮ F				not used		

9.11.1.Registers and Abbreviations

There are 23 CASL commands defined for this computer. This section describes the registers and defines the abbreviations used in the command descriptions:

GR	: GR0-4, the general purpose registers
XR	: XR0-4, the optional index registers. (There are no special XR registers, these correspond to the GR registers).
SP	: The stack pointer. This is represented by register GR4.
adr	: A 16-bit number that corresponds to a label or number to be processed. The number ranges from -32758 – 65535 decimal or #0000 – #FFFF hexadecimal.
Valid address	: An address *adr* which returns the address value and index XR.
[]	: Optional parameter

9.11.2.Commands

LD

Format: `LD GR, adr [, XR]`

Description: The contents of address *adr* is written to the specified register GR0-GR4.

ST

Format: `ST GR, adr [, XR]`

Description: The contents of register *adr* is written to the specified memory address.

LEA

Format: `LEA GR, adr [, XR]`

Description: The value of *adr* is written to the register.

Example:

`LEA GR1, 100`	Load the value 100 to register GR1
`LEA GR1, 10, GR1`	Increase value in register GR1 by 10
`LEA GR1, 0, GR2`	Copy contents of GR1 to GR1

ADD

Format: `ADD GR, adr [, XR]`

Description: The register GR is added to the value in address *adr*.

SUB

Format: `SUB GR, adr [, XR]`

Description: The register GR is subtracted by the value in address *adr*.

AND, OR, EOR

Format:

```
AND GR, adr [, XR]
OR GR, adr [, XR]
EOR GR, adr [, XR]
```

Description: The content in address *adr* are bitwise (16 bit) compared using logical AND, OR, or XOR with the contents of GR.

CPA

Format:

```
CPA GR, adr [, XR]
```

Description: The content at address *adr* is compared to the contents of register GR. CPA compares arithmetically and interprets the values as numbers (-32768 – 32767). CPL compares logically and interprets the content bitwise

(GR) > value	FR = 00 (0)
(GR) = value	FR = 01 (1)
(GR) < value	FR = 10 (2)

JPZ, JMI, JNZ, JZE

Format:

```
JPZ adr [, XR]
JMI adr [, XR]
JNZ adr [, XR]
JZE adr [, XR]
```

Description: Branches the program to the specified address when the condition is met.

JPZ : jump to address if comparison is positive or zero (FR = 00 or 01)
JMI : Jump to address if comparison is negative (FR = 10 [2])
JNZ : jump to address if comparison not zero (FR = 00 or 10)
JZE : jump to address if comparison is zero (FR = 01 [1])

JMP

Format:

```
JMP adr [, XR]
```

Description: Branches the program to the specified address *adr*.

SLA, SRA

Format:

```
SLA GR, adr [, XR]
SRA GR, adr [, XR]
```

Description: Arithmetic bitwise shift. The content of the register is shifted bitwise to the left or right by the number of bits indicated by *adr* (plus the optional content of XR), The sign (bit 15) always remains. In the case of negative numbers, a 1 instead of a 0 is inserted during the right shift. The FR register is set according to the result.

SSL, SLR

Format: `SLL GR, adr [, XR]`
`SRL GR, adr [, XR]`

Description: Logical bitwise shift. The content of the register is shifted bitwise to the left or right by the number of bits indicated by *adr* (plus the optional content of XR).

PUSH

Format: `PUSH adr [, XR]`

Description: Writes the contents of the address *adr* in the stack. The stack address (SP) in register GR4 is set to this new TOP stack address.

POP

Format: `POP GR`

Description: Writes the contents of the TOP address of the stack to the specified register. The TOP stack address (SP) in register GR4 is set to the previous stack address.

CALL

Format: `CALL adr [, XR]`

Description: Calls a subroutine at the specified address *adr*. The return address (address after `CALL`) is placed on the stack.

RET

Format: `RET`

Description: Returns from a subprogram back to the calling program (`CALL`). The return address is taken from the stack.

9.11.3. Assembler Syntax

Label	command	operand	Description
[Label]	START	[Start Label]	Indicates the beginning of the program
	END		End of the program
[Label]	DC	constant	Defines numbers or strings
[Label]	DS	Number of words	Defines a memory area
[Label]	IN	String length	Reads characters from the screen
[Label]	OUT	String length	Writes characters to the screen
[Label]	EXIT		Program return
[Label]	WRITE		Output registers to the screen

START

Format: `START [label]`

Description: Indicates start of a CASL program. Optionally, a label can be specified where program execution should be started. Otherwise, the command following the `START` statement is executed.

END

Format: `END`

Description: Sets the end of the CASL program.

DC

Format:

```
DC n
DC #h
DC 'string'
DC label
```

Description: Define constant. Parameters for the various constant types are listed below:

`n`	Defines a number constant (decimal). The value of the constant must be between -32768 and 65535.
`#h`	Defines a number constant (hexadecimal). The value of the constant must be between #0000 and #FFFF.
`'string'`	Defines a string. Each byte is stored in the right half of an address (16 bits). From the character code table, the characters 32 – 38 (&H20 – &H26), 40 – 95 (&H28 – &H5F), 97 – 122 (&H61 – &H7A), 166 – 223 (&HA6 – &HDF) can be used. No length is stored within the string. The program has to know how long the string is.
`label`	Defines a constant containing the address of the given label.

DS

Format: `DS [n]`

Description: Defines a memory area containing *n* words. If the number is 0, only the label for the next following address is defined.

Special note: if *n* is assigned #0000 during program execution, the execution of the program is interrupted. * STP * will appear in the register display. This can be used to stop the program in one place to check or change registers and memory. The command counter is incremented by 2, so that at the next start, the program can be continued. However, it should be noted that two words with #0000 (for example, with 2x ÍDC 0Í) must be defined in the program because the program counter is always increased by two.

9.11.4. Macro Commands

IN

Format: `IN adr, length`

Description: Enter characters from the keyboard. The input prompt is a question mark ('?'). The entry is completed by pressing ↵. The first operand is the address where the input should be written. The address of the second operand contains the number of characters read. Pressing ▼ will skip the next entry and pass the number 65536 (#FFFF).

OUT

Format: `OUT adr, length`

Description: Output a string. This command corresponds to the BASIC command `PRINT`. The first operand is the address where the characters are to be output. The second operand specifies the address where the number of characters to be output must be stored. The program is interrupted after the output and may be resumed by pressing ↵.

EXIT

Format: `EXIT`

Description: Ends the execution of the program.

WRITE

Format: `WRITE`

Description: Returns the contents of the registers on the screen. Press ↵ to continue the program.

9.11.5. Sample Program

```
10      START
20      IN      A, C              Read characters in A
25      OUT     NL, N             Output of 9x'P' as separation
30      OUT     A, B              Output of the first 2 characters of A
40      EXIT                      end program execution
50A     DS      20                Input buffer with 20 words (characters)
60B     DC      2                 Output length 2
70C     DS      1                 Storage of the number of read characters
80N     DC      9                 Output length for 9x'P'
90NL    DC      'PPPPPPPPP'       String with 9x'P'
100     END                       End of source program
```

Listing of the sample program in memory:

```
IN   A,C     7000   101A   PUSH  A
             7000   102F   PUSH  C
             8000   0000   CALL  #0000
             1244   0002   LEA   GR4,2,GR4
OUT  NL,N    7000   1031   PUSH  NL
             7000   1030   PUSH  N
             8000   0002   CALL  #0002
             1244   0002   LEA   GR4,2,GR4
OUT  A,B     7000   101A   PUSH  A
             7000   102E   PUSH  B
             8000   0002   CALL  #0002
             1244   0002   LEA   GR4,2,GR4
EXIT         6400   0004   JMP   #0004

A DS 20      101A..102D   #0000 (20x)
B DC 2       102E         #0002
C DS 1       102F         #0000
N DC 9       1030         #0009
NL DC ‘PP…   1031..1039   #0050 (9x)
```

10. MACHINE LANGUAGE MONITOR

With this computer, you can write programs in both machine language and BASIC. The computer has a machine language monitor (hereafter referred to as "the monitor") to assist with programming in machine language. The monitor allows you to enter or issue a specific sequence of commands or execute machine language programs. This section describes the functions of the machine language monitor commands for this computer.

The computer's CPU is a Z80 microprocessor (CMOS Z80A), which is commonly used in most 8-bit computers. There are numerous books available about the Z80 processor, which provide information about the machine language instruction set of the Z80 and other important information. This chapter describes the behavior of the machine language monitor commands, how to create a source program, and how to run it.

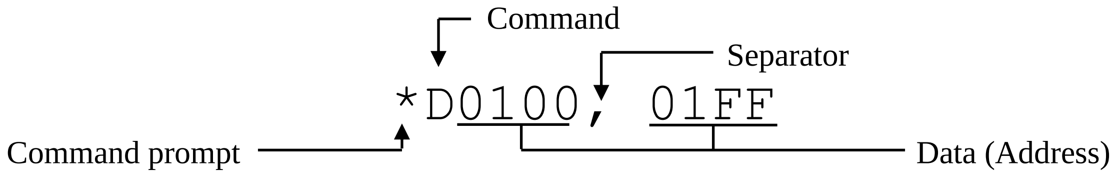
10.1. Using the Monitor

Monitor mode is selected by entering `MON` in **BASIC** mode (**RUN** or **PRO**). The following display appears.

```
MACHINE LANGUAGE MONITOR
*
```

The asterisk (*) on the display is the command prompt waiting for input. All commands are entered here. All necessary addresses or further data can be entered here after the command. At the end of each line, the entry must be executed by pressing ↵.

Example:



Notes:

1. If memory protection is enabled with a password, the computer cannot be set to monitor mode.
2. All addresses and data must be in hexadecimal.
3. To separate more than one address or to separate data parts, a comma (,) is used.
4. If hexadecimal is not used or another symbol is entered other than the comma, an error occurs (`SYNTAX ERROR`).
5. The monitor mode can be excited by selecting a different mode or by turning the computer off and on again.
6. Since machine language is very complicated, it often comes down to program bugs. When running a machine-language program, BASIC programs, data, or other parts of

the computer memory may be destroyed. For this reason, it is recommended that you back up all BASIC programs, data, or other information to a PC before running a machine language program.

7. When using the monitor, accessing anything other than the machine language area (assigned with the USER command) may result in the destruction of BASIC or TEXT programs, destroy data, or cause malfunctions. Be sure to use only the intended machine code area.

10.2.Monitor Commands

USER — Set User Memory

Format: (1) `USER01FF`
(2) `USER`
(3) `USER00FF`

Description: Allocate memory for the monitor and display the addresses of this area.

Comments: Format (1): memory address range of 0100H (start address) to 01FF (end address) is assigned for machine code. The first address is automatically set to 0100H.

Assigned memory→
```
*USER01FF
FREE:0100-01FF
                   *
```

Format (2): displays the address range assigned for machine code. If no area has been assigned for machine code, "`FREE:NOT RESERVED`" is displayed.

```
*USER
FREE:0100-01FF
                   *
```

Format (3): deletes existing machine code assignment from memory and displays the message "`FREE:NOT RESERVED`".

An error message (`MEMORY ERROR`) is displayed when an invalid address range for machine code assignment is entered.

S — Update Memory

Format: (1) `S0100`
(2) `S`

Description: Update memory address.

Comments: Format (1): The contents of address 0100H (first address) is displayed and prompts for a new entry.

```
*S0100
0100:01-
```

Existing content at address

To change the memory contents, enter one byte (two digit hexadecimal) and then press ↵. The computer will now show the contents of the following memory address and will ask for input.

If you not want to change the memory content at the current address, press ↵ without entering any data. The computer then displays the contents of the following memory address and asks for input

A maximum of two hexadecimal digits can be entered. To delete an entry before pressing ↵ press ◀ or **CLS**.

Press ▲ to retrieve the contents of the previous address and ▼ to retrieve the contents of the next address.

Format (2): Display the contents of the address immediately after the last address specified by the `S` command.

Press **BREAK** to return to the command line.

D — Display Memory

Format: (1) `D0100`
(2) `D`
(3) `D0100, 01FF`

Function: Display memory address.

Description: Format (1): Displays the first 16 bytes from the address range 0100H (first address) to 010FH. (The output is printed in the printer mode.)

Example:

First address of 16-byte block→
Checksum→

```
0100 : 3E 01 18 04  >...
(1D)   3A 0F 01 3C  :..<
       32 0f 01 C9  2.."
       31 00 00 00  1...
```

ASCII code is displayed here. Hex values 00H-1FH are displayed as (.)

The address range of the memory displayed is set to XXX0H-XXXFH. If the address specified is within a 16-byte block, the entire block which contains the address is displayed. For example, if you specify the address 0104H, the contents of the 16-byte block, in this case 0100H-010FH, is displayed.

Press ▲ to display the previous 16-byte block and to ▼ display the next 16-byte block.

Format (2): displays the contents of the block that is directly next to the last block displayed with the `D` command.

Format (3): When executed in **PRINTER** mode, the computer prints the contents of the areas, 0100H (first address) – 01FFH (last address), in 16-byte increments to the printer. When the output is finished, the command line is displayed.

The printer mode is toggled with the `P` command (see later) or with **SHIFT** + **P↔NP**.

If the computer is not in **PRINTER** mode, the computer will display the contents of the 16-byte blocks on the screen, beginning with the address 0100H (the first address). The computer does not take into account the last address specified during display.

To return to the command line, press **BREAK**.

Checksum: Checksum refers to the sum of the values of a specific record. This sum is calculated and assigned to a record when that record is written or displayed. The computer calculates the sum of the contents of a 16-byte block output with the `D` command and displays the least significant byte of the sum as the result of the checksum. For example, if you manually enter a machine code program that is copied from a printed program, you can check for errors in each 16-byte block by comparing the checksum results with the values of the original program. However, if the program contains more than one error, the checksum may erroneously match that of the original program.

E — Edit Memory

Format: (1) `E` *`start-address`*
(2) `E`

Function: Edit memory area.

Description: Format (1): memory editing will start at the specified address.

Format (2): continues editing with the subsequent memory block from the last edit.

The command `S` can also be used to change memory contents. The difference is using the editor is more convenient. The editable range is 0000H – 07FFFH.

Use the cursor keys to move in the memory area.

The data is entered in hexadecimal notation 0-F. In addition to the keyboard, the keypad can be used as follows:

7	8	9	/ (F)
4	5	6	* (E)
1	2	3	− (D)
0	. (A)	= (B)	+ (C)

Use the **TAB** key to toggle between hexadecimal input (left) and ASCII input (right). Kana mode is not possible during editing.

P — Toggle Printer

Format: `P`

Function: Enable or disable printer mode.

Description: The printer mode is toggled each time P is pressed (when printer mode is activated, **PRINT** appears in the lower right corner of the display.)

Printer mode can also be toggled by pressing **SHIFT** + **P↔NP**.

Note: The `P` command will not be executed if no printer is connected or a connected printer is not turned on.

G — GOSUB

Format: `G` *`address`*

Function: Execute a machine code program at a specific address.

Description: The `G` command corresponds to the `GOSUB` command in BASIC. A machine code program at the specified address will be run. Execution is complete when a `RET` command (return instruction) occurs. After the return instruction, the computer displays the command line.

Note: A program a return instruction (`RET` command) must be inserted, otherwise the program will not run correctly.

Out of control programs

A “runaway” program cannot run properly because it is out of control. Resetting the system is the only way to interrupt such a program. In most cases, an out of control program destroys the memory contents, including machine code programs, BASIC programs, and other data.

A machine code program can get out of control even if it contains a single bug. For this reason, it is recommended that you save or print all BASIC programs and other information on a PC before running a machine language program.

R — Receive data via the serial interface

Format: (1) `R`
(2) `R <address>`

Function: Receive data via the serial I / O port (SIO).

Description: The `R` command is used to transmit / receive data in Intel hex format over SIO. This command is for receiving machine code from a personal computer or other device.

Format (1): loads data into an address specified by the data.

Format (2): loads data starting at the specified address (e.g., 0100H).

After completion, the address range where the data was loaded is displayed.

To stop receiving data, press and hold the **BREAK** key until the command prompt is displayed.

The settings for the serial interface are set in **TEXT** mode.

W — Send data via the serial interface

Format: `W` *`start-address, end-address`*

Function: Send data via the serial I / O port (SIO).

Description: The `W` command sends data in Intel Hex format from the memory area specified (for example, W0100h, 01FF) to the serial I / O port. This command is for sending machine codes to a personal computer or other device.

To stop sending data, press and hold the **BREAK** key until the command prompt is displayed.

If a printer is connected to the peripheral interface male (11-pin) connector and the `W` command is executed, both the computer and the printer may malfunction. In this case, turn off the printer, and then press and hold the **BREAK** key until the command prompt is displayed.

BP — Set Breakpoint

Format: (1) `BP` *`address`* `[,` *`number`*`]`
(2) `BP`
(3) `BP 0`

Function: Insert a breakpoint at a specific address.

Description: Format (1): inserts a breakpoint at the specified address. Up to 4 breakpoints can be inserted at different addresses. The possible address range is from 0000H to 7FFFH.

With *number*, you can specify how many times execution occurs at the specified address before the program stops. A value of 0-255 can be specified. Specifying 0 clears the breakpoint. If a number is not specified, the value is set to 1, meaning that execution is stopped when the breakpoint is reached for the first time.

When attempting to enter a fifth breakpoint, the first breakpoint is deleted. Therefore, there can never be more than four breakpoints in a program.

A breakpoint should be inserted at an instruction address (OP code). If the breakpoint is inserted at an operand address, the program cannot read the breakpoint and will not run properly.

Format (2): displays the address of the breakpoint. If no breakpoint has been inserted, only the command prompt (*) appears on the following line.

Format (3): all existing breakpoints are deleted.

A breakpoint becomes inactive after execution, so if there is a breakpoint within a program loop, it will only be activated at the first or nth (according to number) execution of the loop. It can be reactivated with the G command.

The computer maintains breakpoints that were set when the monitor was last used. If the computer is set to Monitor mode from another mode, these breakpoints can be re-enabled with the G command.

Note: The contents of an address that contains a breakpoint are temporarily replaced with "F7H" while the program is running. If the RESET switch is pressed before activating the breakpoint, the contents will remain "F7H". In this case, replace "F7H" with the original contents.

10.3. Error Messages in Monitor Mode

Following is a list of error messages that are displayed during monitor mode. To clear the error message, press **CLS**.

Error message	Description
`SYNTAX ERROR`	Invalid command syntax
`MEMORY ERROR`	An attempt was made to assign a machine code area outside the allowable range.
`I/O DEVICE ERROR`	Error in the data transfer or error of the checksum during an I / O operation
`OTHER ERROR`	Other mistakes.

11. ASSEMBLER

The following is a list of specialized vocabulary frequently used when dealing with machine language programs.

Assemble, translate:	Translate assembly language source code into a machine language. A translated machine code program is called "object program" or "object" for short.
Assembler:	Translation program for translating a source program into an object program.
To generate:	Create an object from a mnemonic code.
Assembling by hand:	Manual translation of a source program without an assembler.
Machine language:	A computer language that is interpreted directly by a machine and whose commands are executed. Displayed as hexadecimal code (internally processed as binary code)
Mnemonic Code:	Icons designed to help the programmer keep the machine code instructions. For example, the abbreviation "ADD" for an additional command (additional command). A language whose mnemonic statements have a specific match with the machine code is called "assembly language".
Patch:	A fully assembled program that is ready to load into a computer. The term generally refers to a machine code program that has been translated by a source program. Sometimes referred to simply as an "object". ("Object" can either refer to an individual machine code resulting from a translation, or it can refer to a whole machine language program.)
Pseudo-instruction:	A sequence of assembler control commands that are not translated into a machine code program. Such a sequence is used to determine an address, store a machine code program or generate data.
Source program:	A program written in a mnemonic code (assembly language). A machine code program is a translation of a source program.

11.1. Programming with the Assembler

An assembler program is translated into object code (the machine code program). However, error conditions can occur during program execution. If the machine program contains one or more bugs, the following error states may occur:

- The program is stuck in an infinite loop and stops responding to keypresses. To interrupt the endless loop, press the **RESET** button.

- The program will display random or nonsensical characters or show other issues. In some cases, the program may be stopped using the **BREAK** button, but in other cases, the **RESET** button must be pressed.
- Parts or the whole program will be destroyed or lost. In this case, there is a memory error. It can also lead to the destruction of source programs (`TEXT`), `BASIC` programs or all data on the computer, including the machine code program.

These problems can occur individually or simultaneously. If any of these problems occur and you cannot determine what is happening, press the **RESET** key to clear all memory. Problems (1) and (2) are called "runaway programs". A brief guide to programming the Z80 processor can be found in **APPENDIX L**: Z80 PROGRAMMING .

11.1.1.Example Program

The following program loads the hexadecimal numbers 20H – 9FH to memory addresses 0400H – 477FH (the H at the end indicates that it is a hexadecimal notation):

```
10        ORG   0100H
20START:  LD    A, 20H
30        LD    HL, 0400H
40LBL:    LD    (HL), A
50        INC   A
60        INC   HL
70        CP    0A0H
80        JP    NZ, LBL
90        RET
100       END
```

Note: One or more spaces can be inserted with the **SPACE** or **TAB** key.

Description of the example program:

10: (Load the object starting at address 0100H).
20: Load 20H into the register `A`.
30: Load 0400H into the register pair `HL`.
40: Load content in register `A` into an address specified by register pair `HL`
50: Increase the value of register `A` by one and load the result into register `A`.
60: Increase the value of register pair `HL` by one and load the result in `HL`.
70: Compare the contents of register `A` with the value A0H (A0H-content of `A`).
80: If the result of the last operation is not zero (content of `A` ≠ A0H), jump to the label `LBL` (the label is translated into address 0105H).
90: Return from the subroutine.
100: (end of source program).

Lines 10 and 100 of this source program are called pseudo-instructions. They are used to control the assembler and are not converted into machine codes (objects).

Note: After entering all the lines of this example program, double check for errors. Before assembling the source program, a memory block must be assigned to store the machine code, otherwise it is not possible to assemble the source program.

11.1.2.Assign Machine Code Area

To assign a machine code area, the `USER` command is used in monitor mode.

First, monitor mode is selected.
BASIC MON ↵

```
MACHINE LANGUAGE MONITOR
*
```

Next, memory is allocated for machine code with the `USER` command. In this example, a block of memory from 0100H to 04FFH is allocated.

`USER 04FF`↵

```
MACHINE LANGUAGE MONITOR
*
```

The computer displays the assigned machine code area (user area).

```
MACHINE LANGUAGE MONITOR
*USER04FF
FREE:0100-04FF
*
```

11.1.3.Assemble Source Program

The source program of this example can be converted into machine code.

Select the assembler function **SHIFT** + **ASMBL**
(The size of the work area may be different than that shown in this example.)

```
 ***** ASSEMBLER *****
 user area=0100H-04FFH
 work area=29221bytes
< ASM  Display  Print >
```

Press **A** to start assembling.

```
 ***** ASSWMBLER *****

   --- assembling ---
```

When the assembly is complete, a screen similar to the one shown on the right appears.

```
object:0100H-010DH
size  :000EH(   14)bytes
label :    2
error :    0   complete !
```

If an error occurs during the assembly, the computer displays the corresponding error message and the line number at which the error occurred. In this case, go back to the editor and correct the source program.

```
 ***** ASSEMBLER *****
*FORMAT ERROR        (1)
0105 ****             40
       LBL:   LD   HL),A
```

11.1.4.Check Generated Object Program

The generated object program is checked with the monitor. The program is stored from 0100H to 010DH.

Enter **MONITOR** mode. Press **CLS** (or **BASIC** **MON** ↵).

```
MACHINE LANGUAGE MONITOR
*
```

Display the object program with the D command: D0100↵. The computer displays the mapped dump of the object program.

```
0100 : 3E 20 21 00  > !.
(88)   04 77 3C 23  .w<#
       FE A0 C2 05  . Ã.
       01 C9 00 00  .È..
```

Note: Prior memory contents can be seen starting at 010DH (C9). (88) is the checksum.

11.1.5.Run Object (Machine Code) Program

Now the generated object program can be run. The monitor command G (GOSUB) is used.

Display the command prompt for **MONITOR** mode.
BREAK

```
*
```

Use the G command to run the object program: G0100↵
After execution, the command line of the monitor is displayed.

```
*G0100
*
```

The result of the program execution is checked: D0400↵

```
0400 : 20 21 22 23  !" #
(78)   24 25 26 27  $%&'
       28 29 2A 2B  ()*+
       2C 2D 2E 2F  ,-./
0410 : 30 31 32 33  0123
       34 35 36 37  4567
```

▼
The hexadecimal numbers 20H to 9FH were written to the address range 0400H to 047FH

```
0410 : 30 31 32 33   0123
(78)   34 35 36 37   4567
       38 39 3A 3B   89:;
       3C 3D 3E 3F   <=>?
0420 : 40 41 42 43   @ABC
       44 45 46 47   DEFG
```

11.2.Coding / Editing a Source Program

The assembler translates (assembles) the source program stored in the TEXT area into a machine code program. The assembled machine code program is sequentially loaded into a memory area starting at the specified address.

This section describes the conventions and rules (input formats, etc.) used when creating a source program.

11.2.1.Source Program Format

Each line of a source program usually contains a single statement. A program generally consists of a few lines. Assembly language source code begins with an `ORG` statement and ends with an `END` statement (the `ORG` and `END` statements can be omitted).

Example:

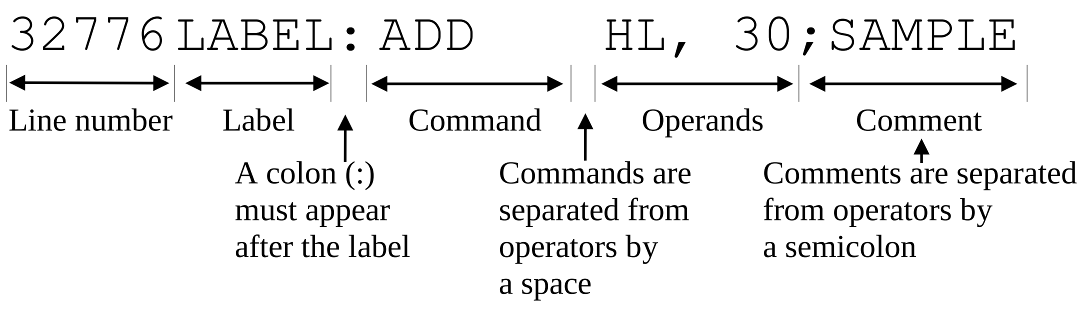
```
10    ORG   0100H
        ⋮
100   END
```

The `ORG` instruction is used to specify the first address of the memory area in which the generated machine code program is to be stored. This means that the lines of the machine code program are stored in order, starting from the address determined by the `ORG` instruction. If no address is determined, the computer uses 0100H as the first address. The `END` statement indicates the end of the source program. The computer stops assembling when it reaches this statement.

These instructions serve to control the assembler; they are not converted into a machine code.

11.2.2.Line Format (Instructions)

Each line of the source program consists of a line number, a label, a command, an operand, a comment, or a pseudo-command.



```
32776 LABEL: ADD    HL, 30;SAMPLE
```

Line number
Label
Command
Operands
Comment
A colon (:) must appear after the label
Commands are separated from operators by a space
Comments are separated from operators by a semicolon

One line can consist of up to 254 characters, including the comment. Small and capital letters are processed like capital letters, except when they are used in operands or comments.

- *Line Number*
  If a line number outside the allowable range of 1 to 65279 is entered, the error message "`LINE NO, ERROR`" will be displayed.
- *Label*:
  A label can be inserted directly after the line number (there must be no empty space between the line number and the label, otherwise an error will occur). Labels can consist of up to six characters. If there are more than six characters, an error occurs.

  The following characters can be used for labels:

  - Letters: A to Z (a to z are read like A to Z).
  - Numbers: 0 to 9
  - Symbols: [,], @, ?, and _

  The first character of a label must be a letter or a symbol (a number cannot be distinguished from the line number).

  A label using the same characters or pairs of characters as the following registers or condition codes cannot be used:

  1. Single register: A, B, C, D, E, H, L, I, R
  2. Register pairs: AF, BC, DE, HL, IX, IY, SP
  3. Condition code: NZ, Z, NC, C, PO, PE, P, M?

  A label must be followed by a colon (:), otherwise an error will occur. An exception is the definition of a value for a label with the pseudo-command EQU; in this case, no colon must follow the label.

  If no label is required, one or more spaces must be inserted between the line number and the following command word. To insert spaces, use the **SPACE** or **TAB** key.

- *Commands (OP code)*
  A Z80 command can be entered as a mnemonic symbol. Other pseudo-instructions can also be inserted here. A command is part of a statement called statement code or OP code.

  The command entered must be separated from the following operand by one or more spaces. To insert spaces, use the **SPACE** or **TAB** key.

- *Operand field*
  Operands are registers, addresses or constants used in executing instructions. Each operand can consist of up to 32 characters and are separated by commas (,). The following types of constants can be used as operands:

Numeric constants

Binary, decimal or hexadecimal numbers:

Binary	:	Represented as a sequence of 1 and 0, with a "B" at the end. Examples: 10111100B, 100000B
Decimal	:	Shown as base 0 to 9. Examples: 188, 32
Hexadecimal	:	Represented by decimal numbers 0 to 9 and the capital letters A to F; with an "H" in the end. If a hexadecimal number begins with a letter, it must begin with a "0" to distinguish it from a command. Examples: 0BCH, 20H

String constants

Character strings for operands must be in single quotes (') be included. ASCII representations of characters are used as constants in operands. For example:

(Specification)	(string)	(constants)
`'A'`	`A`	`41H`
`'FROM'`	`FROM`	`41H,42H`
`'B''C'`	`B'C`	`42H,27H,43H`
`'''D'`	`'D`	`24H,44H`
`'E'''`	`E'`	`45H,27H`
`''''`	`'`	`27H`
`''`	`(ZERO)`	`00H`

Label constants

If a constant is defined for a label with the EQU command, this label can be used as a constant in an operand. Expressions (including arithmetic operators) can be used as operands. The following characters and arithmetic operators can be used in operands, however, no operator takes precedence over another.

Signs: positive (+), negative (-)
Operators: *, /, +, -

The computer performs internal operations with 16-bit data. A capacity overrun is ignored (no error occurs). The object is generated with an 8-bit or 16-bit result.

For statements with expressions, the computer does not check for the correctness of the expression.

Examples: LD A, 4142H -> Read as LD A, 42H
DB 1234H -> Read as DB 34H?

- *Comments*
  Each line of a source program can be followed by a comment, separated by a semicolon (;). The part of the line from a semicolon to the end of the line is considered a comment and not translated into machine code (object),

11.2.3. Deleting a Source Program

Display the main menu in **TEXT** mode and press D to select the delete function. The computer asks for security whether the content of the TEXT area should be deleted. (If no program is stored in the TEXT area, the computer does not respond to pressing D.)

```
TEXT DELETE OK? (Y)
```

To delete all information in the TEXT area, press the Y key. The computer returns to the main menu of the TEXT mode. Pressing a button other than Y returns the computer to the TEXT mode main menu without erasing anything.

11.2.4. Entering a Source Program

Display the main menu in **TEXT** mode. Press E to select the edit function.

Pressing ▲ or ▼ will scroll the contents of the TEXT area, for example, a source program. If nothing is saved, the display does not change. A new program cannot be loaded into the TEXT area until the existing content has been completely deleted. Press BREAK to return to the main menu, select the Delete function, and clear the contents of the TEXT area.

Follow the steps described in the above section to delete a source program.

1. Enter the line number
2. If no label is required, one or more spaces can be inserted by pressing TAB or SPACE. The cursor moves back to the input field for commands.
   A label is entered immediately after the line number, without a space. The label must end with a colon (:). After the colon, one or more blanks can be inserted as desired.
3. Enter a command. If an operand follows the instruction, it must be separated from the instruction by one or more spaces (press .TAB. or .SPACE.).
4. Enter the operands. Operands are separated by commas (,).
5. If you want to annotate this line, a semicolon (;) must be entered before the comment.
6. After entering the entire line, press ↵ to save the line. The cursor disappears after ↵ is pressed.

To enter additional lines, the above steps are repeated.

11.3. Assembler Functions

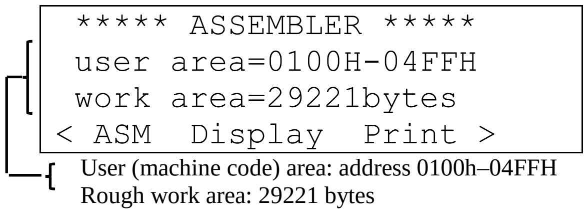
This section describes in detail how to assemble a source program entered in TEXT mode. This assumes the example program is already loaded in the computer.

11.3.1. Assembler Menu

In order to assemble a source program, the assembler has to be activated.

SHIFT + **ASMBL**

The assembler menu shown to the right appears.



```
 ***** ASSEMBLER *****
 user area=0100H-04FFH
 work area=29221bytes
< ASM  Display  Print >
```

User (machine code) area: address 0100h–04FFH
Rough work area: 29221 bytes

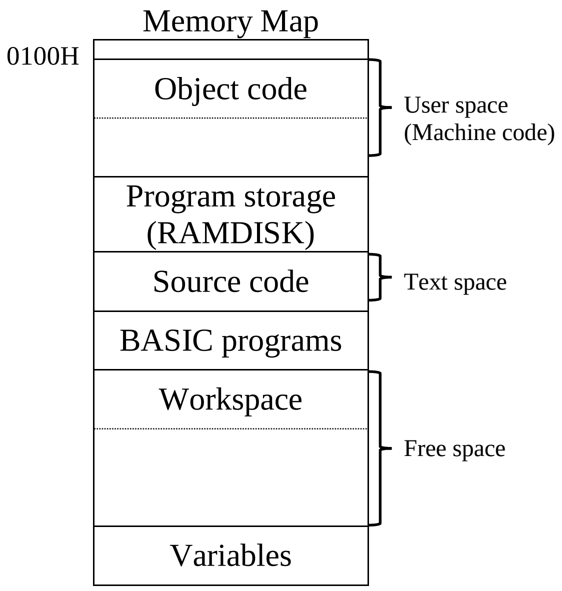
A : Assemble program
D : Display assembled program
P : Print assembled program

The menu shows the assigned machine code area on the second line. To assign the machine code area, the `USER` command is used in **MONITOR** mode. If no machine code area has been assigned or the area is too small to save the object, an error message (`NOT RESERVED` or `USER AREA OVER`) is displayed during assembly. In this case, use **BASIC** `MON` ↵ to select the monitor mode and assign or enlarge the machine code area with the `USER` command.

The third line of the assembler menu specifies the size of the existing work area in bytes. This shows the byte count of the free area in memory. The value corresponds to the number obtained with the `FRE` command from BASIC.

The workspace required for the conversion process is automatically assigned in free space. If the workspace cannot be assigned, a `WORK AREA OVER` error message will be displayed. In this case, increase the free space by deleting existing BASIC programs or other data, or reduce the machine code area.

Note: An error occurs if there are less than 307 bytes of free space while the computer is in **ASMBL** mode. If a source program contains labels, the assembler provides a label workspace with the necessary size. An error occurs if the assembler cannot assign this necessary area.



11.3.2. Assembling

Successful Assembly

To start assembling, press A while the assembler menu is displayed.

```
 ***** ASSEMBLER *****

  --- assembling ---
```

During operation, "--assembling--" is displayed. At the completion of the process, "complete !" is displayed as well as the object area, size of the object code, the number of labels, and the number of errors.

```
object:0100H-010DH
size  :000EH(   14)bytes
label :   2
error :   0   complete !
```

Press CLS to return to monitor mode. In **MONITOR** mode, you can check the assembled object program with the D command or have it executed with the G command.

Unsuccessful Assembly

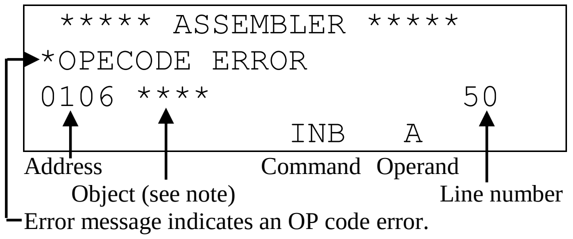
If an error is found in the source program during assembly, the assembler ends the process and displays a corresponding error message and the line number where the error was found. To continue assembling, press ▼.

For example, assume that the example program contains an error in lines 50 and 80:

```
50   INB  A          .....   "INC  A" is correct.
      ⋮
80   JP   NZ, KBL    .....   "JP NZ, LBL" is correct.
```

Press the A key while the assembler menu is displayed to assemble the program with the specified errors.

When the first error is found, the error message shown on the right is displayed



```
 ***** ASSEMBLER *****
*OPECODE ERROR
0106 ****             50
            INB    A
```

Note: When the assembler cannot generate correct object code because of an error in the source program, a series of asterisk will be displayed after the corresponding address.

Press ▼ to continue assembling. Now the error message for the second error is shown in line 80.

```
***** ASSEMBLER *****
*UNDEFINED SYMBOL
0109 ****             80
            JP     NZ,KB
```

Error message (an undefined symbol is used for a label)

Press ▼ again. The last screen of the assembler appears, but this time without the message "`complete !`".

```
object:0100H-010CH
size  :000DH(   13)bytes
label :   2
error :   2
```

At the last screen, press CLS to return to the assembler menu.

Notes: The assembler ignores the statement from line 50 and assumes that the label of line 80 specifies the address 0000H. At this point the example program is assembled.

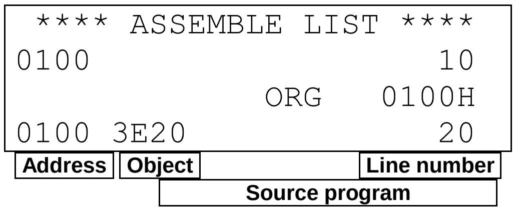
If an error is found in the source program, the generated object code also has errors. When executing the object program, the program may become out of control or destroy the memory contents. The source program must be corrected and reassembled, so that the object program can run without error.

Displaying the Object Code

With the display option, the object program can be checked before the source program is compiled. The assembler log contains the machine code program to be generated, its addresses and further object information.

Pressing D while the assembler menu is displayed displays the first line of the assembler log. Pressing ▼ will display subsequent lines for review. Load the above sample program if it is not already loaded and check its assembled object code.

Press D in **ASSEMBLER** mode.



```
 **** ASSEMBLE LIST ****
0100                  10
             ORG    0100H
0100 3E20             20
```

If the object field is empty, no object is generated. If there are more than 8 digits of machine code, the remaining digits are shown on the following line.

Press ▼ several times to see the subsequent lines of the assembler log.

```
 **** ASSEMBLE LIST ****
0100                  10
             ORG    0100H
0100 3E20             20
       START:LD     A,20H
0102 210004           30
             LD     HL,04
       00H
0105 77
       LBL:  LD     (HL),
       A
0106 3C               50
             INC    A
0107 23               60
             INC    HL
0108 FEA0             70
             CP     010H
010A C20501           80
             JP     NZ,LB
       L
010D C9               90
             RET
010E                 100
            END

 **** SYMBOL TABLE ****
START :0100 LBL   :0105
object:0100H-010DH
size  :000EH(   14)bytes
label :    2
error :    0   complete !
```

The values assigned to the labels are in hexadecimal.

Notes: Press CLS to return to the assembler menu.

You can check the assembler log with the display option. However, the object code of the source program cannot be loaded into the machine code area. To load the object code, the source program must successfully assembled.

Printing the Assembler Log

The assembler log can be printed out with the print command in the assembler menu. Connect the optional CE-126P printer to the computer, switch on the printer and press P while the assembler menu is displayed.

Notes: If the printer option is selected without the CE-126P printer connected or turned on, an error message will appear `(* PRINTER ERROR`). In this case clear the message with CLS and check the printer.

The assembler log is printed, regardless of whether **PRINT** is displayed in the lower right side of the display.

After printing, the assembler shows the final assembler screen. Press CLS to return to the assembler menu.

The log will be printed identically to how it is shown on the display.

To cancel the printout, press and hold the BREAK button until the printer stops. The display will show "`--break--`". Press CLS to go back to the assembler menu.

Sending the Assembler Log to the Serial Interface (SIO)

The assembler log can also be sent to the serial interface by entering L in the assembler menu. The operation is identical to the `Print` operation (see above).

Note: In contrast to the other assembler menu commands, the `L` command is not listed on the screen.

11.4. Assembler Pseudo-Instructions

Pseudo-instructions are used to control the assembler itself and are not converted into machine code. This computer knows the following pseudo-commands:

- `ORG`: Specifies the first address of the machine code area.
- `DEFB/DB/DEFM/DM`, `DEFS/DS,` and `DEFW/DW`: define data within the operand.
- `EQU`: define label values.
- `END`: indicates the end of the assembly program.

The following describes some of the conventions and rules used in the explanation of the pseudo instructions.

Expression:	Expressions can be numbers, formulas, labels, or "strings."
Formulas :	Formulas can be numbers, labels, or any arithmetic expressions that use numbers or labels.
{} :	When multiple elements are combined by a curly brace, only one of these elements can be selected.
[] :	An element within square brackets denotes an optional instruction.
[] ... :	Ellipses after square brackets indicate that the element is optional and can be repeated.

ORG — Beginning

Format: `ORG expression`

Description: Specifies the first address of the machine code area. The expression determines the first address of an area in which the generated machine code is stored. The machine code program is sequentially loaded into memory starting at the address determined by this expression.

If the source program does not contain an ORG statement, the assembler takes the statement 'ORG 100H'; This makes 100H the first address from which the machine code is stored.

Example: `ORG 0400H` This instruction stores machine code starting at address 0400H.

DEFB / DB / DEFM / DM — Define Byte / Message

Format: [Label:] $\left\{\begin{matrix}\text{DEFB}\\ \text{DB}\\ \text{DEFM}\\ \text{DM}\end{matrix}\right\}$ *expression* [, *expression*]…

Description: This instruction returns the least significant byte of a given number or expression converted to machine code.

Example: `DEFB 1234H`; translates 1234H to the machine code "34H".
`DB 1234`; translates 1234 into machine code "D2H".

A string in an operand must be enclosed in quotes (“). It can consist of up to 32 characters. Individual characters of an operand string are translated in the corresponding ASCII codes.

Example: `DEFM 'DATA'`; translates the individual characters of the sequence 'DATA' into the machine code 44H, 41H, 54H and 41H.

Individual operands are separated by commas (,).

Example: `DB 32w4+5,'X2'`; 85H, 58H and 32H are generated in machine code.

Sample Program:

```
                Source program              Machine code
10        ORG   0100H
20        LD    HL, DATA         21 0C 01
30        LD    DE, 300H         11 00 03
40        LD    BC, 5            01 05 00
50        LDIR                   ED B0
60        RET                    C9
70DATA:   DB    'ABCDEFGH'       41 42 43 44 45 46 47 48
80        END
```

The individual characters of the operand string in line 70 are translated into their corresponding ASCII codes. In the example program, five bytes of data are located in an area whose first address is specified by the label `DATA`; they are copied to an area starting with address 300H. This means that the data 41H, 42H, 43H, 44H and 45H are copied to the address 300H to 304H.

DEFW / DW — Define Word

Format: [Label:] $\left\{\begin{matrix}\text{DEFW}\\ \text{DW}\end{matrix}\right\}$ *expression* [, *expression*]…

Description: translates the two least significant bytes of a number or string expression (two characters or less) into machine code. Machine code bytes are ordered least significant, most significant.

Example: `DW 1234H`; translates 1234H to machine codes 34H and 12H (in order of least-significant and highest-value bytes).
`DEFW 34H`; 34H translates into machine codes 34H and 00H.

A string in an operand must be enclosed in quotation marks ('). You can define up to two characters for a string.

Example: `DEFW 'DA'`; translates the string 'DA' into 41H and 44H.
`DW 'Z'`; translates the string 'Z' into machine code 5AH and 00H.

Individual operands are separated by commas (,).

Example: `DW 'AB','CD',5678H`; translated into 42H, 41H, 44H, 43H, 78H and 56H.

DEFS / DS — Define Memory

Format: [Label:] $\begin{Bmatrix}\text{DEFS}\\ \text{DS}\end{Bmatrix}$ expression [, expression]…

Description: Generates the number of `NULL` codes (00H) specified in the operand. 00H is a "no operation code" (`NOP`) that instructs the computer to do nothing.

Example: `DS 12`; Generates 12 bytes with value 00H

Sample Program:

```
Source program                      Machine code
10        ORG   0100H
20        LD    HL, DATA            21 0C 01
30        LD    DE, 300H            11 00 03
40        LD    BC, 5               01 05 00
50        LDIR                      ED B0
60        RET                       C9
65        DS    4                   00 00 00 00
70DATA:   DB    'ABCDEFGH'          41 42 43 44 45 46 47 48
75nxt00:DS      500H-NXT00          (Inserts 00H at all subsequent
                                     addresses up to 04FFH.)
80        END
```

This example program is just like the above, but contains additional lines 65 and 75. Line 65 allocates a memory area for later use. With line 75, `NULL` codes (00H) are inserted to delete unnecessary memory contents.

EQU — Equal

Format: `[Label:] EQU` *`expression`*

Description: Assign value that is specified by the operand to label.

The label is assigned a value specified by the expression. Expressions may be a number or a string of one or two bytes. The colon (:) after the label is omitted.

Example: `START EQU 1000H`; assign the value 1000H to the label `START`. The label can then be used as a constant of value 1000H
`OK EQU 'Y'`; Assign the value 59H to the label `OK`.

END — End

Format: `END`

Description: Indicates the end of a source program.

The end of a source program is determined by the `END` statement. The assembler terminates the conversion process at this point. Information following this instruction will no longer be assembled. If there is no `END` statement at the end of a source program, the assembler assembles until the end of the TEXT area.

11.5. Error Messages

This section contains a list of error messages that may be displayed during assembly, as well as explanations of these messages. To clear the error message, press **CLS**. If assembly is aborted when an error occurs in the source program, the ▼ button can be pressed to resume assembly. The error message is also cleared when the computer is set to a different operating mode.

Error type	Description (cause)
`OPECODE ERROR`	Invalid OP code (command code),
`FORMAT ERROR (1)`	Invalid separator for operators
`FORMAT ERROR (2)`	Invalid code (ASCII code 01H-1FH or similar) or characters in an operand (such codes or characters cannot normally be entered).
`FORMAT ERROR (3)`	Invalid number of operands
`FORMAT ERROR (4)`	Invalid characters were used in a label.
`FORMAT ERROR (5)`	A label has more than six characters
`FORMAT ERROR (6)`	The string in the operand is not enclosed in quotes.
`FORMAT ERROR (7)`	The number of characters in an instruction or a single operand exceeds 32? (E.g., the value of the address or the like in an operand has too many leading zeros.)
`QUESTIONABLE OPERAND (1)`	Invalid operand.
`QUESTIONABLE OPERAND (2)`	Invalid condition (NZ, Z, NC or similar)
`QUESTIONABLE OPERAND (3)`	The value of the operand exceeds the permissible limit.
`QUESTIONABLE OPERAND (4)`	The string in the operand exceeds the permissible length of 32 characters.
`QUESTIONABLE OPERAND (5)`	Divide by zero.
`QUESTIONABLE OPERAND (6)`	Other invalid values ??or expressions.
`UNDEFINED SYMBOL`	An undefined symbol (label) was used.
`MULTI DEFINE SYMBOL`	The same symbol (label) has been defined more than once.
`FILE NOT EXIST`	The program to be assembled is not in the TEXT area.

Error type	Description (cause)
`USER AREA OVER`	The object could not be loaded into the machine code area. (The first address of the object area specified with the ORG instruction is outside the machine code area or the object has exceeded the capacity of the machine code area during loading,)
`WORK AREA OVER`	The size of the free area is too small for the necessary workspace to assemble (if the computer is in assembler mode or assembles).
`PRINTER ERROR`	The printer is not ready to start or does not work. (The printer is not connected, turned off or inoperable due to a discharged battery.)

12. PIC

The SHARP PC-G850V(S) has an interface for PIC devices (Peripheral Interface Controller). This allows these controllers to be programmed with the Pocket Computer.

The following devices are supported (as of 2001):

	program memory	number of pins
PIC16F627	1K words	18
PIC16F83	512 words	18
PIC16F84	1K words	18
PIC16F84A	1K words	18

PIC mode consists of two functions:

Assembler	: programs are created in `TEXT` mode and then assembled.
Loader	: transfer the object program into the PIC module

12.1. Defining the Machine Language Area

Make sure that enough memory is reserved in the machine language area.

For the PIC interface, the system requires more than 1K words. Therefore, at least 3KByte should be defined. Use the `USER` command to define a free area of 3K in the machine language monitor:

BASIC **MON** ↵
`USER0CFF`

(The memory block 0100H-0CFFH is now reserved)

```
MACHINE LANGUAGE MONITOR
*USER0CFF
FREE:0100-0CFF
*
```

12.2. Creating / Editing a Source Program

The source program is created or edited in the same way as Z80 or CASL assembler programs. The assembler programs must be written to conform with the MPLAB specification.

As with the other assembler languages, only one command per line may be written. A command line consists of a line number, label, command/opcode, operand, and comment.

```
<line number>[label] Opcode operand [; comment]
```

Example: `32767LABEL MOVLW    0x0F9   ; SAMPLE`

At least one space or TAB must be before and after each command. A line, including the comment, can be up to 254 characters long.

- *Line Numbers*
  Each line must contain a line number. If a line number outside the allowable range of 1 to 65279 is entered, the error message "`LINE NO, ERROR`" is displayed.
- *Label*
  The optional label must begin immediately after the line number. The length of the label is between 1 and 8 characters. Only alphanumeric characters (A-Z and 0-9) may be used. The label must start with a letter.
- *Opcode*
  The opcodes of the 35-bit 14-bit kernel are given here. These include the special assembler commands. As a delimiter to the operand, at least one blank or TAB must be entered.
- *Operand*
  One or more operands (separated by commas). The following types of constants are possible:

  Numeric constants

Decimal	:	includes 0–9. Examples: 188, 32
Hexadecimal	:	starts with 0x. Includes 0–9, A, B, C, E, D, F. Example 0xBC, 0x20.

  Character constant

  Character constants must be enclosed in single quotes (') be included. For example:

Character	Operand	Numeric Value
A	'A'	0x41
NULL	''	0x00

  Address constant

  The operand is a label, e.g. an EQU statement.

- *Comment*
  The optional comment must start with a semicolon. Until the end of the line, all subsequent characters are treated as a comment. These characters are not assembled, therefore do not belong to the object program.

12.3.PIC Assembler

The source program must be entered or loaded in the TEXT editor. Then change to PIC mode:

Press SHIFT + ASMBL and then P to enter **PIC** Mode

```
 *** PIC ASSEMBLER ***

 Assembler   Loader
```

To assemble the program press A.

```
 *** PIC ASSEMBLER ***

→Assembler   Loader

Complete! (***** words)
```

During assembly, "`Assembling ....`" appears in the lower left area. When the process is complete, the message "`Complete! (***** words)`" appears, where `*****` is the size of the program in words.

12.3.1.PIC Assembler Directives

The assembler has commands to control the assembler itself and to declare definitions. These commands are not part of the object program.

`__CONFIG`	:	Defining the configuration
`ORG`	:	Specify address for the beginning of the program
`EQU`	:	Define values
`DW`	:	Define data

__CONFIG — Configuration

Format: `__CONFIG` *`expression`*

Function: Configure PIC

Description: Configuration bit for each PIC. The values to be specified can be found in the documentation for the PIC module. According to MPASM specification, the bits can be linked with "&" (ampersand). However, this does not work with this computer.

Example: `__CONFIG 0x3FA8`

ORG — Set Start Address

Format: `ORG` *`address`*

Function: Define the start address of the program

Description: specifies the start address of the object program. If the `ORG` instruction is not specified, a start address 0x0 (`ORG 0`) is assumed. A value from 0x0 to 0x1FFF can be specified, depending on the requirements of the PIC module

Example: `ORG 0x0006`

EQU — Define a Constant

Format: `label EQU expression`

Function: Associates a label with a constant.

Description: The expression can be a numeric value or a character.

Example:

`START EQU 0x1000`	Defines the constant 0x1000 for `START`
`OK EQU "Y"`	Defines the value 0x59 for `OK`

DW — Define a Word

Format: `[label] DW` *`expression`*

Function: Define a word (2 bytes).

Description: The *expression* can be a numeric value or a character. Note that this is a 14-bit system and the values up to 0x3FFF are allowed.

Example: `DW 0x1234`

#INCLUDE — Insert a File

Format: `#INCLUDE "`*`file`*`"`

Function: Inserts a file for the PIC modules into the source program during assembly. This file contains standard definitions for the specific module.

Description: These files contain LABEL definitions of the MPASM specification for a specific module. The file specified in the operand must be enclosed in double quotes.

The following files can be used:

PIC modules: P16F627.INC, P16F83.INC, P16F84.INC, P16F84A.INC

14bit flash memory: PIC.INC

The labels defined by the #INCLUDE statement are not included as part of the 102 labels that can be defined by the user. Each program can contain only one #INLCUDE statement. This should be at the beginning of the program.

Labels in the MPASM specification that are longer than 8 characters are limited to 8 characters on this computer.

MPASM label	Label in Computer
OPTION_REC	OPTION_R
NOT_T1SYNC	NOT_T1SY

12.3.2.PIC Assembler Error Messages

Using the assembler may cause errors (see table). Press **CLS** to clear the error. You can the correct the error (for example, in the TEXT Editor).

Error message	Description
`File not exist!`	No program included in the TEXT Editor
`No USER AREA!`	No machine language area has been defined
`Not __CONFIG data!`	There is no __CONFIG directive
`Syntax error! (*****)`	Wrong __CONFIG_Parameter Wrong ORG parameter The EQU command has no label illegal memory address No Space / TAB / CR after the operand No Space / TAB / CR after the OPcode Operands separated with space / TAB. Wrong operand Wrong OPcode Wrong preprocessor command
`Out of range! (*****)`	Address, content is outside the permitted range.
`Undefined label! (*****)`	The specified label does not exist.
`Undefined line! (*****)`	The specified address is higher than that of the allocated memory
`Label too long! (*****)`	The label has more than 8 characters
`Out of memory! (*****)`	Address in the ORG command over 8K, the program has run out of memory, too many labels.
`Multi define! (*****)`	Only one #INCLUDE command may be included in the program. There are 2 or more identical labels
`Not include file! (*****)`	The specified include file is invalid.

12.4.PIC Loader

The loader transfers the successfully assembled PIC program from the machine language area to the PIC module.

Press **SHIFT** + **ASMBL** and then **P** to enter **PIC** Mode

```
 *** PIC ASSEMBLER ***

 Assembler    Loader
```

To load the program, press **L**.

```
 *** PIC ASSEMBLER ***

 Assembler   →Loader

Complete! (***** words)
```

During the transfer, "`Loading ....`" appears in the lower left area. When the process is complete, the message "`Complete! (***** words)`" appears, where `*****` is the size of the program in words.

12.4.1.PIC Loader Error Messages

When using the loader, errors may occur (see table). Press CLS to clear the error. Then you can correct the error (for example, in the TEXT Editor).

Error message	Description
`No USER AREA!`	No machine language area defined
`Not PIC data!`	PIC data size is 0
`Illegal PIC data!`	PIC data is larger than the machine language area. The word in the __CONFIG parameter is incorrect
`Connection error!`	The connection to the PIC module could not be established.
`Low battery!`	Weak battery was detected.
`Verify error!`	Error while comparing / checking the transmitted program
`Break!`	The transfer was aborted.

13. BASIC COMMAND GLOSSARY

The following pages contain a listing of the BASIC commands that you can use on the computer. Descriptions of the logical functions AND, OR, XOR, and NOT can be found in **Chapter 5: Logical Expressions.**

For simplicity, the following conventions have been adopted:

`expression, exp`	Indicates a numeric value, numerical variable, or a formula including numeric values and numerical variables.
`variable, var`	Indicates a numerical variable or string variable, including array variables.
`“string”`	indicates a character string enclosed in quotation marks.
`string-variable`	Indicates a string variable or string array variable.
`*label`	Indicates a *label (both *label and “label” forms may be used with this computer)
`d:`	Indicates a device name.
`[ ]`	The parameter in square brackets is optional. The brackets themselves are not part of the command entry.
`( )`	Used to enclosed parameter values in certain commands. They should be entered as part of the command.
`" "`	Used to enclose string parameter values in certain commands.
`A|B`	A or B can be selected
P	Program execution is possible
D	Direct input operation is possible
Abbr	Most of the commands can be abbreviated. The shortest abbreviation allowed is given in this manual. The following abbreviations are also valid: `PR.` `PRI.` `PRIN.`

13.1.Scientific and Mathematical Functions

The computer has a wide range of built-in function for scientific, mathematical and statistical calculations. All the functions listed below can be use as part of calculations when using the computer in **RUN** mode in addition to use in BASIC programs.

For trigonometric functions, entries can be made in degrees, radian or gradian values as appropriate:

`DEGREE`:	Set the computer to degree entry mode (the status line on the display shows **DEG**). This is the default mode.
`RADIAN`:	Set the computer to radian entry mode (the status line on the display shows **RAD**).
`GRAD`:	Set the computer to gradian entry mode (the status line on the display shows **GRAD**).

These three modes (**DEG**, **RAD** and **GRAD**) can also be set within a program. Once a mode is set, all entries for trigonometric functions must be in the units set (degree, radian or gradian values) until the mode is changed manually or from within a program. In the following examples, values for the trigonometric functions are in degrees.

Most functions can also be entered by pressing the corresponding function key.

It is not possible to perform manual calculations directly in **PRO** mode.

ABS $|x|$

Format: `ABS` *`expression`*

Function: Absolute value

Description: Returns the absolute value of the numeric argument. The absolute value is the value of a number, regardless of its sign.

Example: `ABS -10↵` `10`

ACS $\cos^{-1} x$

Format: `ACS` *`expression`*

See also: `ASN`, `ATN`, `COS`

Function: Inverse or arc cosine

Description: Returns the arc cosine of the *expression*.

The value of *expression* must be in the range of -1 ≤ *expression* ≤ 1. Since the arc cosine function is the inverse of the cosine function, the returned value is an angle. The result is dependent on the current angle mode (**DEG**, **RAD**, or **GRAD**) and falls in the following ranges:

Degrees: 0° … 180°
Radians: 0 … π
Gradians: 0 … 200

The corresponding key is $\boxed{\text{COS}^{-1}}$.

Example:

```
DEGREE↵
ACS -0.5↵                                            120
```

```
10:DEGREE
20:PRINT "arccos(0.5) =";ACS(.5);" degrees"
30:PRINT "arccos(0) =";ACS(0);" degrees"
40:END
>
RUN
arccos(0.5) = 60 degrees
arccos(0) = 90 degrees
>
```

AHC $\cosh^{-1} x$

Format: `AHC` *`expression`*

See also: `AHS`, `AHT`, `HCS`

Function: Inverse hyperbolic cosine

Description: Returns the inverse hyperbolic cosine of *expression*.

Example:

```
AHC 10↵                                      2.993222846
```

AHS $\sinh^{-1} x$

Format: `AHS` *`expression`*

See also: `AHC`, `AHT`, `HSN`

Function: Inverse hyperbolic sine

Description: Returns the inverse hyperbolic sine of *expression*

Example: `AHS 27.3↵` `4.000369154`

AHT $\tanh^{-1} x$

Format: AHT(*expression*)

See also: AHS, AHC, HTN

Function: Inverse hyperbolic tangent

Description: Returns the inverse hyperbolic tangent of *expression*.

Example: `AHT 0.7↵` `0.867300527`

ASN $\sin^{-1} x$

Format: ASN *expression*

See also: ACS, ATN, SIN

Function: Inverse or arc sine

Description: Returns the arc sine of *expression*.

The value of *expression* must be in the range of -1 ≤ *expression* ≤ 1. Since the arc sine function is the inverse of the sine function, the returned value is an angle. The result is dependent on the current angle mode (**DEG RAD**, or **GRAD**) and falls in the following ranges:

Degrees:	-90° … 90°
Radian:	-π/2 … π/2
Gradians:	-100 … 100

The corresponding key is $\boxed{\text{SIN}^{-1}}$.

Example:

```
DEGREE↵
ASN 0.5↵                                    30
```

```
10:DEGREE
20:PRINT "arcsin(0.5) =";ASN(.5);” degrees”
30:PRINT "arcsin(0) =";ASN(0);" degrees"
40:END
>
RUN
arccos(0.5) = 30 degrees
arccos(0) = 0 degrees
>
```

ATN $\tan^{-1} x$

Format: `ATN` *`expression`*

See also: `ACS`, `ASN`, `TAN`

Function: Inverse or arc tangent

Description: Returns the arc tangent of *expression*.

There is no restriction on the value of *expression*. Since the arc tangent function is the inverse of the tangent function, the returned value is an angle. The result is dependent on the current angle mode (**DEG**, **RAD**, or **GRAD**) and falls within the following ranges:

Degrees:	-90° … 90°
Radian:	-π/2 … π/2
Gradians:	-100 … 100

The corresponding key is $\boxed{\textbf{TAN}^{-1}}$.

Example:

```
DEGREE↵
ATN 1↵                                              45
```

```
10: DEGREE
15: WAIT 100
20: GOSUB 100
30: FOR DX = 0 TO 100
40: X = DX * .1
50: F = ATN (X): Z = Z + 1
60: IF Z = 3 THEN GOSUB 100
70: PRINT ""; STR$ (X), F
80: NEXT DX
90: END
100: CLS: PRINT "ARGUMENT", "ARC-TANGENT"
110: Z = 0: RETURN
```

COS — cos *x*

Format: `COS expression`

See also: `ACS`, `SIN`, `TAN`

Function: Cosine

Description: Returns the cosine of the angle *expression*

The specified angle can be in degrees, radians, or gradians. To obtain the correct value, the computer must be in the correct angle mode (**DEG**, **RAD**, **GRAD**). The corresponding key is **COS**

Example:

```
DEGREE↵
COS 120↵                                         -0.5
```

```
10:DEGREE
20:G$=CHR$ (&F8)
30:PRINT "cos(60";G$;") = ";COS(60)
40:PRINT "cos(90";G$;") = ";COS(90)
50:END
>RUN
cos(60°) = 0.5
cos(90°) = 0
>
```

CUB — x^3

Format: `CUB expression`

See also: `CUR`

Function: Cube

Description: Returns the cube of *expression.*

Example:

```
CUB 3↵                                            27
```

CUR — $\sqrt[3]{x}$

Format: `CUR expression`

See also: `CUB`

Function: Cube root

Description: Returns the cube root of *expression*.

Example:

```
CUR 125↵                                           5
```

DEG — dd°mm’ss” → ddd.dddd°

Format: `DEG` *`expression`*

See also: `DMS`, `VDEG`

Function: “degrees, minutes, seconds” (sexagesimal) to decimal conversion.

Description: Converts the angle argument in DMS (degrees, minutes, seconds) format to the DEG (decimal degrees) format. The angle to be converted must be in the form dd.mmssrr, where:

dd	: degrees
mm	: minutes
ss	: seconds
rr	: fractional seconds (00 ... 99)

The following ranges must be observed:

hh	: 0 to …
mm	: 00 to 59
ss	: 00 to 59
rr	: 00 to 99

The result is displayed with up to ten significant digits. The corresponding function key is **→DEG**.

Example: `DEG 30.5230`↵ (30°52'30") `30.875`

```
10:X = DEG 50.3000
20:PRINT X
30:END
> RUN
50.5
>
```

DMS ddd.dddd° → dd°mm’ss”

Format: `DMS` *`expression`*

See also: `DEG`, `VDEG`

Function: Decimal to "degrees, minutes, seconds" (sexagesimal) conversion

Description: Converts the angle *expression* in decimal degree format to "degrees, minutes, seconds" format.

The result of the angle conversion is in the form dd°mm’ss.rr”, where:

dd	: degrees
mm	: minutes
ss	: seconds
rr	: fractional seconds (00 ... 99)

The corresponding function key is [→DMS].

Example:

```
DMS 124.8055↵                        124.48198
```
(124°48'19.8")

```
10:X = DMS 50.5
20: PRINT X
30: END
>RUN
50°30’
>
```

EXP e^x

Format: `EXP` *`expression`*

See also: `LN`, `LOG`, `^`, `TEN`

Function: Exponential

Description: Returns the value of *e* (~2.718281828…the base of natural logarithms) raised to the value of *expression*.

Expression can be a numeric constant, variable or a numeric expression. The corresponding function key is [e^x].

Example:

```
EXP 1.2↵                             3.320116923
```

```
>PRINT EXP (10)
220026.46579
>
```

FACT *n*!

Format: `FACT` *`expression`*

Function: Factorial

Description: Returns the factorial of *expression*.

Example: `FACT 7↵` `5040`

```
>PRINT FACT (7)
>RUN
5040
>
```

FIX

Format: `FIX` *`expression`*

See also: `INT`

Function: Integer

Description: Returns the integer portion of *expression*. If *expression* is negative, `FIX` returns the first negative integer greater than or equal to *expression*.

Example: `FIX -8.4↵` `-8`

HCS cosh *x*

Format: `HCS` *`expression`*

See also: `AHC`, `HSN`, `HTN`

Function: Hyperbolic cosine

Description: Returns the hyperbolic cosine of *expression*.

Example: `HCS 3↵` `10.067662`

HSN sinh *x*

Format: `HSN` *`expression`*

See also: `AHS`, `HCS`, `HTN`

Function: Hyperbolic sine

Description: Returns the hyperbolic sine of *expression*.

Example: `HSN 4↵` `27.2899172`

HTN — tanh *x*

Format: `HTN` *`expression`*

See also: `AHT`, `HCS`, `HSN`

Function: Hyperbolic tangent

Description: Returns the hyperbolic tangent of *expression*.

Example: `HTN 0.9`↵ `0.71629787`

INT

Format: `INT` *`expression`*

See also: `FIX`

Function: Integer

Description: Returns the integer portion of *expression*. If *expression* is negative, `INT` will return the first negative integer smaller than or equal to *expression*.

Example: `INT -1.9`↵ `-2`

LN — $\log_e x$

Format: `LN` *`expression`*

See also: `EXP`, `LOG`, `^`, `TEN`

Function: Natural logarithm

Description: Returns the logarithm of the base *e* (~2.718281828 ...) of *expression*.

This function is the inverse of the `EXP` function. Any numerical expression is allowed, provided that its result is within the permissible value range. The corresponding function key is **ln**.

Example: `LN 2`↵ `0.69314718`

```
10:CLS: INPUT "ARGUMENT ="; X
20:PRINT "THE LOGARITHM TO THE BASE"
30:PRINT "e IS:"; LN (X)
40:INPUT "FURTHER CALCULATION (Y/N)"; A$
50:IF A$ = "Y" THEN 10
60:END
```

LOG $\log x$

Format: `LOG` *`expression`*

See also: `EXP`, `LN`, `^`, `TEN`

Function: Common logarithm

Description: Returns the common (base 10) logarithm of *expression*.

To obtain a logarithm in a base other than 10, e.g. for any base *B*, use the following formula:

$$log_B expression = \frac{\log expression}{\log B}$$

The inverse of the common logarithm can be calculated with the power operator (^), if one chooses the number 10 as the power base. The corresponding function key is $\boxed{\textbf{log}}$.

NCR ${}_nC_r = \frac{n!}{r!(n-r)!}$

Format: `NCR(`*`expression1`*`,`*`expression2`*`)`

See also: `NPR`

Function: Combination

Description: Returns the number of combinations of *expression2* elements out of a group of *expression1* elements. The corresponding key is $\boxed{{}_nC_r}$.

Example: `NCR(6,3)` ↵ `20`

NPR ${}_nP_r = \frac{n!}{(n-r)!}$

Format: `NPR(`*`expression1`*`,`*`expression2`*`)`

See also: `NCR`

Function: Permutation

Description: returns the number of permutations of *expression2* elements out of a group of *expression1* elements. The corresponding key is $\boxed{{}_nP_r}$.

Example: `NPR(6,3)` ↵ `120`

PI π

Format: `PI`

Function: π

Description: `PI` is a numeric pseudo variable that has the value of π. The use of `PI` is identical to the use of the $\boxed{\pi}$ key.

Example: `PI` ↵ `3.141592654`

POL $(x, y) \rightarrow (r, \theta)$

Format: `POL(`*`expression1`*`,`*`expression2`*`)`

See also: `REC`

Function: Rectangular to polar coordinate conversion

Description: Converts the rectangular coordinate pair (*expression1*, *expression2*) to polar coordinates. *Expression1* is the distance from the y-axis and *expression2* is the distance from the x-axis (the order is reversed). The distance and the angle in the polar coordinates are assigned to the fixed variables `Y` and `Z`, respectively. The value of converted angle depends on the angle mode (**DEG**, **RAD** or **GRAD**).

Example:

```
DEGREE↵
POL(8,6)↵                                   10 (r = 10)
Z↵                                  36.86989765
                                     (θ ≈ 36.9°)
```

```
10:X=POL (10,10)
20:PRINT X
30:PRINT Z
40:END
>RUN
14.14213562
45.
>
```

^ (Power) y^x

Format: *`expression1^expression2`*

Function: x^{th} power

Description: Returns *expression1* raised to the *expression2* power. The corresponding function key is $\boxed{y^x}$.

Example: `4^2.5↵` 32

RCP $^1/_x$

Format: `RCP `*`expression`*

Function: Reciprocal

Description: Returns the reciprocal of *expression*

Example: `RCP 4↵` 0.25

REC $(r, \theta) \rightarrow (x, y)$

Format: `REC(`*`expression1`*`,`*`expression2`*`)`

See also: `POL`

Function: Polar to rectangular coordinate conversion

Description: Converts the polar coordinate pair (*expression1*, *expression2*) to rectangular coordinates. *Expression1* is the distance from the origin while *expression2* is the angle. The angle can be in degrees, radians, or gradians. To obtain the correct value, the computer must be set to the correct angle mode (**DEG**, **RAD** or **GRAD**). The converted values indicate the distances from the x-axis and y-axis, and are assigned to the fixed variables `Y` and `Z`, respectively.

Example:

```
DEGREE↵
REC(12,30)↵                          10.39230485 (x ≈ 10.4)
Z↵                                             6  (y = 6)
```

```
10:X=REC (12,30)
20:PRINT X
30:PRINT Z
40:END
>RUN
10.39230485
6.
>
```

SGN

Format: `SGN `*`expression`*

Function: Sign

Description: Returns the sign of the *expression*. *Expression* can be any numeric expression.

If x > 0, the function returns 1.
If x < 0, the function returns -1.
If x = 0, the function returns 0.

Example:

```
5:WAIT 100
10:FOR N = -2 TO 2
20:PRINT N, SGN (N)
30:NEXT N
40:END
> RUN
-2      -1
-1      -1
0       0
1       1
2       1
>
```

SIN — sin *x*

Format: `SIN` *`expression`*

See also: `ASN`, `COS`, `TAN`

Function: Sine

Description: Returns the sine of *expression*. The specified angle can be in degrees, radians, or gradians. To obtain the correct value, the computer must be in the correct angle mode (**DEG**, **RAD**, **GRAD**). The corresponding function key is $\boxed{\text{SIN}}$.

Example:

```
DEGREE↵
SIN 30↵                                      0.5
```

```
10:DEGREE
20:G$ = CHR$(&F8)
30:PRINT "sin (30";G$; ") =";SIN(30)
40:PRINT "sin (45";G$; ") =";SIN(45)
50:END
>
RUN
sin (30°) = 0.5
sin (45°) = 7.071067812E-01
>
```

SQR — $\sqrt{x}$

Format: `SQR` *`expression`*

See also: `SQU`

Function: Square root

Description: Returns the positive square root of *expression*. The value of *expression* must be zero or positive. If *expression* is negative, `ERROR 22` is displayed. The corresponding function key is $\boxed{\sqrt{x}}$.

Example:

```
SQR 3↵                               1.732050808
```

SQU — x^2

Format: `SQU` *`expression`*

See also: `SQR`

Function: Square

Description: Returns the square of *expression*. The corresponding key is $\boxed{x^2}$.

Example:

```
SQU 4↵                                        16
```

TAN — tan x

Format: `TAN` *`expression`*

See also: `ATN`, `COS`, `SIN`

Function: Tangent

Description: Returns the tangent of *expression*. The specified angle can be in degrees, radians, or gradians. To obtain the correct value, the computer must be in the correct angle mode (**DEG**, **RAD**, **GRAD**). Because the value of `TAN` is undefined at certain angles, an error code will be displayed at these angles.The corresponding key is **TAN**.

Example:

```
DEGREE↵
TAN 45↵                                              1
```

```
10:DEGREE
15:WAIT 128
20:PRINT "ANGLE IS IN DEGREES!"
30:PRINT "ANGLE: 0, TANGENT:";TAN(0)
40:PRINT "ANGLE: 45, TANGENT:";TAN(45)
50:PRINT "ANGLE: 90, TANGENT:";TAN(90)
> RUN
ANGLE IS IN DEGREES!
ANGLE: 0, TANGENT: 0
ANGLE: 45, TANGENT: 1
ANGLE: 90, TANGENT:
ERROR 20 IN 70 (press CLS key!)
```

TEN — 10^x

Format: `TEN` *`expression`*

See also: `EXP`, `LN`, `LOG`, `^`

Function: Common antilogarithm

Description: Returns the value of 10 (the base of the common log) raised to the value of *expression*. The corresponding key is **10^x**.

Example:

```
TEN 3↵                                            1000
```

&H

Function: Hexadecimal to decimal conversion

Description: Converts a hexadecimal value to a decimal value.

Example:

```
&HF82↵                                            3970
```

13.2. General Commands

ASC

Format: `ASC("`*`string`*`"|`*`string-variable`*`)`

Abbr: `AS.`

See also: `CHR$`

Description: Returns the ASCII code

If *string* consists of more than one character, the ASCII code of the first character is returned. The relationship between the ASCII code and the associated character is shown in Appendix H: Table of Character Codes.

Example:

```
10: WAIT 0: CLS
20: PRINT "PLEASE ENTER A CHARACTER OR"
30: INPUT "ENTER A STRING:", S$
40: WAIT 100
50: PRINT "THE ASCII CODE IS:"; ASC (S$)
60: END
>RUN
PLEASE ENTER A CHARACTER OR STRING: SHARP
THE ASCII CODE IS: 83
>RUN
PLEASE ENTER A CHARACTER OR STRING:
       ⋮
```

AUTO

Format: `AUTO [[`*`line-number`*`] [, `*`increment`*`]]`

See also: `RENUM`

Description: automatic insertion of line numbers in **PRO** mode.

The `AUTO` command can be used to facilitate programming in **PRO** mode by automatically generating line numbers.

The starting line number and incremental value may be specified. If not specified, the computer automatically sets the first line number to 10 and the increment to 10. However, if the `AUTO` command has been previously set to other values, those values are used. An error is generated if the starting line number exceeds 65279.

When the mode is changed from **PRO** to **RUN** and then back to **PRO**, entering `AUTO` assumes the previously set increment and resumes line numbering from the most recently generated line number.

Pressing **SHIFT** + **CA**, turning the power off then on, or entering an operation mode other than **PRO** or **RUN** will exit `AUTO`.

Example:

```
AUTO                      10,20,30,40, ......
AUTO 100                  100,110,120, ......
AUTO 400,20               400,420,440, ......
```

BEEP

Format: `BEEP `*`number`*` [, [`*`tone`*`] [, `*`duration`*`]]`

Description: generates beeps of the specified tone and duration through the computer’s internal speaker..

number : determines how often the beep will sound. Specify a number or expression between 0 ... 65535.

tone : specifies the frequency of the beep in the range of 255 to 0. As the value of the tone parameter increases, the frequency drops. A value of 0 is about 7 kHz. A value of 255 is about 230 Hz

If this parameter is missing, the default frequency is approximately 4 kHz.

duration : determines the duration of a beep in the range of 0 to 65279. The beep duration varies with the tone parameter. A given duration value will give a relatively longer beep at low frequencies.

If the duration is omitted, a default value of 160 is set.

If the tone is omitted, the frequency of the beep is set to approximately 4kHx

BLOAD

Format: `BLOAD ["`*`filename`*`"]`

Abbr: `BLO.`

See also: `BLOAD?, BSAVE`

Description: loads a BASIC program from cassette to the computer. If `"`*`filename`*`"` is specified, the computer will search the tape for a program with the label `"`*`filename`*`"` then load the program. If `"`*`filename`*`"` is not found, the computer continues to search even if the end of the tape is reached. Press **BREAK** to stop searching.

`BLOAD` can also be used to load a BASIC program from another Sharp PC-G850 into memory via the serial (11-pin) interface.

The `BSAVE` must be entered simultaneously on the second Sharp.

Note: This transmission uses an internal protocol and is therefore not suitable for exchanging data between the Sharp PC-G850 and a PC. Likewise, the parameters for the serial interface in **TEXT** mode under **SIO** are ignored.

BLOAD M

Format: `BLOAD M [`*`start-address`*`]`

Abbr: `BLO. M`

See also: `BSAVE M`

Description: loads a machine code program from cassette to the computer. The program is loaded starting at *start-address* and overwrites any prior program stored at that address.

`BLOAD M` can also load machine code from another Sharp PC-G850 into memory via the serial (11-pin) interface.

`BSAVE M` must be entered simultaneously on the second Sharp.

Note: This transmission uses an internal protocol and is therefore not suitable for exchanging data between the Sharp PC-G850 and a PC. Likewise, the parameters for the serial interface in **TEXT** mode under **SIO** are ignored.

BLOAD?

Format: `BLOAD?`

Abbr: `BLO. ?`

See also: `BLOAD`

Description: compares a BASIC program from cassette with the BASIC program in memory. If `"filename"` is specified, the computer will search the tape for a program with the label `"filename"` then compare the program with the one in memory. If `"filename"` is not found, the computer continues to search even if the end of the tape is reached. Press **BREAK** to stop searching. If the program on cassette does not match the one in memory, an error message is displayed.

`BLOAD?` can also be used to compares the program in memory with a program from another Sharp PC-G850 with the program through the serial (11-pin) interface.

Note: This transmission uses an internal protocol and is therefore not suitable for exchanging data between the Sharp PC-G850 and a PC. Likewise, the parameters for the serial interface in **TEXT** mode under **SIO** are ignored.

BSAVE

Format: `BSAVE ["filename"]`

Abbr: `BS.`

See also: `BLOAD`

Description: saves a BASIC program to the cassette tape. If `"filename"` is specified, the program is saved to tape with and assigned the name `"filename"`.

`BSAVE` can also send a BASIC program to another Sharp PC-G850 via the serial (11-pin) interface.

The `BLOAD` command must be entered simultaneously on the second Sharp.

Note: This transmission uses an internal protocol and is therefore not suitable for exchanging data between the Sharp PC-G850 and a PC. Likewise, the parameters for the serial interface in **TEXT** mode under **SIO** are ignored.

BSAVE M

Format: `BSAVE M [`*`start-address`*`,`*`end-address`*`[,`*`start-address2`*`]]`

Abbr: `BS. M`

See also: `BLOAD M`

Description: saves a machine code program from the computer to a cassette. The program starting at *start-address* and ending at *end-address*. Is sent to the cassette.

`BSAVE M` can also send machine code to a second Sharp PC-G850 via the serial (11-pin) interface.

The `BLOAD M` command must be entered simultaneously on the second Sharp.

Transfer starts at address *start-address* and ends at address *end-address*. Optionally, the destination address (*start-address2*) can be specified.

Note: This transmission uses an internal protocol and is therefore not suitable for exchanging data between the Sharp PC-G850 and a PC. Likewise, the parameters for the serial interface in **TEXT** mode under **SIO** are ignored.

CALL

Format: `CALL [#`*`bank`*`,]`*`address`*

Abbr: `CA.`

See also: `PEEK`, `POKE`

Description: run a machine language program.

With `CALL`, a machine language program can be started from a BASIC program or in **RUN** mode and then returned to the calling mode.

bank : determines the memory bank from the range 0 … 7, where the machine language program is stored. If this parameter is not specified, memory bank 0 is used.

address : identifies the starting address of the program within the valid memory bank. The addresses must be in the range from 0 … 65535 (&0 … &FFFF). The address must be given and cannot be omitted.

CHR$

Format: `CHR$(`*`expression`*`)`

Abbr: `CH.`

Description: Returns the character that corresponds the character code of the expression.

The argument can be either a constant, a variable, or an expression. The argument must be an integer. A hexadecimal number can be specified with "&H" in front of the character code,(e.g. `A$=CHR$(&H5A)`).

See Appendix H for a table of character codes.

Example:

```
10: FOR X = 33 TO 126
20: PAUSE CHR$ (X);
30: NEXT X
40: END
```

CIRCLE

Format: `CIRCLE(`*`exp1`*`,`*`exp2`*`),`*`expr3`*`[,`*`exp4`*`,`*`expr5`*`,`*`exp6`*`[,S|R|X],[`*`exp`*`7]]]`

Abbr: `CI.`

See also: `LINE`

Description: Draws a circle.

The command can be used to draw circles, arcs, sectors, and ellipses with a solid line.

Exp1 and *exp2* specify the x and y coordinates, respectively, of the center of the circle. The origin (0,0) of the underlying coordinate system is located in the upper left corner of the display. The values must be in the range of -32768 to 32767. To specify points within the screen, use the following ranges:

exp1: 0 … 143
exp2: 0 … 47

Exp3 is used to specify the radius of the circle. The value of *exp3* must be within the range of 1 to 32767.

Exp4 and *exp5* are used to specify the starting and ending angle, respectively, of an arc or sector in degrees. The values must be within the range of -360 to 360. A value of 0 specifies the positive x-axis. Angles increase in a counterclockwise direction. If a negative value is specified, a radius is drawn from the origin to the arc. If a positive value is specified, this radius is not drawn. The default value for *exp4* is 0 degrees and that of *exp5* is 360 degrees.

Exp6 is used to specify the following ratio:

$$\text{ratio} = \frac{r_y\ (radius\ in\ y - axis)}{r_x\ (radius\ in\ x - axis)}$$

If the value of *exp6* is 1, a circle is drawn. If the value is other than 1, an ellipse is drawn. The default value of *exp6* is 1.

Options S, R, and X are used to set, reset, or reverse the pixel on the screen.

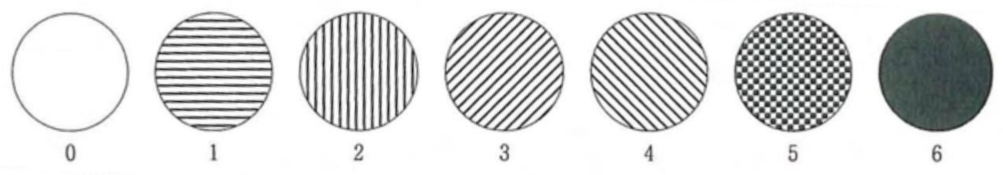
S: Draws a line while activating the corresponding dots on the screen (set).

R: Draws a line while deactivating the corresponding dots on the screen (reset). This option is useful in reverse video or to erase a line on the screen.

X: Draws a line, activating the corresponding dots if they are inactive, or deactivating the corresponding dots if they are already active. (reverse)

The default parameter is S.

Exp7 specifies a pattern for filling the circle. The value must be in the range of 0 to 6. Patterns are as follows:



Example:	`CIRCLE(71,23),20`	Simple circle with radius 20
	`CIRCLE(71,23),20,,,0.5,,2`	flattened circle with vertical fill
	`CIRCLE(71,23),20,-45,-135`	sector from 45° to 135°

CLEAR

Format: `CLEAR`

Abbr: `CL.`

See also: `DIM`, `NEW`, `ERASE`

Description: erase variables used in the program and resets all preallocated variables to zero or null

`CLEAR` recovers memory space used to store simple numeric variables and array variables secured using the `DIM` statement. It can also be used at the beginning of a program to clear space occupied by variables from previously run programs if several programs are in memory. Do not use the `CLEAR` command in a `FOR…NEXT` loop. Use the `ERASE` command to clear specific array variables.

Example:

```
5: WAIT 30                       Sets wait time for PRINT
10: DIM C(5)                     Dimensioned array C(N)
20: FOR N = 1 TO 5               These lines read the
30: READ A: LET C(N) = A           DATA values
40: PRINT C(N)                     and
50: NEXT N                         prints them
60: DATA 10,20,30,40,50          Provides the data
70: CLEAR                        Deletes all variables
80: PRINT A                      Verification of deletion
90: END
```

CLOSE

Format: `CLOSE [#`*`file-number1`*` [, #`*`file-number2`*`]…]`

Abbr: `CLOS.`

See also: `END, OPEN`

Description: closes all specified files.

`CLOSE` terminates the ability to access files. Without any parameters, `CLOSE` closes all open files. Specifying the parameter *file-number* will only close the file associated with that file number. The file number is then released for use with other files. All files are closed in the following cases:

- An end or run command is executed.
- The power is turned off.
- The computer is changed to an operation mode other than PRO or RUN.
- The program is written or read (by LOAD).

Example:

```
10: OPEN "E: PAYMENT" FOR INPUT AS #1
20: OPEN "E: UPDATE" FOR INPUT AS #2
       ⋮
400: CLOSE #1, #2
```

CLS

Format: `CLS`

See also: `LOCATE`

Description: clears the display.

Clears the display and resets the display start position to (0,0).

CONT

Format: `CONT`

Abbr: `C.`

Description: continues a program that was temporarily halted.

Enter `CONT` to continue a program that was stopped under the following conditions:

- Abort by STOP instruction
- Abort by actuation of the BREAK key
- Interruption by PRINT instruction

Example:

```
10: PRINT "PROGRAM STOP HERE"
20: STOP
30: PRINT "PROGRAM CONTINUED"
40: PRINT "PROGRAM ENDED
50: END
>RUN
PROGRAM STOP HERE
BREAK IN 20
>CONT
PROGRAM CONTINUED
PROGRAM ENDED
>
```

DATA

Format: `DATA` *`list-of-values`*

Abbr: `DA.`

See also: `READ`, `RESTORE`

Description: provide values for use by `READ`.

When assigning initial values to an array, it is convenient to list the values in a `DATA` statement and use a `READ` statement in a `FOR…NEXT` loop to load the values in the array. When the first `READ` is executed, the first value in the first `DATA` statement is returned. Successive `READ`s use the succeeding values in the sequential order in which they appear in the program, regardless of how many values are listed in each `DATA` statement or how many `DATA` statements are used.

A `DATA` statement may contain any numeric or string values, separated by commas. Enclose string values in quotes. Spaces at the beginning or end of the string should be enclosed in quotes.

`DATA` statements have no effect if encountered in the course of regular program execution, so they can be inserted wherever appropriate. Many programmers include them after the `READ` that uses them. If desired, the values in a `DATA` statement can be read a second time sing the `RESTORE` statement.

Example:

```
10: FOR I = 1 TO 5
20: READ N
30: PRINT N
40: NEXT I
50: END
60: DATA 10,20,30,40,50
```

DEGREE

Format: `DEGREE`

Abbr: `DE.`

See also: `RADIAN`, `GRAD`

Description: Set the angle mode to decimal degrees.

In this mode, all angular data is assumed to be in decimal degrees. To mark this, the symbol **DEG** appears in the status line. All arguments of the functions `SIN`, `COS` and `TAN` and the results from `ASN`, `ACS` and `ATN` are in degrees. The `DMS$` and `DEG` functions can be used to convert angles from decimal degrees to degrees, minutes, second form and vice versa.

Example:

```
10: DEGREE
20: PAUSE "ANGLE DATA IN DEGREES"
30: PRINT ASN (0.5), ASN (1)
40: PRINT ACS (0.5), ACN (1)
50: PRINT ATN (0.5), ATN (1)
60: END
```

DELETE

Format: `DELETE [line number][-][line number]`

Abbr: `DEL.`

See also: `NEW`, `RENUM`

Description: deletes the specified lines of a BASIC program.

`DELETE <line number>`
Deletes the specified line if it exists in the program.

`DELETE <line number> -`
Deletes program lines from the given line to the end of the program.

`DELETE <line number> - <line number>`
Deletes all lines of a program, starting with the first and ending with the second line. The second line number must be greater than the first named number.

`DELETE - <line number>`
Deletes all lines of a program starting from the beginning of the program up to and including the specified line.

To completely delete a program, the command `NEW` should be used.

Example:

```
DELETE 150

DELETE 50-150

DELETE -35
```

DIM

Format: `DIM variable[$](`*`index1`*`[,`*`index2`*`])[*`*`string-length`*`][,]…`

Abbr: `D.`

See also: `CLEAR`, `ERASE`, `READ`

Description: reserves space for numeric and string array variables.

`DIM` is used to reserve space for an array variable. The size of the array is the number of elements in that array.

The variable name consists of up to 2 alphanumeric characters starting with a letter. For string variables, "$" is attached to the end of the variable name. With the exception of the standard variables A to Z and A$ to Z$, which are equivalent to the two one-dimensional arrays A(1) to A(26) and A$(1) to A$(26), all array variables are sized with DIM to provide sufficient space in memory. If an array is not dimensioned, then it cannot be used.

index1 and *index2* are called "subscripts" and specify the number of elements in the n^th dimension of the array. An array with one subscript is called a one-dimensional array, with 2 subscripts, a two-dimensional array. Values of *index1* and *index2* are restricted to the range 0…255. The number of elements in the array is (*index1* + 1) * (*index2* + 1).

string-length determines the length of the string for the string arrays. However, if the strings have more characters than specified with *string-length*, they will be truncated to the appropriate size and all extra characters will be lost. If *string-length* is omitted, strings can contain up to 16 characters by default. The maximum string length is 255 characters.

Once an array has been dimensioned, it cannot be resized unless the computer is reset or a `CLEAR`, `NEW`, `RUN`, or `ERASE` command is performed. A running program aborts with the display of an `ERROR` code when it either encounters an array not declared with `DIM` or it tries to re-dimension a previously sized array. Indexes that exceed the maximum values set with *index1* or *index2* also lead to a program termination. Negative indices are illegal.

Example:

```
10: DIM C(13)
20: DIM F$(10)
30: DIM H(4,6)
40: DIM G$(7,5) *25
```

END

Format: `END`

Abbr: `E.`

See also: `STOP`

Description: Signals the end of a program.

The program will be terminated when the `END` statement is executed. Statements after the `END` statement in the same line cannot be executed. All opened files are closed.

If the `END` statement is missing, the program ends with execution of the last program line.

Example:

```
10: GOSUB 50
20: PRINT "AFTER REPEAT ENDS THE"
30: PRINT "MAIN PROGRAM WITH LINE 40"
40: END
50: PRINT "THIS IS THE SUB-PROGRAM"
60: RETURN
>RUN
THIS IS THE SUB-PROGRAM
AFTER REPEAT ENDS THE MAIN PROGRAM WITH LINE 40
```

EOF

Format: `EOF (`*`file-number`*`)`

Abbr: `EO.`

Description: determines is the end of a sequential file is reached.

`EOF` returns a value that indicates whether all the data in a sequential file specified by *file-number* has been read.

If all data has been read, `EOF` returns -1 (true) as its value. If not, `EOF` returns 0 (false). An error occurs if a file with the specified *file-number* has not been opened for input.

ERASE

Format: `ERASE` *`variable1|array1[, variable2|array2]…`*

Abbr: `ER.`

See also: `CLEAR`

Description: erases specified variables and arrays

`ERASE` deletes specified simple variables and string variables. Only numeric variables and string variables that are not standard variables (A to Z or @(1) to @(26) and A$ to Z$ or @$(1) to @$(26)) are valid arguments for `ERASE`.

The `ERASE` statement cannot delete individual elements of an array. The whole array is cleared and its memory area is freed. Arrays are specified with empty parenthesis (). To resize an array, first `ERASE` it then re-define it with a `DIM` statement.

Example: `10: ERASE AB, Z$()`

FILES

Format: `FILES`

Abbr: `FI.`

See also: `LFILES`

Description: returns a list of the files on the RAM disk (Disk E)

`FILES` displays the filename, filename extension, and file length on the RAM disk. File length is measured in bytes. The filename extensions are:

.BAS	:BASIC programs
.TXT	:assembler, C, CASL programs

FOR … NEXT

Format:

```
FOR variable=expression1 TO expression2 [STEP
  expression3]
      ⋮
NEXT variable
```

Abbr: F. N. STE.

Description: repeats a series of operations a specified number of times.

FOR and NEXT are used in pairs to enclose a group of statements that are to be repeated. The first time this group of statements is executed the loop variable (the variable named immediately following FOR) is assigned its initial value (*expression1*).

When execution reaches the NEXT statement, the loop variable is increased by the STEP value (*expression3*) and then this value is tested against the final value (*expression2*). If the value of the loop variable is less than or equal to the final value, the enclosed group of statements is executed again, starting with the statement following FOR. If *expression3* is omitted, the increment becomes 1. If the value of the loop variable is greater than the final value, execution continues with the statement that immediately follows NEXT. Because the comparison is made at the end, the statements within a FOR…NEXT pair are always executed at least once.

When the increment is zero, FOR…NEXT will continue in an infinite loop.

The loop variable may be used within the group of statements, for example as an index to an array, but care should be taken in changing the value of the loop variable.

Write programs so that the program flow does not jump out of a FOR…NEXT loop before the counter reaches the final value. To exit a loop before it has been repeated the specified number of times, set the loop variable higher than the final value

The group of statements enclosed by a `FOR…NEXT` pair can include another pair of `FOR…NEXT` statements that use a different loop variable as long as the enclosed pair is completely enclosed; i.e., if a `FOR` statement is included in the group, the matching `NEXT` must also be included. `FOR…NEXT` pairs may be nested up to six levels deep. Illegally jumping out of an inner loop will generate an `ERROR`.

Do not use `CLEAR`, `DIM`, or `ERASE` within a `FOR…NEXT` loop.

Example:

```
10: FOR I = 1 TO 20
50: NEXT I
230: FOR K = 2 TO 17 STEP 2
290: NEXT K

10: FOR M = 1 TO 10
20: FOR N = 5 TO 20 STEP 5
80: NEXT N
90: NEXT M

10: A = 2: B = 5
20: FOR I = A TO B STEP 0.2
30: NEXT A
```

FRE

Format: `FRE`

Abbr: `FR.`

Description: Returns the free space available in the program data area in bytes.

`FRE` indicates the byte count of the free space (not occupied by program, array variables, or simple variables) in the program and data area of memory. As a function, `FRE` can pass its value to a variable.

GCURSOR

Format: `GCURSOR(`*`expression1, expression2`*`)`

Abbr: `GC.`

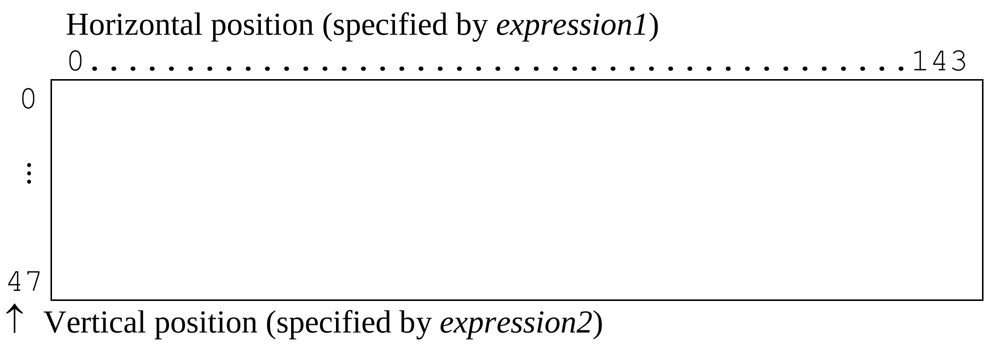
See also: `GPRINT`

Description: specifies starting point on the display.

`GCURSOR` specifies the starting point on the display for the dot pattern to be displayed by the `GPRINT` command.

The display consists of 144 columns and 48 rows of dots, which can be addressed by column numbers 0..143 and row numbers 0…47. Any dot on the screen can therefore be addressed as a starting point by specifying the column number with *expression1* and row number with *expression2*.

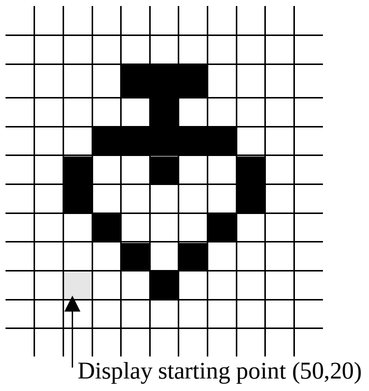
The values of *expression1* and *expression2* may range from -32768 to 32767. If the value of *expression1* is outside 0…143 or that of *expression2* is outside 0…47, the display starting point will become a virtual point outside of the screen boundaries.



Horizontal position (specified by *expression1*)
0...143
0
⋮
47
↑ Vertical position (specified by *expression2*)

Example:

```
5: CLS
10: GCURSOR(50,20)
20: GPRINT"1824458F452418"
```



GOSUB … RETURN

Format:

```
GOSUB line-number|"label"|*label
   ⋮
RETURN
```

Abbr: `GOS. RE.`

See also: `GOTO, ON…GOTO, ON…GOSUB`

Description: diverts program execution to a subroutine.

A subroutine is a group of consecutive program lines that are executed several times in the course of the program. The group of statements is included in the program at some location that is not reached in the normal sequence of execution. A common location is following the `END` statement that marks the end of the main program.

At each location in the main body of the program where a subroutine is to be executed, include a `GOSUB` statement with a line number or a label that indicates the starting point of the subroutine. The last line of each subroutine must be a `RETURN`.

When `GOSUB` is executed, the computer transfers control to the indicated line number or label and processes the statements until a `RETURN` is reached. Control is then transferred back to the statement following the `GOSUB`.

Subroutines may be “nested” with a maximum depth of 10 levels deep. If the depth is greater than this limit, the program is aborted and an `ERROR` code `50` is shown on the display.

Since there is an `ON…GOSUB` structure for choosing different subroutines at given locations in the program, the expression in a `GOSUB` statement usually consists of just the desired line number or label.

Example:

```
10: GOSUB 90
20: GOSUB "A"
      ⋮
90: "A" PRINT "SUB-PROGRAM STARTED"
95: RETURN
```

GOTO

Format: `GOTO` *`line-number`*`|"`*`label`*`"|*`*`label`*

Abbr: `G.`

See also: `GOSUB`, `ON…GOTO`, `CONT`

Description: Transfers program control to the specified *line-number* or *label*

`GOTO` performs a non-conditional jump to the specified line number or label. The jump automatically executed and does not depend on any condition (unless one uses the instruction `IF…THEN…GOTO`).

If a line containing the commands `DATA` or `REM` is specified as the jump destination, program execution continues at the next line (or executable instruction).

In **RUN** mode, `GOTO` can also be used to start a program from a specific line. Unlike the `RUN` command, no variables are deleted.

`GOTO` can also be used to resume a program that has been interrupted with the **BREAK** key.

Example:

```
10: INPUT A$
20: IF A$ = "Y" THEN 40
30: PRINT "NO": GOTO 50
40: PRINT "YES"
50: END
```

GPRINT

Format:

```
GPRINT string
GPRINT expression [;expression]…
GPRINT
```

Abbr: `GP.`

See also: `GCURSOR`

Description: displays the specified dot pattern.

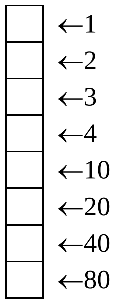
The `GPRINT` command displays the specified dot pattern. Each column of a bit image is represented by 8 vertical dots. The height of a column of pixels is the height of a character in TEXT mode.

If `GPRINT` is followed by a string, each 8 dot column is divided into a lower and upper group of 4 dots. Each group of dots is then represented by a hexadecimal number. Each pair of hexadecimal numbers represents one 8 dot column, with the first number representing the lower 4 dots and the second number representing the upper 4 dots. The string is enclosed by " ".

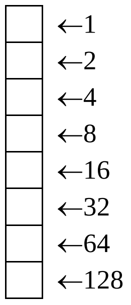
`GPRINT "`**`XX`**`XX`**`XX`**`XX"`

Hex number	Dot pattern	Hex number	Dot pattern	Hex number	Dot pattern	Hex number	Dot pattern
0	□□□□	4	□□■□	8	□□□■	C	□□■■
1	■□□□	5	■□■□	9	■□□■	D	■□■■
2	□■□□	6	□■■□	A	□■□■	E	□■■■
3	■■□□	7	■■■□	B	■■□■	F	■■■■

The vertical 8-dot pattern can be specified using a hexadecimal or decimal value. A "weight" is assigned to each dot as shown below.



Weight of each dot (hexadecimal)



Weight of each dot (decimal)

Specify the dot pattern with a numeric value equal to the sum of the "weights" of the dots to be displayed. The value is a number between 0 and 255.

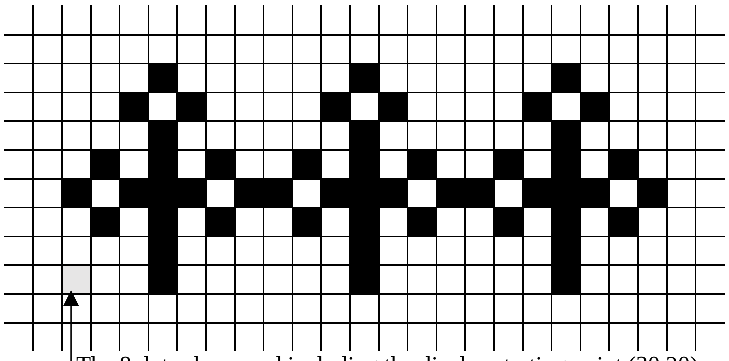
The following instructions are identical in their effect:

`GPRINT 16;40;18;253;18;40;16`	(decimal)
`GPRINT &10;&28;&12;&FD;&12;&28;&10`	(hexadecimal)
`GPRINT "102812FD122810"`	(hex string)

If no dot pattern is specified, the graphic cursor is moved down one line without affecting the contents of the display. If a `GPRINT` statement ends with a semicolon (;), the next `GPRINT` command takes effect from the next cursor position (the ";" concatenates the commands).

Example:

```
10: AA$ = "102812FD122810"
20: GCURSOR(30,20)
30: GPRINT AA$;AA$;AA$
```


The 8 dots above and including the display starting point (30,20) specified by the `GCURSOR` command are used to display the first value given in `GPRINT`

GRAD

Format: `GRAD`

Abbr: `GR.`

See also: `DEGREE`, `RADIAN`

Description: Sets the angle mode to gradian.

In this mode, all angular data is assumed to be in gradians. To mark this, the symbol **GRAD** appears in the status line. All arguments of the functions `SIN`, `COS` and `TAN` and the results from `ASN`, `ACS` and `ATN` are in gradians. Gradian form represents the angular measurement in terms of percent gradient, i.e. a 45° angle is a 50 percent gradient.

Example:

```
10: GRAD
20: PAUSE "ANGLE IN GRADIANS"
30: PRINT ASN (0.5), ASN (1.0)
40: PRINT ACS (0.5), ACS (1.0)
50: PRINT ATN (0.5), ATN ( 1.O)
60: END
```

HEX$

Format: `HEX$(`*`number`*`)`

Abbr: `H.`

See also: `VAL`

Description: converts a decimal number into its hexadecimal character string equivalent.

The value of the expression must be an integer in the range of -9999999999 to 9999999999. The resulting hexadecimal character string will be up to 10 digits long. `HEX$(64)` returns the string: "&40"

Example:

```
10: PRINT "CHANGE: DECIMAL TO HEX"
20: INPUT "DECIMAL NUMBER ="; X
30: IF X> 65535 THEN 100
40: IF X <0 THEN 110
50: PRINT "HEXADECIMAL VALUE ="; HEX$(X): PRINT
60: INPUT "ONE NUMBER (Y/N)"; A$
70: IF A$ = "Y" THEN 20
80: IF A$ = "N" THEN END
90: GOTO 60
100: PRINT "ERROR: MAXIMUM = 65535!": GOTO 20
110: PRINT "ERROR: MINIMUM = 0!": GOTO 20
120: END
```

IF … THEN … ELSE

Format:

```
IF condition THEN line-number|*label|statement
  [ELSE line-number|*label|statement[:statement]…]
```

Abbr: `IF T. EL.`

Description: controls program flow, depending on whether a condition is fulfilled or not.

The decision depends on the *condition* to be checked between the words `IF…THEN`. If this is true, the line specified after `THEN`, which is either a *line-number*, **label*, or *statement*, is executed. Otherwise, the next line will be executed.

If the instruction `IF…THEN` contains an `ELSE` statement, then if the condition is false, the program does not continue with the next line, but with the line or instructions specified after `ELSE`.

If `ELSE` is not followed by a line number or label, all statements (including those separated by a colon) are executed as long as they are on the same line.

The instructions `IF…THEN…ELSE` can also be nested within a program line.

The *condition* to be tested between the `IF…THEN` is formed by a logical expression which, however complex, can always be built from the following forms:

`X = Y`	:X is equal to Y
`X <> Y`	:X is not equal to Y
`X < Y`	:X is less than Y
`X > Y`	:X is greater than Y
`X <= Y`	:X is less than or equal to Y
`X >= Y`	:X is greater than or equal to Y
`X AND Y`	:Logical AND
`X OR Y`	:Logical OR
`NOT y`	:Logical NOT

Examples of logical expressions:

`X = 1`

Condition is fulfilled if `X` has the value 1.

`(P = 2 AND Q = 4) OR P = 1`

The condition is fulfilled if either `P` has the value 1 (independent of `Q`) or if `P` = 2 and `Q` = 4.

Example:

```
10: INPUT "SHOULD I SQUEEK", A$
20: IF A$ = "N" THEN 60
30: IF A$ = "Y" THEN BEEP 3: GOTO 10
40: PRINT "Y OR N ENTER!"
50: GOTO 10
60: PRINT "SORRY!"
70: END
```

IF…THEN…ELSE…ENDIF

Format:

```
IF condition THEN
    statement1
  [ELSE
    statement2]
ENDIF
```

Abbr: `IF T. EL. ENDI.`

Definition: conditionally executes statements at runtime.

When the *condition* of the `IF` statement is true, the statement after `THEN` is executed; if it is false, the statement following `ELSE` is executed. Program execution continues at the statement after `ENDIF`.

`IF`, `ELSE`, `ENDIF` must always follow directly after a line number, not a label

An instruction, expression, or remark should not follow the same line after `THEN` (or `ELSE`). Otherwise, the statement is treated like a normal `IF…THEN…ELSE` statement.

The use and interpretation of the conditional expressions conforms to the `IF…THEN…ELSE` statement.

Example:

```
10: WAIT:CLS
20: INPUT “COORDINATE”;A
30: LOCATE 14,0:INPUT “COORDINATE”;B
40: IF (4*A)<B OR (2*A)>B THEN
50:  PRINT "IMPOSSIBLE"
60: ELSE
70:  C=B-INT(B/2)*2
80:   IF C=1 THEN
90:    PRINT "ODD"
100:  ELSE
110:   X=(2*A)-B/2:Y=(B/2)-A
120:   WAIT 0:PRINT “ROW”;X
130:   WAIT:PRINT “COLUMN”;Y
140:  ENDIF
150: ENDIF
160: GOTO 10
```

INKEY$

Format: `INKEY$`

Abbr: `INK.`

See also: `INPUT`

Description: Gives the specified variable the value of the key pressed while the `INKEY$` function is executed.

`INKEY$` is used to respond to the pressing of individual keys without waiting for the ↵ key to end the entry. The `INKEY$` command reads the **SHIFT** or **CAPS** key if it is pressed. Thus it is unable to read the function or symbol key that is pressed following either of these keys. See the following table for the list of applicable keys and the characters that are returned. If no key is pressed, a value of 0 is returned.

Example:

```
300: A$ = INKEY$
310: IF A$ = "" THEN 300
320: IF A$ = "*" THEN 500
330: GOTO 300
          ⋮
500: PRINT "HI"
```

		0	16	32	48	64	80	…	128	144	…	240
	Hi / Lo	0	1	2	3	4	5		8	9		F
0	0		2ND F	SPACE	0		P					
1	1				1	A	Q			ln		
2	2	CLS			2	B	R			log		
3	3				3	C	S					
4	4	▲			4	D	T					
5	5	▼	CAPS		5	E	U			sin		
6	6				6	F	V			cos		
7	7	ANS	BS		7	G	W		1/x	tan		
8	8	BASIC	R-CM	(	8	H	X		x^2			
9	9	TEXT	M+	)	9	I	Y					
10	A	TAB		*		J	Z					
11	B	INS		+	;	K				→DEG		π
12	C	CONST		‘		L				F↔E		√
13	D	↵		—	=	M				nPr		
14	E	▶		.		N				MDF		
15	F	◀		/		O						

INPUT

Format:

```
INPUT variable [, variable]…
INPUT "prompt", variable [[,"prompt"], variable]]…
INPUT "prompt"; variable [[,"prompt"]; variable]]…
```

Abbr: `I.`

Description: allows keyboard entry of values.

Executing an `INPUT` instruction stops the program and displays *prompt* on the display, if specified in the instruction. The display of the question mark can be suppressed by adding a semicolon (;) after *prompt*. If *prompt* is missing, a question mark will appear.

During this pause in the program, data can be entered via the keyboard. The received data is assigned sequentially to the *variable* listed in the parameter. The variables in the list separated by commas. Entry is completed by pressing the ↵ key.

In all the cases just described, the cursor is positioned after the question mark or *prompt*. However, if *prompt* is followed by a comma, the cursor will move to the first column and erase *prompt*.

Example:

```
10: INPUT A
20: INPUT "A ="; A
30: INPUT "A =", A
40: INPUT "X =?"; X, "Y =?"; Y
```

INPUT#

Format: `INPUT# file-number, variable [, variable]…`

Abbr: `I.#`

See also: `DIM`, `INPUT`, `OPEN`, `PRINT`

Description: reads data from a file

`INPUT#` reads data from a file that resides on the RAM disk or from the serial interface.

File number is the number of the file that was assigned to it when it was opened by the `OPEN` command. An attempt to read an unopened file ends with an `ERROR`. For reading from the serial interface, the file number is 1. For RAM disk files, either 2 or 3.

The list of variables determines the names of the variables into which the data records are to be read. Variables can consist of simple variables, standard variables or arrays. The data format must match the order and type of the variable list. String variables must be dimensioned to the appropriate length. Arrays must have a pseudo-index (*) in the variable list, for example: `A(*)`. Comma, space, `LF`, `CF`, or `CR + LF` are used as delimiters when reading data into numeric variables. If the data file starts with a double quotes ("), all data up to the next comma is assigned to one variable.

Example:

```
10: A$ = "AB" + CHR$(34)+ "CDE" + CHR$(34)
20: B$ = CHR$(34) + "CD, EF" + CHR$(34)
30: PRINT A$
40: PRINT B$
50: OPEN "E:ABC.DAT" FOR OUTPUT AS #2
60: PRINT #2, A$; ","; B$
70: CLOSE #2
80: OPEN "E:ABC.DAT" FOR INPUT AS #2
90: INPUT #2, C$, D$
100: PRINT C$
110: PRINT D$
120: CLOSE: END
```

KILL

Format: `KILL "`*`filename`*`[.BAS]"`

Abbr: `K.`

See also: `SAVE`

Description: deletes a BASIC program

`KILL` deletes basic programs stored on the RAM disk. *filename* determines which file is to be deleted. The extension `.BAS` is optional. It is not possible to specify the name of the RAM disk (E:) or other devices. The use of "wildcards" (* or ?) is not allowed.

All other file types created or deleted via the **TEXT** monitor.

Example: `KILL "TEST"` This instruction deletes the basic program `TEST` from the RAM disk

LCOPY

Format: `LCOPY` *`startline`*`,` *`endline`*`,` *`targetline`*

Abbr: `LC.`

Description: copy lines.

`LCOPY` copies lines of BASIC programs from *startline* to *endline* to *targetline*. Line numbers for jumps in BASIC commands are not adjusted (unlike `RENUM`).

LEFT$

Format: `LEFT$(`*`string, number`*`)`

Abbr: `LEF.`

See also: `LEN`, `MID$`, `RIGHT$`

Description: returns the specified number of characters starting from the left end of the string.

`LEFT$` returns a substring of length <number> characters of the given *string* starting from the left.

The number of characters of the substring must be in the range of 0 to 255. A fractional number will be rounded to the nearest whole number. If the number is greater than the number of characters of the given string, the entire string is returned.

Example:

```
10: X$ = "SHARP"
20: FOR N = 1 TO 6
30: TS$ = LEFT$(X$,N)
40: PRINT TS $
60: NEXT N
>RUN
S
SH
SHA
SHAR
SHARP
SHARP
SHARP
```

LEN

Format: `LEN(`*`string`*`)`

See also: `LEFT$`, `MID$`, `RIGHT$`

Description: number of characters in a string

`LEN` determines the length of a string, i.e. the number of characters contained in it. This number also takes into account spaces and non-printable characters, such as control codes, e.g. a "carriage return" (symbol: `<CR>`, code: `&OD`).

Example:

```
10: INPUT "ENTER A WORD:", W$
20: N = LEN(W$)
30: PRINT "THE WORD HAS ";N;" LETTERS"
40: END
```

Notice what happens when W$ has more than 16 characters.

```
10: A$ = "ONE"; B$ = "TWO"; C$ = "THREE"
20: S$ = A$ + CHR$(13) + B$ + CHR$(7) + C$
30: PAUSE S$
40: PRINT "NUMBER OF CHARACTERS ="; LEN(S$)
50: END
>RUN
ONE TWO THREE
NUMBER OF CHARACTERS = 14
>
```

LET

Format: `[LET]` *`var1e`*`=exp1[, var2=exp2]…`

Abbr: `LE.`

Description: variable assignment

Assigns values to variables. Numeric variables can only be assigned numeric values and string variables can only be strings. The command word `LET` is optional and can be omitted. This makes the following two assignments identical: `LET A = 5` or simply: `A = 5`

`LET` must be used if variable assignment occurs immediately after a `THEN` or `ELSE`.

LFILES

Format:	`LFILES`
Abbr:	`LF.`
See also:	`FILES`

Description: prints a list of files

`LFILES` prints a list of the files on the RAM disk (Disk E:), i.e. a table of contents, on the connected printer. Each file is displayed with the following information:

Filename	Extension	(for example: `.BAS` for BASIC programs, `.TXT` for assembler, C, CASL)

LINE

Format:	`LINE [(`*`exp1`*`, `*`exp2`*`)]-(`*`exp3`*`, `*`exp4`*`)[,S|R|X][, `*`exp5`*`] [,B|BF]`
Abbr:	`LIN.`
See also:	`CIRCLE`

Description: draw a line or a rectangle.

`LINE` draws a line or rectangle from the first point with the coordinates (*exp1*, *exp2*) to the second point with the coordinates (*exp3*, *exp4*) on the display. The origin (0,0) of the underlying coordinate system is located in the upper left corner of the display.

If the first point is omitted, the current position of the graphic cursor is assumed.

The values of expressions1-4 should be between -32768 to 32767. To specify points within the screen, use the following ranges:

exp1 and *3*: 0 to 143
exp2 and *4*: 0 to 47

Options S, R, and X are used to set, reset, or reverse the pixel on the screen.

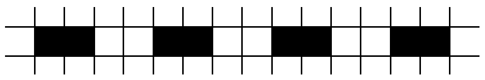
S: Draws a line while activating the corresponding dots on the screen (set).

R: Draws a line while deactivating the corresponding dots on the screen (reset). This option is useful in reverse video or to erase a line on the screen.

X: Draws a line, activating the corresponding dots if they are inactive, or deactivating the corresponding dots if they are already active. (reverse)

The default parameter is S.

Exp5 is used to specify a line pattern. The value of *exp5* ranges from 0 to 65535. (&0- &FFFF). This number represents a bit pattern. For example, if the value of *expn5* is 26214 (&H6666), the following line pattern is drawn:


16 dots

The binary representation of 26214 (&H6666) is: 0110011001100110. A 1 represents an active dot while a 0 represents an inactive dot. A solid line is drawn if *exp5* is omitted.

Options B and BF are used to draw a rectangle whose opposite corners are specified by (*exp1*, *exp2*) and (*exp3*, *exp4*).

B: draws an empty rectangle
BF: draws a filled rectangle.

Example:

```
10: CLS
20: FOR N = 10 TO 100 STEP 30
30: M = N + 20
40: LINE (N, 10) - (M, 20),BF
50: NEXT N
60: END

10: LINE -(124,31)

10: LINE (24,0)-(124,47),&HF18F,B

10: LINE (34,3)-(114,44),X,BF
```

LIST

Format: `LIST [line-number|"label"]`

Abbr: `L.`

See also: `LLIST`

Description: output lines of a BASIC program on the display.

If *line-number* or "*label*" is not specified, `LIST` starts at the first line of the program and displays the following program lines until the bottom of the display. The cursor is positioned invisibly behind the first line number.

Additional lines can be displayed by moving the cursor downwards with the $\boxed{\text{M}}$ key and the lines at the top of the display are scrolled off.

If *line-number* or "*label*" is specified, the list starts with that line. If there is no line with this line number, the list is started with the line having the next highest number. If *line-number* is greater than the highest line number in the program or if the specified label is not found, an `ERROR` code is displayed.

A program protected using `PASS` cannot be listed because access to the **PRO** mode is blocked in this case. The `LIST` command is only accepted in **PRO** mode.

LLIST

Format: `LLIST [`*`line-number1`*`|"`*`label1`*`"][-[`*`line-number2`*`|"`*`label2`*`"]]`

Abbr: `LL.`

See also: `LIST`

Description: outputs lines of a BASIC program.

If *line-number* or “*label*” is specified, `LIST` starts the list with the first line of the program. Although `LLIST` is used in much the same way as the `LIST` command, it is more flexible.

`LLIST`
lists the complete program, i.e. all lines of the program.

`LLIST` *`line-number`*
lists only the desired line

`LLIST` *`line-number1`*`|"`*`label1`*`"-`*`line-number2`*`|"`*`label2`*`"`
lists from *line-number1* or *label1* to *line-number2* / *label2*.

`LLIST` *`line-number`*`|"`*`label`*`"-`
lists from the specified *line-number* or *label* and continues until the end of the program.

`LLIST -`*`line-number`*`|"`*`label`*`"`
lists from the first line of the program to the specified line number / label.

Example:

```
LLIST

LLIST 10-100

LLIST 10-"A"

LLIST "A" -
```

LNINPUT#

Format: `LNINPUT#` *`file-number`*`,` *`string-variable1`*`[,` *`string-variable2`*`]…`

Abbr: `LNI.#`

Description: reads data from a file

This command reads data from a file on the RAM disk or the serial interface.

File-number is the number assigned to the file when opened with the `OPEN` command. An attempt to read an unopened file will display an `ERROR` code. File number 1 is used for reading from the serial interface. For RAM disk files, use either 2 or 3.

The list of string variables are the names of the variables into which the data are to be stored. Variables may consist of simple string variables, standard variables or arrays. The variables should be dimensioned to appropriate length. Arrays must have the pseudo-index (*) in the variable list, for example: `A(*)`. A `CR+LF` is used as a delimiter when reading data.

Example:

```
10: LNINPUT #2, AA$
```

```
10: LNINPUT #2, AA$, AB$, AC$
```

```
10: DIM AA$(4)*16
20: LNINPUT #2, AA$(*)
```

Reads 5 records

LOAD

Format: `LOAD "`*`filename`*`[.BAS]"`

Abbr: `LO.`

See also: `RUN`, `SAVE`

Description: load a file

Loads a file on the RAM disk into internal memory. *filename* determines which BASIC file to load. The extension `.BAS` is optional. Additionally, the name of the RAM disk (E:) or other device is also optional.

All open files are closed by `LOAD`.

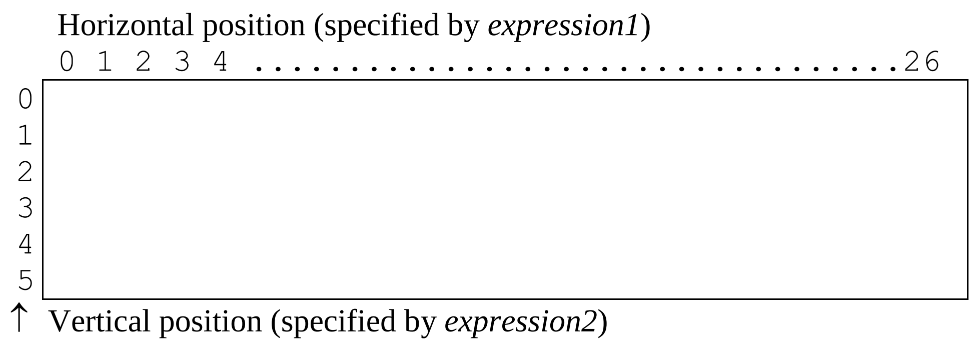
LOCATE

Format: `LOCATE [`*`expression1`*`[, `*`expression2`*`]]`

Abbr: `LOC.`

Description: specifies the display start position in column units.

Specifies the start position of the display in units of character position for the contents displayed by the `PRINT` command. The display position is defined as follows:


Horizontal position (specified by *expression1*)
0 1 2 3 4 26
0
1
2
3
4
5
↑ Vertical position (specified by *expression2*)

A position on the display is specified by its horizontal and vertical position. *Expression1* specifies the horizontal position while *expression2* specifies the vertical position. The range of *expression1* is 0 to 39. The range of *expression2* is 0 to 5. An error occurs if the expressions are not in the specified range.

If *expression1* or *expression2* is omitted, the current position is assumed.

Example:

```
10: LOCATE 5
20: PRINT "TEXT1";
30: LOCATE, 4
40: PRINT "TEXT2";
50: LOCATE 0.3
60: PRINT "TEXT3"
```

LOF

Format: `LOF(`*`file-number`*`)`

Description: returns the size of the specified file.

The LOF command returns the size of a file with the specified *file-number*. The size of the file is displayed in bytes. The file must be opened via the `OPEN` command and assigned a file number. If the specified file is not open, an `ERROR` will occur.

This file number is used to address the file whose size is to be determined.

Example:

```
10: OPEN "E: FILE1.TXT" FOR INPUT AS #2
20: N = LOF(2)
30: PRINT "FILE1 CONSISTS OF"; N; "BYTES"
40: CLOSE #2
50: END
```

LPRINT

Format: `LPRINT [USING "`*`format`*`"] [`*`expression1`*`|`*`string1`*`[,|;`
`[`*`expression2`*`|`*`string2`*`]…][;]`

Abbr: `LP.`

See also: `PRINT`

Description: sends output to the printer.

The commands `LPRINT` and `LPRINT USING` are used in the same way as `PRINT` and `PRINT USING`.

`LPRINT` without parameters feeds the paper by one line.

`LPRINT` with parameters prints the values of the listed expressions one after the other. These expressions can be either numeric or a string. If a semicolon is used to separate the expressions, their values print immediately after each other. If a comma is used, the value of the next expression is printed at the next column (For the CE-126P, the column starts at position 13 or 0)

If the list of expressions ends with a semicolon, the following `LPRINT` continues at the next position. If there is no semicolon at the end of the statement, a line feed is sent.

`LPRINT USING` behaves identically to `PRINT USING.`

MID$

Format: `MID$(`*`string`*`,`*`expression1`*`,`*`expression2`*`)`

Abbr: `MI.`

See also: `LEN`, `LEFT$`, `RIGHT$`

Description: returns a string of characters from inside another string

`MID$` returns a string of *expression2* characters from inside *string* starting from the *expression1* character in the string.

expression1	value in the range 1...255. Values outside of the range results in an `ERROR`. If the value is greater than the number of characters contained in the string, a `NULL` string is generated.
expression2	value in the range 0...255. determines how many characters from the given *string* are to be copied. Values with decimal places are rounded down to the nearest whole number.

Example:

```
10: Z$ = "ABCDEFG"
20: Y$ = MID$(Z$,3,4)
30: PRINT Y$
>RUN
CDEF
>
```

MON

Format: `MON`

Abbr: `MO.`

Description: Switches to the machine language monitor.

NEW

Format: `NEW`

See also: `DELETE`, `CLEAR`

Description: Clears existing programs and data

The `NEW` command clears all programs and data that are in memory. Password protected programs cannot be deleted.

ON…GOSUB

Format: `ON` *`expression`* `GOSUB` *`line-number1`*`|"`*`label1`*`",` *`line-number2`*`|"`*`label2`*`",…`

Abbr: `O. G.,O. GOS.`

See also: `GOSUB`, `GOTO`, `ON…GOTO`

Description: Execute on of a set of subroutines depending on the value of a control expression.

When `ON…GOSUB` is executed, the expression between `ON` and `GOSUB` is evaluated and reduced to an integer. If the value of the integer is 1, control is transferred to *line-number1* / "*label1*" in the list, as in a normal `GOSUB`. If the expression evaluates to 2, then control is transferred to *line-number2* / "*label2*", and so forth.

If the expression is zero, negative, or larger than the number of line numbers provided in the list, no subroutine is executed and execution proceeds with the next statement or line of the program.

Use commas (,) to separate line numbers or labels in the list.

Example:

```
10: INPUT "NUMBER (1-3) ="; N
20: ON N GOSUB 100,200,300
     ⋮
90: END
100: REM FIRST SUBPROGRAM
     ⋮
190: RETURN
200: REM SECOND SUBPROGRAM
     ⋮
290: RETURN
300: REM THIRD SUB-PROGRAM
     ⋮
380: RETURN
```

ON…GOTO

Format: `ON` *`expression`* `GOTO` *`line-number1`*`|"`*`label1`*`",`*`line-number2`*`|"`*`label2`*`",…`

See also: `GOSUB`, `GOTO`, `ON…GOSUB`

Description: Transfer control/executes one of a set of subroutines, depending on the value of a control expression.

When `ON…GOTO` is executed, the expression between `ON` and `GOTO` is evaluated and reduced to an integer. If the value of the integer is 1, control is transferred to *line-number1* / "*label1*" in the list. If the expression evaluates to 2, then control is transferred to *line-numbe2* / "*label2*", and so forth.

If the expression is zero, negative, or larger than the number of line numbers provided in the list, execution proceeds with the next statement or line of the program.

Use commas (,) to separate line numbers or labels in the list.

Example:

```
10: INPUT A
20: ON A GOTO 100,200,300
30: GOTO 900
100: PRINT "FIRST"
110: GOTO 900
200: PRINT "SECOND"
210: GOTO 900
300: PRINT "THIRD"
310: GOTO 900
900: END
```

OPEN

Format: `OPEN "E:`*`file`*`" FOR INPUT|OUTPUT|APPEND AS #file-number>`
`OPEN "COM:|COM1:|LPRT:|PIO:"`

Abbr: `OP.`

See also: `CLOSE`, `END`

Description: opens a file or I/O on a device for reading, writing, or appending.

`OPEN "E:`*`file`*`"` opens a file on the RAM disk. The `OPEN` command must be accompanied by an appropriate attribute (`INPUT`, `OUTPUT` or `APPEND`). The file can then be accessed for that purpose. Before a file can be opened for another purpose, it must first be closed with `CLOSE`.

file	:indicates the complete file name, including the extension.
`INPUT`	:allows sequential reading data records from the file using `INPUT#` or `LNINPUT#`.
`OUTPUT`	:allows data to be sequentially written to the file with `PRINT#`. In this mode, any previously written information in the file is lost.
`APPEND`	:allows data to be added to the end of the file with `PRINT#`.
file-number	:may only have the value 2 or 3. All other input and output commands, such as `PRINT#` or `LNINPUT#`, uses this number. This also means a maximum of 2 RAM disk files may be opened at the same time.

Files cannot be created with `OPEN`. Files must first be created in **TEXT** mode under `RFILES`. Additional file administration must be performed in **TEXT** mode.

Example:

```
10: OPEN "E:DATA.TXT" FOR OUTPUT AS #2
20: FOR J = 1 TO 5
30: PRINT #2, J
40: NEXT J
50: CLOSE #2
60: OPEN "E:DATA" FOR INPUT AS #2
70: IF EOF(2) THEN 110
80: INPUT# 2, J
90: PRINT J
100: GOTO 70
110: REM FILE END REACHED
120: CLOSE #2
130: END
```

PAINT

Format: `PAINT (`*`expression1`*`, `*`expression2`*`), `*`expression3`*

Abbr: `PAI.`

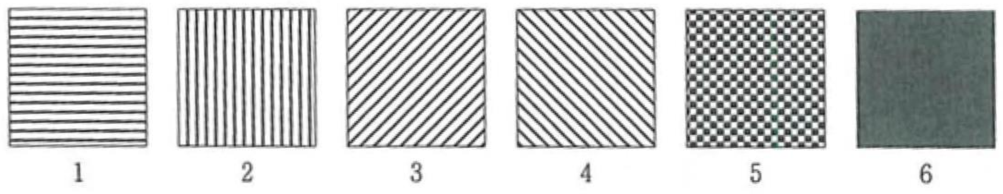
See also: `CIRCLE`, `GCURSOR`, `LINE`

Description: fills an area with a pattern.

`PAINT` fills an area surrounding the coordinates *expression1*, *expression2* with the pattern specified by *expression3*. The values must be in the range of -32768 to 32767. To specify points within the screen, use the following ranges:

expression1	: 0 (left) … 143 (right)
expression2	: 0 (top) … 47 (bottom)

If an off screen point is specified, the `PAINT` command is ignored. The fill pattern is specified by *expression3*. Legal values for *expression3* are:



Example: `>PAINT (71,23),3`

PASS

Format: `PASS "`*`character-string`*`"`

Abbr: `PA.`

Description: sets and cancels passwords.

The `PASS` command protects a program against unauthorized access by assigning a password. *Character-string* consists of up to eight arbitrarily combined alphanumeric characters, which are enclosed in quotation marks like a string constant. The quotation mark (") cannot be used within the password.

Once a password has been set, the computer can no longer be put into **PRO** mode. The following commands remain as ineffective as well as the ▲ and ▼ buttons:

```
AUTO    RENUM   LCOPY   LIST   LLIST
BSAVE   SAVE    BLOAD   LOAD   NEW      DELETE
```

The program cannot be listed, saved or changed. Similarly, overwriting by loading another program is prevented. If there are several programs in memory, protection applies to all programs. The only way to remove the protection is to re-enter the `PASS` command with the correct password.

The `PASS` command is only applicable if there is actually a program in program memory.

Example: `>PASS "SECRET"` This command protects all programs stored in memory with the password "SECRET".

PEEK

Format: `PEEK(`*`address`*`)`

Abbr: `PE.`

See also: `POKE`, `CALL`

Description: returns the contents of the specified memory address.

address is in the range of &0…&FFFF (0…65535).

Example:

```
A = PEEK(100)
A = PEEK(&H4001)
```

POINT

Format: `POINT(`*`expression1, expression2`*`)`

Abbr: `POI.`

Description: returns status of the specified point.

The arguments *expression1* and *expression2* can be any numeric expression. The values of *expression1* and *expression2* determine the display point. If the point is set, then `POINT` returns a value of 1, in all other cases the value is 0. If the specified point is outside the display boundary, the command returns 0.

The values of *expression1* and *expression2* may be within the range of -32768 ... 32767. A point within the display boundaries is addressed only with the value of *expression1* is from 0…143 and the value of *expression2* is from 0…47.

POKE

Format: `POKE address, byte[, byte]…`

Abbr: `POK.`

See also: `CALL`, `PEEK`

Description: write value into memory

`POKE` provides direct access to the memory of the computer. It is thus possible to write data in the form of bytes to the specified RAM addresses.

address	: determines in which memory address the (first) byte is to be written. The address must be in the range of 0…65535 or &0…&FFFF.
by*te*	: specifies an 8-bit value in the range of 0 to 255 (&0…&FF) to be written to the memory specified by *address*.

If several bytes are listed, which must be separated by commas, they will be written consecutively into consecutive addresses. The parameter *address* acts as start address. If the available memory space is insufficient for all listed bytes, an `ERROR` code will be displayed.

Example: `POKE &FF00, &13, &B7, &37, &C9`

PRESET

Format: `PRESET(expression1, expression2)`

Abbr: `PRE.`

See also: `PSET`, `LINE`

Description: clears the pixel at the specified coordinates

The arguments *expression1* and *expression2* can be any numeric expression. The value of *expression1* and *expression2* must be in the range from -32768 … 32767. A pixel on the screen is only addressed if the value of *expression1* is from 0 (left) – 143 (right) and the value of *expression2* is from 0 (top) – to 47 (bottom).

PRINT

Format: `PRINT [USING "`*`format`*`"] [`*`exp1`*`|`*`string1`*`[,|;` *`exp2`*`|`*`string2`*`]…][,|;]`

Abbr: `P.`

See also: `LPRINT`, `USING`

Description: displays information.

`PRINT` sends the values of the listed expressions one after the other. These expressions can be either numeric or a string. If a semicolon is used to separate the expressions, their values are displayed one after each other.

BASIC divides each line of the display into two equal zones of 12 columns. If a comma is used as the delimiter, the value of the subsequent expression is displayed at the next zone (column 1 or 13).

Numeric data is displayed right justified, but strings are displayed left-justified. Single numeric data is displayed in the right zone, single string data in the left zone.

A `PRINT` statement without any parameter displays a blank line, which is equivalent to a printer line feed.

The `USING` statement allows formatting of printed data. The format is determined by a *format* string, which is used as a parameter of the `USING` statement. See `USING` for additional details.

`PRINT`→`LPRINT`

The computer can switch all `PRINT` commands to `LPRINT`. Attach the printer before executing the following statements:

`PRINT = LPRINT`	: redirects the PRINT command to the printer.
`PRINT = PRINT`	: resets PRINT output to the screen.

PRINT#

Format: `PRINT# file-number,variable1[,|;variable2]…[,|;]`

Abbr: `P.#`

Description: write data to a file or the serial interface.

File-number is the number under which the file was opened using the `OPEN` command. Attempting to write something to an unopened file results in an `ERROR` code. This value must be 1 (serial), 2, or 3.

If the variables are delimited by semicolons, they are written to the file without a space. If a comma is used as a delimiter, the data is separated into zones of 20 characters. If a comma needs to be written to the file, it should be put in quotation marks.

Data can include both numeric and string variables. In both cases, attention should be paid to the use of the correct separators. Otherwise, there may be problems with reading using `INPUT#`. With array variables, individual elements can be addressed. The entire array must be specified in the form A(*).

PSET

Format: `PSET(expression1,expression2)[,X]`

Abbr: `PS.`

See also: `PRESET`, `LINE`, `CIRCLE`

Description: sets or clears a point at the specified coordinates.

The arguments *expression1* and *expression2* can be any numeric expression. The values of *expression1* and *expression2* must be in the range from -32768 … 32767. A point on the screen is only addressed if the value of *expression1* is from 0 (left) – 143 (right) and the value of *expression2* is from 0 (top) – to 47 (bottom).

If the 3rd parameter is not used, the point will be set. If the 3rd parameter is present, `PSET` will invert the current state of the pixel. An on pixel will turn off and vice versa.

RADIAN

Format: RADIAN

Abbr: RAD.

See also: DEGREE, GRAD

Description: sets angular mode radians.

In this mode, all angular data is assumed to be in radians. To mark this, the symbol **RAD** appears in the status line. All arguments of the functions SIN, COS and TAN and the results from ASN, ACS and ATN are in radians. Radian form represents the angle in terms of the length of the arc with respect to the radius, i.e., 360° is 2π radians since the circumference of a circle is 2π times the radius.

Example:

```
10: RADIAN
20: PAUSE "ANGLES IN RADIANS"
30: PRINT ASN (0.5), ASN (1)
40: PRINT ACS (0.5), ACS (1)
50: PRINT ATN (0.5), ATN (1)
60: END
RUN ANGLE IN RADIANS
5.235987E-01 1.570796327
1.047197551 0
0.463647609 7.853981E-01
>
```

RANDOMIZE

Format: RANDOMIZE

Abbr: RA.

See also: RND

Description: resets the seed for random number generation.

When random numbers are generated using the RND function, the computer begins with a predetermined “seed” or starting number. RANDOMIZE resets this seed to a new randomly determined value.

The starting seed will be the same each time the computer is turned on, so the sequence of random numbers generated with RND is the same each time, unless the seed is changed. This is very convenient during the development of a program because it means the behavior of the program should be the same each time it is run, even with the RND function. When you want the numbers to be truly random, the RANDOMIZE statement can be used to make the seed itself random.

Example:

```
10: RANDOMIZE
20: X = RND(10)
```

READ

Format: `READ` *`variable`*`[,` *`variable`*`]…`

Abbr: `REA.`

See also: `DIM`, `RESTORE`

Description: reads values contained in `DATA` statements and assigns them to the listed variables. At least one `DATA` statement must be present within the program.

Each variable listed as is sequentially assigned to the next element in a `DATA` statement. The variables in the parameter list must be of the same type as constants in the `DATA` statement.

The values to be read do not all have to be in a single `DATA` statement, it can be distributed over any number of `DATA` statements. It always applies that the next read variable in a `DATA` statement is read with the next `READ` variable.

If all the values in the `DATA` lines are read, the next `READ` instruction results in an `ERROR` code.

Example:

```
10: DIM B (10)
20: FOR I = 1 TO 10: READ B (I)
30: PRINT B (I): NEXT I
40: DATA 10,20,30,40,50
50: DATA 60,70,80,90,100
```

REM (‘)

Format: `REM`

Description: allows insertion of comments in the program text.

Comments are used to identify parts of the program. The comments are ignored during program execution. Instead of `REM`, the apostrophe (') can also be used. Program lines marked as such are non-executable instructions. If these are jumped to by `GOTO` or `GOSUB`, program execution continues with the next non-comment line. Comments can be added after statements in a line by using the colon (:) as the delimiter before `REM`.

```
10: V = G * H / 3: REM VOLUME OF A PYRAMID
```

If using an apostrophe, a colon is not required, as it includes the separation function:

```
10: V = G * H / 3 'VOLUME OF A PYRAMID
```

After the `REM` instruction, the rest of the line is considered a comment. Any statements after the comment on the same line are ignored.

```
10: REM VOLUME OF PYRAMID: V = G * H / 3
20: 'VOLUME OF PYRAMID: V = G * H / 3
```

RENUM

Format: `RENUM [oldline[, newline][, increment]]`

Abbr: `REN.`

Description: renumbers the lines of a BASIC program.

Line numbers are changed from old line numbers to new line numbers with the specified increment.

oldline	: selects the first line for renumbering.
newline	: defines the new starting line number.
increment	: specifies the interval which the numbers are generated. If this information is missing, line numbers are generated in increments of ten.

`RENUM` automatically corrects all line numbers in `GOTO` and `GOSUB` statements accordingly. This also applies to the line numbers used in other branch instructions, e.g. `IF…THEN…ELSE`, are included. If renumbering is attempted with a non-existent line number, an error message is displayed and renumbering is stopped.

REPEAT … UNTIL

Format:

```
REPEAT
  statement
UNTIL condition
```

Abbr: `REP. UN.`

Description: execute statements between `REPEAT` and `UNTIL` until *condition* is true.

`REPEAT` and `UNTIL` includes a set of statements to be repeated. After executing the statements after `REPEAT`, *condition* is checked by `UNTIL`. If *condition* is true, execution continues with the line after `UNTIL`. This completes the loop. If *condition* is false, the statements after `REPEAT` are executed until *condition* is true.

A `REPEAT…UNTIL` loop may be nested within another one. The inner loop must be completely nested within the outer loop.

If the program exits the `REPEAT…UNTIL` loop before *condition* is true, a nesting error may occur, depending on how the loops are executed (for example, if the program contains several `REPEAT` statements).

`REPEAT` and `UNTIL` must always be used together. The commands `CLEAR`, `DIM`, and `ERASE` cannot be use in a `REPEAT…UNTIL` loop.

RESTORE

Format: `RESTORE [`*`line-number`*`|"`*`label`*`"]`

Abbr: `RES.`

See also: `DIM`, `READ`

Description: resets the `DATA` pointer or to the beginning of the indicated `DATA` line. This allows the values provided by the `DATA` statements to be read again.

If `RESTORE` is used without parameters, the pointer will be set to the first value of the first `DATA` line found in the program.

If a *line-number* or a "*label*" is specified, the pointer is set to the first element of the `DATA` statement occurring in this line.

If the line specified as a parameter does not contain a `DATA` statement, the pointer is set to the beginning of the next `DATA` statement.

Example:

```
100: DIM A$(3*10)
110: GOSUB "FRUIT"
120: RESTORE
130: GOSUB "FRUIT"
140: RESTORE 310
150: GOSUB "FRUIT"
160: END
200: "FRUIT"
210: FOR N = 1 TO 3
220: READ A$(I)
230: PAUSE A$
240: NEXT N
250: PAUSE
260: RETURN
300: DATA "PLUM", "PEACH" "," NECTARINE "
310: DATA" APPLE "," PEAR "," MANDRIN”
>RUN
PLUM
PEACH
NECTARINE
PLUM
PEACHCH
NECTARINE
APPLE
PEAR
MANDRIN
```

RIGHT$

Format: `RIGHT$(`*`string`*`, `*`number`*`)`

Abbr: `RI.`

See also: `LEN`, `LEFT$`, `MID$`

Description: returns a substring of *number* characters starting from the right of *string*.

number must be in the range 0 to 255. If a fractional number is used, it will be rounded to the nearest whole number. If *number* is greater than the number of characters of the given string, the entire string is returned.

Example:

```
5: WAIT 32
10: X $ = "SHARP"
20: FOR N = 1TO 6
30: S $ = RIGHT$(XS, N)
40: PRINT S$
50: NEXT N
60: WAIT
70: END
>RUN
P
RP
ARP
HARP
SHARP
SHARP
>
```

RND

Format: `RND(`*`expression`*`)`

Abbr: `RN.`

See also: `RANDOMIZE`

Description: generates a random number

The argument *expression* can be any numeric expression. If *expression* is less than 1 but greater than or equal to 0, the random number is less than 1 and greater than 0. If *expression* is an integer greater than or equal to 1, the result is a random number greater than or equal to 1 and less than or equal to *expression*. If *expression* is greater than or equal to 1 and not an integer, the result is a random number greater than or equal to 1 and less than or equal to the smallest integer that is larger than the expression. (In this case, the generation of the random number changes dependent on the decimal portion of the argument.) If *expression* is negative, the previously set numeric expression is used to generate the random number.

<number>	Lower Bound	Upper Bound
.5	0<	<1
2	1	2
2.5	1	3

The same sequence of random numbers is normally generated because the same “seed” is used each time the computer is turned on. To randomize the seed, use the `RANDOMIZE` command.

Example:

```
10: FOR I = 1 TO 3
20: FOR J = 1 TO 10: R = RND(9): PRINT R; : NEXT J
30: PRINT: NEXT I
40: END
>RUN
6425682768
5577126536
3157345742
```

RUN

Format: `RUN [`*`line-number`*`|"`*`label`*`"]`

Abbr: `R.`

See also: `END`, `STOP`

Description: starts a BASIC program in memory.

Without a parameter, the execution of the program begins with its first line, that is, the smallest occurring line number. Specifying a *line-number* or a *label* starts the program from the specified point.

`RUN` deletes variables and sets the internal pointer of `DATA` statement to the first possible position.

Example:

```
>RUN
>RUN 100
>RUN "F"
>RUN "BRAND"
```

SAVE

Format: `SAVE "`*`filename`*`[.BAS]"`

Abbr: `SA.`

Description: saves the BASIC program in memory to the RAM disk. If *filename* already exists, it will be overwritten. If no extension is specified, `.BAS` is assumed.

See also: `LOAD`, `KILL`

Example:

```
SAVE "TEST"
```

STOP

Format: `STOP`

Abbr: `S.`

See also: `CONT`

Description: interrupt a program during the test phase.

If a `STOP` command is issued, the program is aborted and a message `BREAK IN <line number>` is displayed stating which line was aborted.

By querying the variables and other examinations, the source of program errors can be discovered. Likewise, incorrectly assigned variables can now be assigned the expected values and further behavior can be tested by restarting the program with `CONT`. Continuation is only possible if no changes have been made to any program lines.

Unlike `END`, `STOP` does not close any open files.

Example:

```
10: FOR N = 1 TO 10
20: LET S = N * 5
30: STOP
40: GRAPH
50: LINE(0,0) - (N, S)
60: NEXT N
```

STR$

Format: `STR$(`*`expression`*`)`

Abbr: `STR.`

Description: converts *expression* into a string.

The resulting string consists of the characters of the numeric value. However, the string cannot be used for calculations.

The function `STR$` can be regarded as the inverse of the `VAL` function. If the numeric value is negative, the string also contains the relevant sign.

If the numerical value is too large to represent with ten digits, it will appear in floating point notation.

Example:

```
     ⋮
110: N = N * 3
120: A$ = STR$(N)
130: B$ = LEFT$(A$, 1)
140: M = VAL(B$)
     ⋮
```

SWITCH … CASE … DEFAULT … ENDSWITCH

Format:

```
SWITCH variable
  CASE value1 | letter-sequence1
    statement1
  CASE value2 | letter-sequence2
    statement2
          ⋮
  [DEFAULT
    statement#]
ENDSWITCH
```

Abbr: `SW. CAS. DEFA. ENDS.`

Description: executes specific instructions according to the value of a given variable.

This command compares the value of *variable* following `SWITCH` with a number or a string of letters that follows each `CASE` statement. If they match, the statements between the matching `CASE` statement and the next `CASE` statement are executed. If the value of the variable does not match any `CASE` statement, the `DEFAULT` statement is executed. If no `DEFAULT` statement is available, the `ENDSWITCH` statement is executed.

If the same sequence of numbers or strings is used in more than one `CASE` statement, the `CASE` statement closest to the `SWITCH` statement will be executed if they match.

`SWITCH` and `ENDSWITCH` must always be used together. `CASE`, `DEFAULT` and `ENDSWITCH` must always follow after a line number, not a label. `SWITCH` statements cannot be nested.

TRON / TROFF

Format: `TRON`
`TROFF`

Abbr: `TR.` `TROF.`

Description: switch **TRACE** mode on or off.

If this mode is activated via the `TRON` command, the computer stops after executing each BASIC line and displays the line number on the screen.

Each subsequent keypress will execute and display the line number of the next program line. If a key is pressed continuously, the computer processes the program lines one after the other without displaying the corresponding line numbers. A line which has just been processed can be made visible by pressing ▲.

If execution halts as a result of a `PRINT` or `INPUT` command, execution can be continued by pressing ↵.

If the program halts due to a `STOP` command or if it was aborted by the **BREAK** key, the program can be restarted with **SHIFT** + **CA**.

`TROFF` turns off the trace mode.

`TRON` and `TROFF` are also programmable. **TRACE** mode will remain in effect until the next `TROFF` command is found in the program.

USING

Format: `USING [`*`format-string`*`]`

Abbr: `U.`

Description: controls the format of displayed or printed output.

`USING` can be used by itself or as a clause in a `PRINT`, or `LPRINT` statement. When used in a `PRINT`, or `LPRINT` statement, it is valid only for the values or strings output by that statement. If used independently (on an independent line), it is valid for all subsequent `PRINT` or `LPRINT` commands. `USING` establishes a specified output format for all output that follows until changed by another `USING`.

Format is determined by *format-string*, which consists of a series of characters that must be enclosed in quotation marks. The characters that make up *format-string* are:

#	Right-justified numeric field character
.	Decimal point (delimiter between the integer and decimal part of a number
,	3-digit separator in numeric fields
^	Display the number in scientific notation
&	Left-justified alphanumeric field

The number sign (#) and the ampersand (&) are placeholders. For each # contained in *format-string*, one digit of the numerical value can be displayed. For each &, one character of a string can be displayed. All other format symbols are used to describe the numeric formats in more detail. With the numerical formats, both positive and negative values can be represented. However, the sign is only displayed for negative values.

Together with the string format can be a total of six basic formats can be defined:

(1) "###"	43
(2) "###."	98.
(3) "###. ##"	64.29
	5.00
	-13.44
	-2.00
(4) "##. ##^"	23.11E 05
	-4.33E-02
	7.00E 00
(5) "###, ###."	34,567. 2
	-230.
	2,345.
(6) "&&&&&&"	ABCDEF

The maximum number of # allowed in formats (1), (2), (3), (4) and (6) is 11 and 14 in format (5).

VAL

Format: `VAL(`*`string`*`)`

Abbr: `V.`

Description: converts *string* to a numeric value.

The `VAL` function can be regarded as the inverse of two functions, `STR$` and `HEX$`. It converts a string consisting of numeric characters into a numeric value.

If *string* is a decimal string, it must be composed of the characters 0 through 9. It may contain a decimal point and an exponent, plus a sign for the mantissa and one for the exponent. In this case, `VAL` is the exact inverse of `STR$`.

If *string* is a hexadecimal string, the first character in the string must be ‘&’, and the subsequent characters must be symbols used to represent hex digits. In this case, `VAL` acts as the inverse of the `HEX$` function.

If *string* contains invalid characters, 0 is returned.

Example:

```
10: INPUT "FREQUENCY ="; A $
15: IF ASC(A$) <48 OR ASC(A$)> 57 THEN 100
20: F = VAL(A$)
30: PRINT F
40: END
     ⋮
100: PRINT "ONLY NUMBER ENTRY ALLOWED"
```

VDEG

Format: `VDEG(`*`string`*`)`

Abbr: `VD.`

See also: `DEG`, `DMS`

Description: converts a string in the format "dd°mm’ssrr" (sexagesimal) to an angle in degrees, where:

dd	: degrees
mm	: minutes
ss	: seconds
rr	: fractional seconds (00 ... 99)

Example:

```
10: AA$ = "1°30'36""
20: B = VDEG AA$
30: PRINT B
>RUN
1.51
```

WAIT

Format: `WAIT[`*`expression`*`]`

Abbr: `W.`

Description: controls the length of time that displayed information is shown before program execution continues.

`WAIT` without a parameter sets the waiting time to 0. After a `PRINT` statement, the program stops completely. However, since you can see a >, the program is not aborted. It only paused and can be continued by pressing ↵.

If *expression* is specified, the waiting time is determined as a multiple of 1/64 second. A value 64 results in a pause of 1 second, a value 128 of 2 seconds, etc. The value of *expression* ranges from 0…65535. *expression* can be any numeric expression.

A default `WAIT` duration of 0 is set on starting a program.

Example:

```
10: FOR I = 1 TO 10
20: WAIT (64 * I)
30: PRINT "*";
40: NEXT I
50: WAIT
60: END
>RUN
**********
>
```

Each star appears 1 second later than the previous one.

WHILE … WEND

Format:

```
WHILE condition
  statement
WEND
```

Abbr: `WH. WE.`

Description: The instructions between `WHILE` and `WEND` are executed as long as *condition* is true.

First, *condition* of the `WHILE` statement is checked. If *condition* is false, execution resumes at the statement after `WEND`. If *condition* is true, the instructions between `WHILE` and `WEND` are repeated until *condition* is false.

`WHILE` and `WEND` must always be used together. The commands `CLEAR`, `DIM` or `ERASE` cannot be used within a `WHILE` loop.

13.3.I/O Commands

CLOSE

Format: `CLOSE [#`*`file-number1`*` [, #`*`file-number2`*`]…]`

Abbr: `CLOS.`

See also: `END, OPEN`

Description: closes all I/O ports.

Closing an I/O port does not require any additional parameters.

INP

Format: `INP(`*`port`*`)`
`INP`

See also: `OUT`

Description: returns a byte of data from the specified port.

The value *port* determines the input port from which a byte is to be fetched. The port is specified by a 16-bit address, i.e. with a value in the range 0…65535 or &0…&FFFF

`INP` without parameters returns the values of `XIN`, `DIN` and `ACK` from the 11-pin interface.

Example:

```
10: A = INP
20: PRINT A
>3                       0x4      +    1x2      +    1x1    = 3
                      XIN = Lo     DIN = HI     ACK = HI
```

LLIST

Format: `LLIST [`*`line-number1`*`|"`*`label1`*`"][-[`*`line-number2`*`|"`*`label2`*`"]]`

Abbr: `LL.`

Description: outputs lines of a BASIC program.

Use of `LLIST` is as described under **General Commands** (page 211)**.** However, if a I/O port is open, output is sent as ASCII characters to the I/O port. If the I/O port is closed, output is directed to the printer (CE-126P).

LPRINT

Format: `LPRINT [USING "`*`format`*`"] [`*`expression1`*`|`*`string1`*`[,|; [`*`expression2`*`|`*`string2`*`]…][;]`

Abbr: `LP.`

See also: `PRINT`

Description: sends output to the printer, I/O port.

Use of `LPRINT` is as described under **General Commands** (page 214). However, if the I/O port is open, output is sent as ASCII characters to the I/O port. If the I/O port is closed, output is directed to the printer (CE-126P).

OPEN

Format: `OPEN "COM:|COM1:|LPRT:|PIO:"`

Abbr: `OP.`

See also: `CLOSE, END`

Description: opens a port on a device for reading or writing.

Opening an I/O device does not require any additional options.

`COM:` Serial input/output. Input/output takes place via the usual input and output commands. The file number is #1 (for example, `PRINT#1,"Hello world."`) It is also possible to communicate using the `INP` and `OUT` commands. The settings for the serial interface are found in **TEXT** mode under `SIO Format`.

`COM1:` Similar to `COM:`

`PIO:` communication with the `PIO`. Input / output is via the commands `PIOSET`, `PIOGET`, `PIOPUT`, `INP`, `OUT`

`LPRT:` output to a serial printer. Commands like `LPRINT`, `LLIST` are redirected to the serial port

Example:

```
10: OPEN "LPRT:"

10: OPEN "COM1:"

10: OPEN "COM:"
```

OUT

Format: `OUT` *`address`*`,` *`byte`*`[`*`byte`*`]…`
`OUT` *`value`*

Description: sends one byte to the desired memory address of the Z80-compatible microprocessor.

address	: is an address (16-bit value) in the range 0…65535 (&0…&FFFF), which selects the desired port.
byte	: specifies the value to be inserted to the address. If several bytes are listed, each subsequent byte is sent to the next memory address. Successive bytes are thus passed to consecutive addresses in memory.

`OUT` also sets the bits for `BUSY`, `DOUT` and `XOUT` on the 11-pin interface. Before setting, `OPEN` must be executed on `COM:`, `COM1:` or `LPRT`.

See also: `INP`

Example:

`OUT 80,187`	This statement sends the value 187 (= &BB) to memory address 80 (= &50).
`OUT 6`	6 = 1x4 + 1x2 + 0x1 `BUSY` = HI, `DOUT` = HI, `XOUT` = LO

PIOGET

Format: `PIOGET`

Description: reads a byte from the PIO port.

The PIO port must be initialized with `OPEN "PIO:"` and `PIOSET` prior to running `PIOGET`.

PIOPUT

Format: `PIOPUT` *`byte`*

Description: writes byte to the PIO port.

The PIO port must be initialized with `OPEN "PIO:"` and `PIOSET` prior to running `PIOPUT`.

PIOSET

Format: `PIOSET` *`byte`*

Description: sets input and output mode of the PIO port.

byte is interpreted bit by bit:

Bit 7	: EX2
Bit 6	: EX1
Bit 5	: ACK
Bit 4	: Din
Bit 3	: Xout
Bit 2	: Xin
Bit 1	: Dout
Bit 0	: Busy

Example: `PIOSET &HF0` (Din, ACK, EX1, EX2)

APPENDIX A: 11-PIN INTERFACE

Signals and Pin-Out

On the left side of the PC-G850V(S) there is an 11-pin interface intended for communication with other devices. It is a multi-functional interface, hence, it can operate in different (sub-)modes. The actual mode is selected through operational commands or menu items of the PC-G850V(S).

1. SIO / RS-232C mode (e.g. `OPEN"COM:"`)
2. SSIO mode (Synchronous Serial Input/Output)
    a. CE-126P print protocol (e.g. `LPRINT` without preceding `OPEN`)
    b. LPRT-protocol (e.g. `OPEN"LPRT:"`)
3. PWM mode (Pulse Width Modulation)
    a. CE-126P tape protocol (e.g. `BSAVE/BLOAD` with a CE-126P)
    b. Generic PWM-protocol (e.g. `BSAVE/BLOAD` with another PC-G850V)
4. PIO mode (e.g. `OPEN"PIO:"`)
   Programmable, 8-bit parallel port interface
5. PIC mode (activated by the PIC-loader in the assembler menu)
   Programming interface for PIC microcontrollers

The association of physical pins to logical signals (called pin-out) as well as the configured direction for input (I) or output (O) depends on the active mode. The following table gives an overview. Looking at the left side of the PC-G850V(S) pin-1 is the leftmost and pin-11 the rightmost.

Pin	SIO mode		SSIO/PWM mode		PIO mode		PIC mode	
#	Signal	I/O	Signal	I/O	Signal	I/O	Signal	I/O
1	–	–	–	–	–	–	–	–
2	VCC (+5V)	–	VCC (+5V)	–	VCC (+5V)	–	VCC (+5V)	–
3	GND	–	GND	–	GND	–	GND	–
4	RTS	O	BUSY	O	Bit0	I/O	CP	O
5	DTR	O	DOUT	O	Bit1	I/O	CLK#	O
6	RXD	I	XIN	I	Bit2	I/O	DATAIN	I
7	TXD	O	XOUT	O	Bit3	I/O	DATAOUT	O
8	CD	I	DIN	I	Bit4	I/O	LOWBATT#	I
9	CTS	I	ACK	I	Bit5	I/O	–	–
10	DSR	I	EX1	I	Bit6	I/O	–	–
11	CI	I	EX2	I	Bit7	I/O	–	–

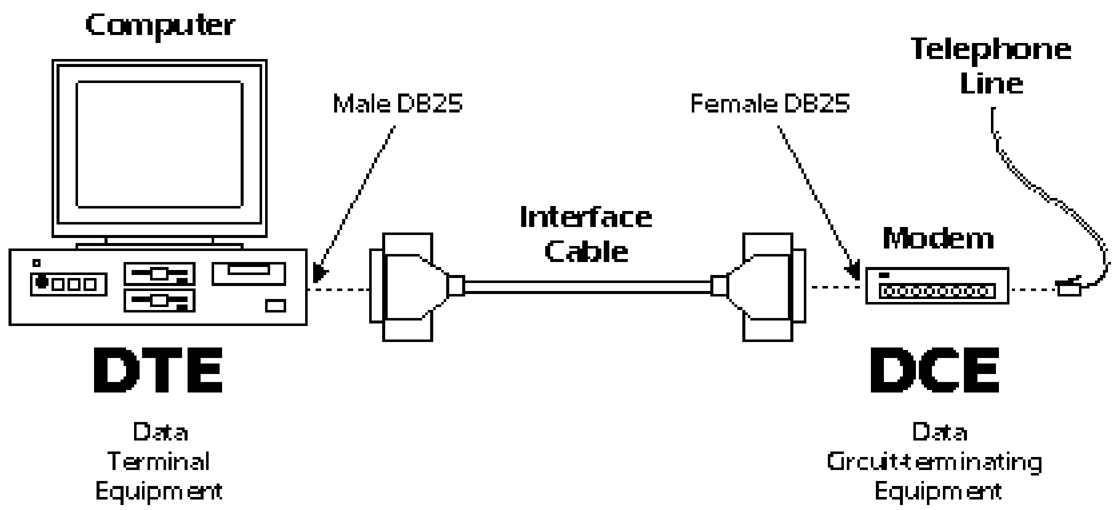
The next sections describe the SIO mode and the respective connection options in detail. The other modes are covered subsequently.

SIO mode

RS-232 Standard and Conventions

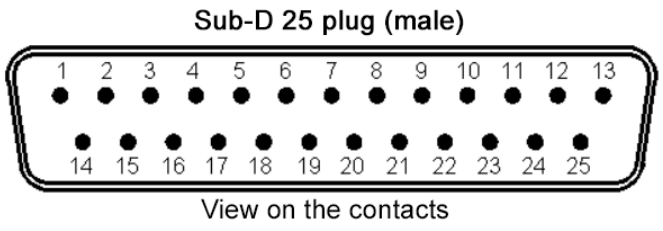
The PC-G850V(S) in SIO mode exposes the signals of the RS-232 standard, but with different voltage levels (see below). This section provides the necessary basics of the standard and covers some specifics of the PC-G850V(S).

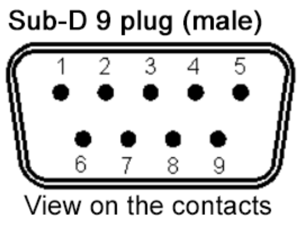
Within the RS-232 standard the terms DTE (Data Terminal Equipment) and DCE (Data Communication Equipment) are introduced. The DTE is the PC-G850 for example and the DCE is a modem or another peripheral device, like a serial printer.

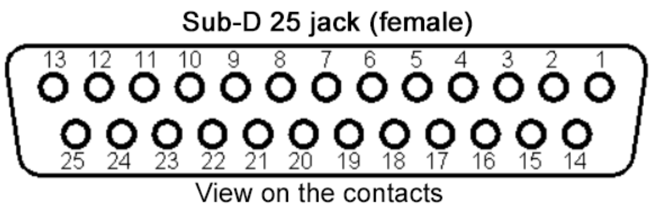


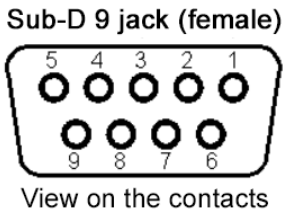
When two computers shall communicate directly (i.e. without a modem), you need a so called null-modem (cable/adaptor), which connects the outputs of one DTE with the inputs of the other and vice versa (crossed signals).

Typically 25-pin (Sub-D 25 / DB-25) or 9-pin (Sub-D 9 / DB-9) plugs and jacks are used to connect RS-232 capable devices.









The pin-out and meanings are summarized in the following table.

Signal	Name	Alternative Name	Direction (DTE view)	Definition	Pin # DB9	Pin# DB25
TXD	Transmitted Data	SD	Out	Data from DTE to DCE	3	2
RXD	Received Data	RD	In	Data from DCE to DTE	2	3
RTS RTR	Request (Ready) to Send Ready to Receive	RS	Out	DTE requests permission from DCE to send data DTE is ready to receive data from DCE	7	4
CTS	Clear to Send	CS	In	DCE is ready to receive data from DTE	8	5
DTR	Data Terminal Ready	ER	Out	DTE interface ready for operation	4	20
DSR	Data Set Ready	DR	In	DCE interface ready for operation	6	6
CD	Carrier Detect		In	DCE detects remote DCE	1	8
CI	Call Indicator	DI	In	Call of a remote DCE	9	22
GND	Signal Ground	SG	None	Signal ground (reference)	5	7
FG	Frame Ground	PG	None	Shield	–	1

Note: In the late 1980's there was a shift in the meaning of the RTS signal: Originally the DTE (computer) requests the DCE (modem) for permission that the DTE may send data - and the DCE "answers" via CTS. But this protocol is asymmetric because the DTE has no means to notify the DCE to wait for internal computations when the DCE sends data. For this reason, "Request To Send" was re-claimed: The DTE requests the DCE to send data – or in other words, the DTE is "Ready To Receive" (RTR). RTR and CTS are now independent of each other and the protocol between DTE and DCE is symmetric. But in most cases the name "Request To Send" (RTS) was kept, hence it is fairly ambiguous.

The PC-G850V(S) implements the newer, symmetric RTR-semantics (but the signal name RTS has been kept). This is in contrast to the preceding pocket computer model PC-E500(S), which implements the original RTS meaning and therefore needs the XON/XOFF-protocol when it reads data/programs from a PC. The PCG850V(S) sets the DTR signal to HIGH, when the SIO-interface is active, but it does not care about the DSR input. So there is no DTR/DSR-handshake. The RTS/CTS handshake, or alternatively the XON/XOFF-protocol can be configured in the TEXT/Sio/Format-submenu by the item "flow".

Signal Levels

The following table summarizes the logic and voltage levels of the RS-232 standard in comparison with UART-TTL and the PC-G850V(S).

	Logic level	Voltage level	Description of data signals (RXD, TXD)	Description of control signals (RTS, CTS, etc.)
RS-232	LOW	-15V to -3V	1 (Mark), Idle, Stop	Inactive
	HIGH	+3v to +15V	0 (Space), Start	Active
UART-TTL	LOW	0V	0 (Space), Start	Active
	HIGH	+3.3V/+5V	1 (Mark), Idle, Stop	Inactive
PC-G850V(S)	LOW	0V	1 (Mark), Idle, Stop	Inactive
	HIGH	5V	0 (Space), Start	Active

So the PC-G850V(S) exposes inverted UART-TTL level signals in SIO-mode, just as most other SHARP pocket computers do. That means the logic is identical to the RS-232 standard (HIGH=0/active), but the voltage level is TTL.

Caution: In order to connect peripheral devices with the PC-G850V(S) that operate at RS-232 voltage levels, a level converter is mandatory!

The state of the TXD- and RTS-signals in SIO mode is undefined, except for the following cases:

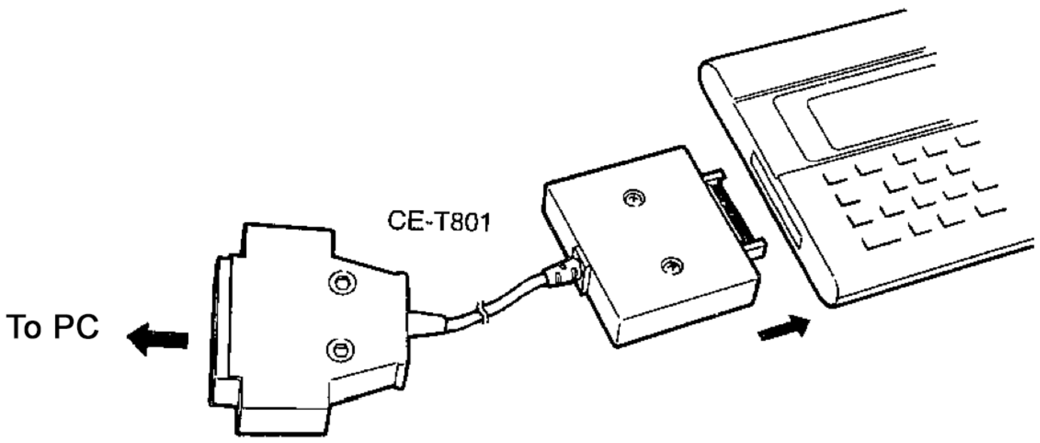
1. The interface has explicitly been opened in SIO mode (e.g. `OPEN"COM:"`) when in **BASIC** mode.
2. R- or W-commands are executed in **MON** mode.
3. Data transfer via SIO in **TEXT** mode.

Data Transfer Cable CE-T800 and CE-T801

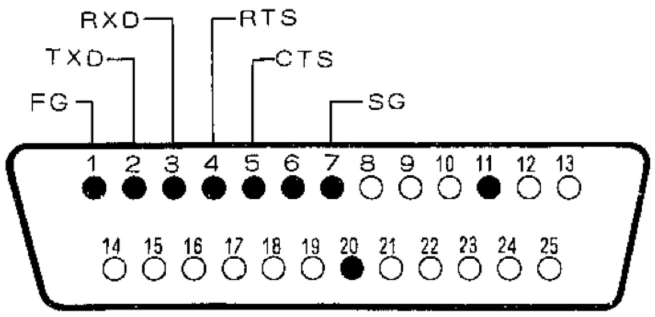
The data transfer cables CE-T800 and CE-T801 are RS-232 level converters with an integrated null-modem wiring. They can be used to connect the PC-G850V(S) to a personal computer (PC) or other devices.

With these cables you can transfer data, program source-code or machine language programs from or to a PC by using the TEXT/SIO submenu or the SIO commands (R, W) of the integrated hex-monitor (**MON**). The DB-25 plug of the cable can be connected directly to a PC (if necessary via a DB-9 adaptor), when there is a physical COM port. Alternatively, it can be connected to a USB-port through an additional serial-to-USB adaptor. Don't use a null-modem adaptor or wiring for a PC-connection (because it's already integrated in the cable).

However, if a peripheral RS-232 device like the 4-color plotter CE-515P is to be connected, a null-modem adaptor/wiring is mandatory in order to compensate for the integrated one.



Pin-out of the CE-T801 plug (DB-25)



Pin-6 and pin-20 are interconnected

On the CE-T800 pins 6 and 20 are not connected, pin 11 is not connected on both models (CE-T800 and CE-T801).

Caution: Never touch the pins of the DB-25 plug. Static electricity may be harmful for the circuits.

A free working area of about 300bytes is required for data transfer from a PC.

USB PC Adapter Cable with Hardware Handshake

An elegant, powerful yet simple DIY alternative to connecting the PC-G850V (S) to a modern PC is based on a pre-fabricated USB-UART adapter cable with an open ends.

Specification: FTDI USB-UART / TTL Adapter Cable with FT232R Chip, 5V, 6-pin (GND, 5V, RXD, TXD, RTS, CTS)

For physical connection with the 11-pin interface of the PC-G850V a common multi-pin connector with 2.54mm spacing can be used. You need to solder the UART lines of adaptor to the multi-pin connector by using a null-modem wiring.

FTDI-UART signal (color)	PC-G850V(S) signal (pin)
GND (black)	GND (3)
RXD (yellow)	TXD (7)
TXD (orange)	RXD (6)
CTS (brown)	RTS (4)
RTS (green)	CTS (9)
VCC (red)	-

Additionally, a 10Kohm resistor should be between pin 4 and pin 3 on the connector. This serves as a pulldown for the RTS signal in order to produce a defined LOW level. Without this, I/O errors in the data transfer from the PC to the PC-G850V(S) may occur because the PC may not receive wait requests from the Pocket Computer.

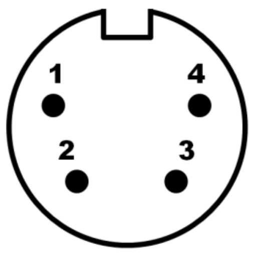
Finally, download the tool FT_PROG from the manufacturer's website: www.ftdichip.com. With this tool, you must invert the signals RXD, TXD, RTS and CTS of the FTDI chip since the serial interface of the PC-G850V(S) works with inverted UART logic (see above). This only needs to be done once as the settings are stored permanently in the integrated EEPROM of the FTDI chip.

RS-232 printer

The SIO mode can also be used to control printers that have an RS-232C interface, such as the 4-color plotter CE-515P or CE-516P.

Never connect an RS-232 printer without a level converter to the PC-G850V(S)! You can use the data transmission cable CE-T800 / 801 in combination with null modem wiring or adaptor. For the CE-515P / 516P via RS-232, a DIN-4 plug is required with the following wiring (null modem included):

DIN-4 plug		CE-T800 / 1 DB-25 plug	
Pin#	Signal	Pin#	Signal
1	+12V	–	
2	BUSY#	4	RTS
3	GND	7	GND
4	DATA#	3	RXD



DIN-4 plug, view on the pins

Be sure to properly configure the DIP switches on the back panel of the CE-515P / 516P (see printer manual for details). Additionally, the RS-232 settings in the TEXT/Sio/Format submenu of the PC-G850V(S) must be changed to the following to communicate with the CE-151P:

Baud rate	= 1200
Data bit	= 8
Stop bit	= 1
Parity	= none
End of line	= CR
Flow	= RS/CS

To direct output to a RS-232 printer, the 11-pin interface must be opened explicitly in SIO mode (`OPEN”COM:”`) and closed after use (`CLOSE`). Character strings and control codes are transmitted via the `PRINT#1` command.

```
OPEN”COM:”
PRINT#1,”HELLO WORLD”
  ⋮
CLOSE
```

The commands `LPRINT`, `LLIST`, `LFILES`, however, are not routed to the 11-pin interface in SIO mode.

SSIO mode

The SSIO mode is for synchronous, serial data transfer, in contrast to the asynchronous serial data transfer of the SIO mode. “Synchronous” means that the sender also provides and additional strobe/clock signal to which the receiver aligns. This makes an explicit baud rate obsolete. Therefore, SIO parameters in the TEXT / Sio / Format menu are irrelevant.

The SSIO mode of the PC-G850V (S) has several sub-modes or protocols.

CE-126P printer protocol

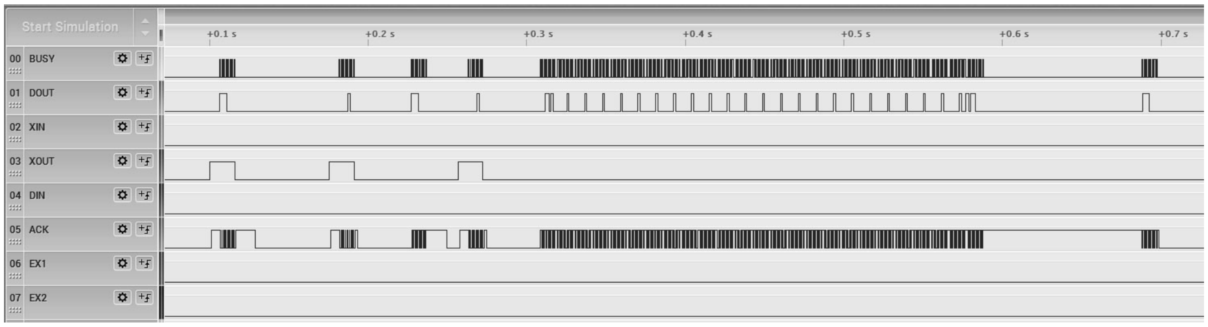
This is the default protocol for the 11-pin interface of the PC-G850V(S). It is the protocol for the CD-126P printer and it is active if and only if there are no other sub-modes selected. The commands `LPRINT`, `LLIST` and `LFILES` are routed to the printer in this mode.

The integrated cassette interface of the CE-126P can also be controlled by the PC-G850V. The corresponding protocol shares the same handshake as the printer protocol, but it uses PWM for data transfer instead of SSIO (see below).

The pin-out and signal descriptions within the CE-126P print protocol is as follows:

Pin#	Signal	Direction	Description
4	BUSY	Out	Clock for synchronous, serial data transmission
5	DOUT	Out	Data line
6	XIN	In	No function
7	XOUT	Out	HIGH: CE-126P sub-device select (printer vs. cassette interface) resp. command transfer LOW: idle or data transfer
8	DIN	In	No function
9	ACK	In	CE-126P ready to receive data or commands (handshake)
10	EX1	In	No function
11	EX2	In	No function

The following diagram shows the timing of the `LPRINT "X"` command with the CE-126P connected:



The PC-G850V (S) waits for the ACK signal before setting BUSY high. This synchronous serial protocol is also used by the CE-126P interface of the PC-E500(S).

LPRT Protocol and Mini I/O Port

The mini-I / O port of the PC-G850V(S) is just the logical grouping of the six main signals of the SSIO mode into two groups of three signals/bits each:

Mini I/O output port (3-bit)
- XOUT (Bit-0)
- DOUT (Bit-1)
- BUSY (Bit-2)

Mini I/O input port (3-bit)
- ACK (Bit-0)
- DIN (Bit-1)
- XIN (bit 2)

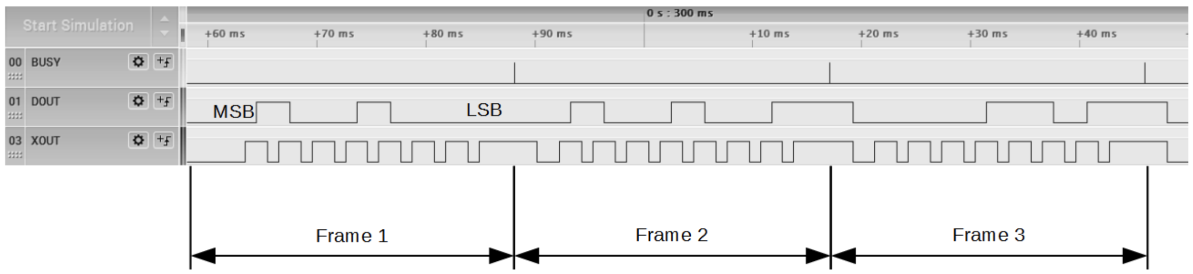
The bits of the mini I/O port can be explicitly controlled via the functions `OUT/miniput()` and `INP/miniget()`, so that custom communication protocols can be implemented on that basis

In addition, the PC-G850V(S) offers a synchronous, serial protocol for data transfer to a respective peripheral device. To enable this, the 11-pin interface must be opened with the command `OPEN("LPRT:")`. The data streams of the commands `LPRINT`, `LLIST` and `LFILES` are then sent over this protocol using ASCII code.

The definitions of the signals of the LPRT protocol is as follows:

Pin#	Signal	Direction	Function
4	BUSY	Out	Frame indicator for each transmitted byte
5	DOUT	Out	Data line
7	XOUT	Out	Clock with pause after each byte
9	ACK	In	LOW: Receiver is ready HIGH: PC- G850V(S) must wait

The following diagram shows the signal timings:



Data is transmitted byte byte-wise with the most significant bit (MSB) first. DOUT is valid on the rising edge of the pulse. The BUSY signal provides an additional reference frame for each byte.

PWM Mode

CE-126P Tape Protocol

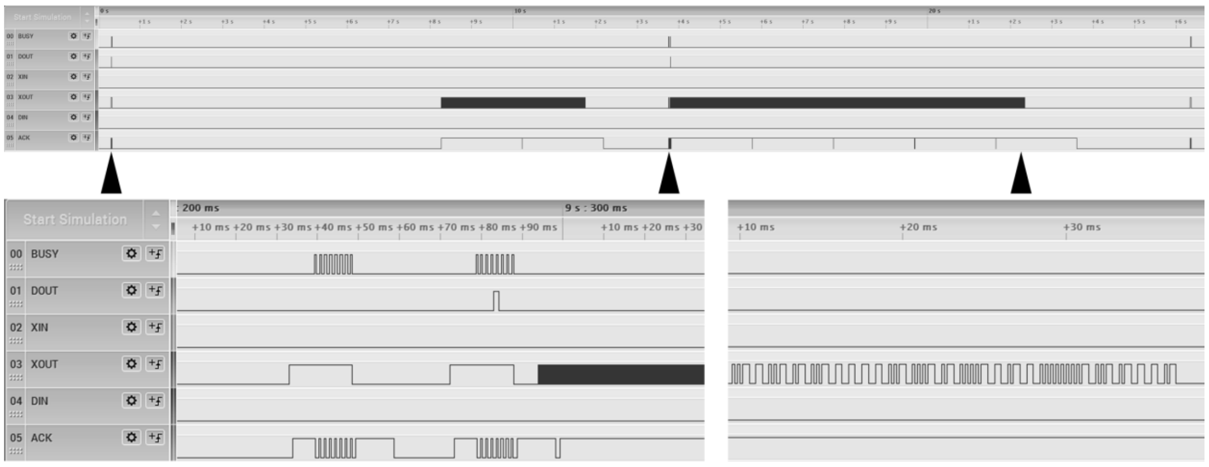
This protocol is implemented by the commands `BSAVE`, `BSAVEM`, `BLOAD`, `BLOADM`, `BLOAD?` when the CE-126P (or compatible cassette interface) is connected for storing, loading, and verifying BASIC programs or binary data (such as machine programs) by means of a cassette recorder, like the CE-152.

The protocol includes the SSIO handshake of the CE-126P printer protocol is identical, but data transfer is achieved using pulse width modulation (the digital equivalent of analog waveforms). It is a mixture of SSIO and PWM protocol.

Here are the definitions of the CE-126P tape protocol signals:

Pin#	Signal	Direction	Description
4	BUSY	Out	Clock for synchronous, serial handshake
5	DOUT	Out	Data line for handshake
6	XIN	In	PWM encoded data from the cassette interface (load)
7	XOUT	Out	Handshake: See CE-126P printer protocol Data: PWM encoded data to the cassette interface (save)
8	DIN	In	No function
9	ACK	In	CE-126P ready to receive data or commands (handshake)

The following diagram shows the signal paths of the CE-126P band protocol of a BSAVE execution (saving a single-line BASIC program):



CE-126P Handshake PWM Data Transfer

The dynamics for a `BLOAD` operation is equivalent. The difference that the PWM data is received on the XIN signal.

Generic PWM protocol

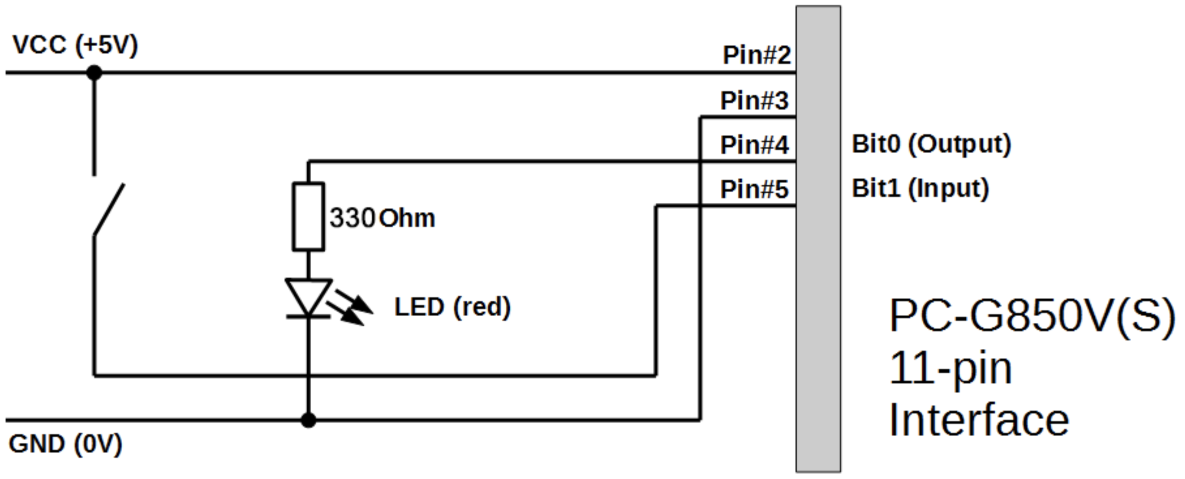
This protocol is equal to the CE-126P tape protocol reduced to XOUT and XIN (i.e., handshake is omitted). It is activated by the commands `BSAVE`, `BSAVEM`, `BLOAD`, `BLOADM`, `BLOAD?` if no CE-126P (or compatible cassette interface) is connected. Typically, this would be the case when two PC-G850V(S) are directly connected by data exchange cable, like the EA-129C. The PC-G850V(S) distinguishes between the CE-126P tape protocol and the generic PWM protocol by setting XOUT to HIGH at the start of a `BSAVE/BLOAD` command and then observes the response of ACK. If ACK is not set HIGH, then the generic PWM protocol is used (i.e. BUSY/DOUT/ACK handshake is skipped).

PIO mode

The PIO mode is primarily intended for controlling external digital hardware rather than communication with other devices. In this mode, the pocket computer becomes a microcontroller with an on-board development environment.

The 11-pin interface becomes a programmable 8-bit port. The logic levels (LOW/HIGH) can be set and read by the PIO API (application programming interface) in BASIC or C. Each of the 8 signals/bits can be configured individually to serve as input or output. The direction can be set by the `pioset/PIOSET` function (see Command Reference). The function `pioput/PIOPUT` sets the individual logic levels of each signal by setting the respective bit of 0 (LOW) or 1 (HIGH). Signals that are configured as input are ignored. The function `pioget/PIOGET` reads all 8 logic levels of the port into one byte.

The following is a very basic example of PIO mode use:



In this example, bit-0/pin-4 serves as an output which lights up a LED when it is HIGH. Bit 1/pin 5, on the other hand, Bit-1 / Pin-5, on the other hand, serves as an input and represents the status of a push button switch. An open input (i.e., undefined input level) is interpreted as logical 0, which is the case when the button is open. In order to distinguish that state from the closed state, the button is connected to VCC (i.e., HIGH / logic 1) and not to GND.

The goal of the "microcontroller" code would be to switch on the LED with the first button press and to switch it off with the next, and so on. An example in the C programming language is shown below (BASIC would be similar, however, less structured).

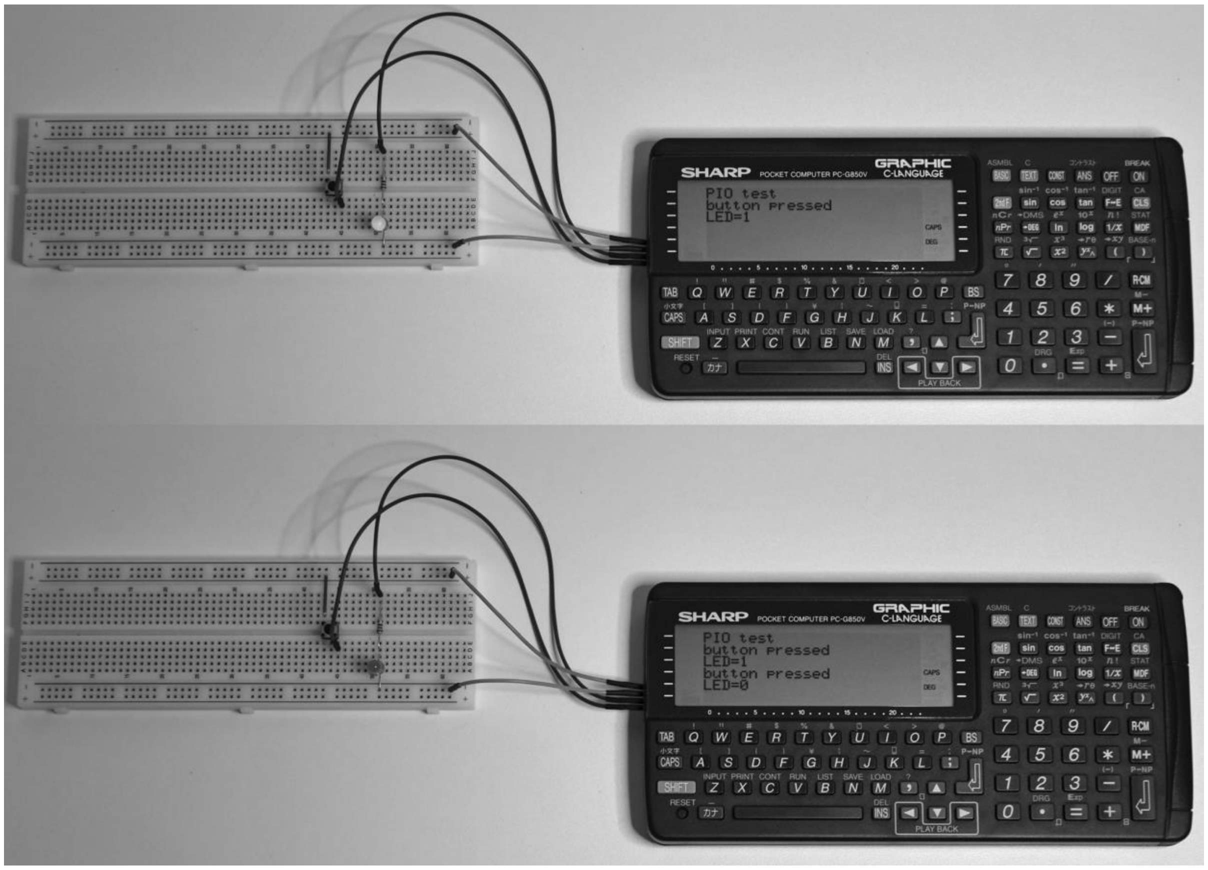
```
1 #define BOOL char
2 #define TRUE 1
3 #define FALSE 0
4 #define BTN 0x02
5
6 char BTNstate = 0;
7 char LEDstate = 0;
9
10 BOOL setupPIO() {
11  if(!fopen("pio","a+")) {
12   printf("can't open port\n");
13   return FALSE;
14  }
15  pioset(BTN);
16  return TRUE;
17 }
19
20 BOOL pressed() {
21  BOOL rtn=FALSE;
22  char btn;
23  btn=pioget()&BTN;
24  if(btn && BTNstate==0)
25   rtn=TRUE;
26  BTNstate=btn;
27  return rtn;
28 }
29
30 toggleLED() {
31  LEDstate=!LEDstate;
32  printf("LED=%x\n",LEDstate);
33  pioput(LEDstate);
34 }
39
100 main() {
101  printf("PIO test\n");
102  if(!setupPIO())
103   abort();
104  while(TRUE) {
105   if(pressed()){
106    printf("button pressed\n");
107    toggleLED();
108   }
109  }
110 }
```

To enter the symbol '\', press .SHIFT. G in TEXT mode. It is displayed as ¥.

Comments on the code:

- Line 4: Mask for bit-1 (0b00000010), i.e. push button input
- Line 6: Global state variable for the push button
- Line 7: Global state variable for the LED
- Line 11: Opens the interface in PIO mode for read and write.
- Line 15: Configure bit 1/pin 5 as input. All other signals are output.
- Line 20: This function detects the transition from bit 1 = 0 to bit 1 = 1, i.e. the close event of the push button.
- Line 23: The PIO port is read and all bits except bit 1 are masked (hidden).
- Line 30: This function changes the state of the LED
- Line 33: The new LED state (bit 0) is written to the port. Unused outputs are set to 0.
- Line 104: Main loop, abort with the ON / BREAK button

The following images shows an example of the test setup and the respective trace outputs on the display of the PC-G850V.



```
PIO test
button pressed
LED=1
```

```
PIO test
button pressed
LED=1
button pressed
LED=0
```

PIC mode

The PIC mode of the PC-G850V(S) is used to transfer an assembled PIC program (see chapter 12) to a PIC microcontroller. This process is called PIC programming, PIC program (up)loading, or PIC burning, since a specific “burning” voltage is needed that is much higher than normal operating voltage. This mode is activated by the PIC loader option in the PIC assembler submenu. The PC-G850V(S) supports the serial ICSP (In-Circuit Serial Programming) protocol of the PIC16F8x family and compatible models.

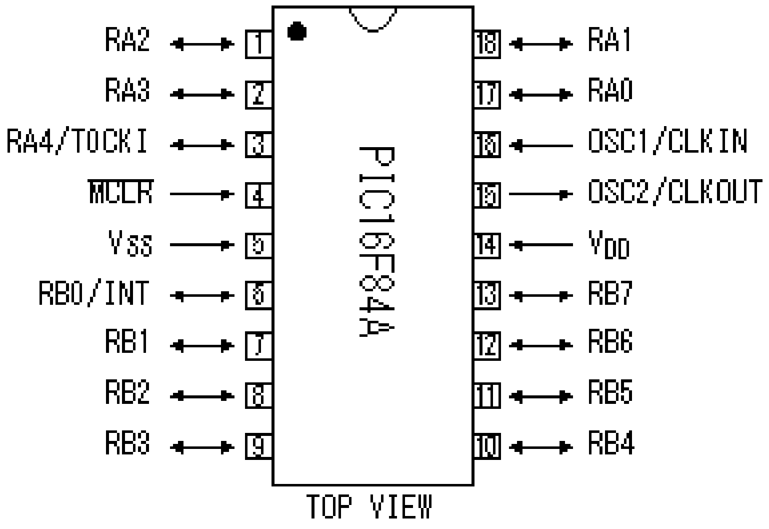
The pin out and signal description of the 11-pin interface in PIC mode are:

Pin#	Signal	Direction	Description
4	CP	Out	This signal controls the ICSP programming mode of the PIC. If the signal is HIGH, the burning voltage (+12 to +14V) must be applied to the $\overline{\text{MCLR}}$ of the PIC. If the signal is LOW, the $\overline{\text{MCLR}}$# must be at GND or VDD (+5V).
5	$\overline{\text{CLK}}$	Out	This signal provides the ICSP clock pulse for the PIC to be programmed. However, the inverted signal (i.e., CLK) must be provided at RB6 of the PIC. The latter latches data bits on the falling edge of the CLK pulse.
6	DATAIN	In	This input must be connected to RB7 of the PIC. Data is read by the PIC for programming and verification.
7	DATAOUT	Out	This output is used for serial data and command transfer to the PIC in ICSP mode. It needs to be connected to RB7 of the PIC.
8	$\overline{\text{LOWBATT}}$	In	This input can be connected to a voltage monitoring circuit (especially when using an external power supply). LOW is interpreted as power supply is too weak.

According to the specification of the PIC16F8x family, the PIC switches to ICSP programming mode as soon as the following conditions are fulfilled:

- VDD = + 5V, VSS = GND
- MCLR# = +12 to +14V
- RB6 (CLK) = LOW.
- RB7 (DATA) = LOW.

The following figure shows the pin assignment of the PIC16F84A:

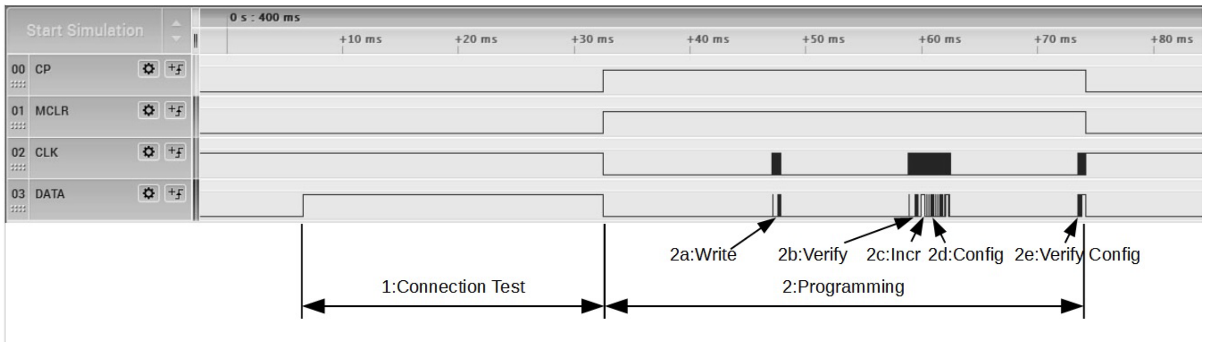


The PIC loader of the PC-G850V(S) supports the ICSP protocol as stated above, but this is preceded by a connection check. If it fails, the programming process is stopped and the error message `Connection error!` is displayed.

To explore the details of the PIC loader, we will use a minimalistic example code for the PIC16F84A. It consists of the configuration word and an infinite loop without a body:

```
10 #include “p16f84a.inc”
20 __config 0x3ff6
30loop goto loop
```

The PIC assembler compiles this source program into a PIC machine program which is only one PIC word (14-bit). The next diagram shows the phases of a successful burning process controlled by the integrated PIC loader of the PC-G850V(S):

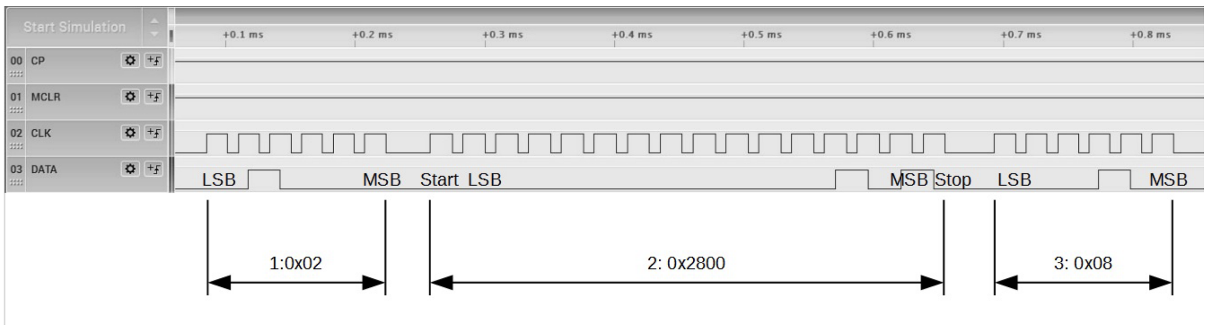


CLK was strobed at the RB6 input of the PIC (i.e., the already inverted $\overline{\text{CLK}}$ signal). Data was strobed at RB7. CP directly controls the programming voltage at $\overline{\text{MCLR}}$.

1. Connection Tests: The PC-G850 sets DATAOUT to HIGH and checks if DATAIN also goes high. The two signals must therefore be connected, otherwise the ICSP protocol will not be activated!

2. This is the ICSP phase. It is initiated by $\overline{\text{MCLR}}$ = 12.5V, CLK = LOW, DATA = LOW.
    a. Writing the assembled 14-bit word (PIC op-code)
    b. Verification of the last programmed PIC op-code
    c. Increment the PIC program counter. If the PIC program consisted of more than one word, there would be a 2a/b/c loop for every other word.
    d. Writing the PIC configuration word
    e. Verification of the PIC configuration word

An enlargement of the phase 2a shows the following:



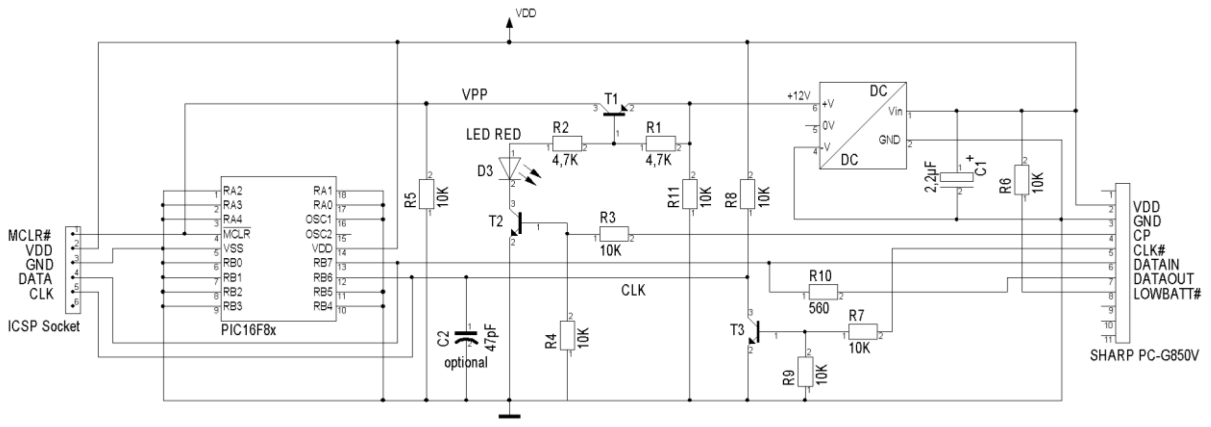
PIC-ICSP commands are 6 bits wide (see specifications for your PIC). A command can be followed by a data word, reading or writing. Data words are 14-bit wide, but they are framed by a start and a stop bit, so they are 16-bits overall. Transmission occurs with the least significant bit (LSB) first. As previously mentioned, all bits are latched/provided at the falling edge of the CLK pulse.

1. PIC ICSP command “Load Data for Program Memory” (0x02).
2. Data transfer for the “Load Data for Program Memory” command. For this example, the 14-bit op-code compiled by the PIC assembler is 0x2800.
3. PIC ICSP command “Start Programming Cycle” (0x08). This command has no data parameter and starts the PIC burning process for the latched 14-bit word.

A suitable PIC burner circuit is needed for the 11-pin interface of the PC-G850V(S), which uses the integrated PIC loader and supports the PIC16F8x microcontroller family. The circuit must meet the following criteria:

1. The CP signal must control the programming/burn voltage for the PIC.
2. DATAIN and DATAOUT must be interconnected for the connection check.
3. The $\overline{\text{CLK}}$ signal must be inverted at the RB6 pin of the PIC.
4. The $\overline{\text{CLK}}$ signal is very sensitive to crosstalk, especially from DATOUT. Shielding may be necessary. Additionally, a pull-down resistor is required for a defined LOW level of the $\overline{\text{CLK}}$ signal.
5. The $\overline{\text{LOWBATT}}$ input should either be connected to a programming voltage monitoring circuit or be constantly HIGH.

The following circuit meets these requirements. An additional feature is that it does not require an external power supply for the programming voltage, but generates it from the supply voltage (+5V) by means of a DC/DC converter:



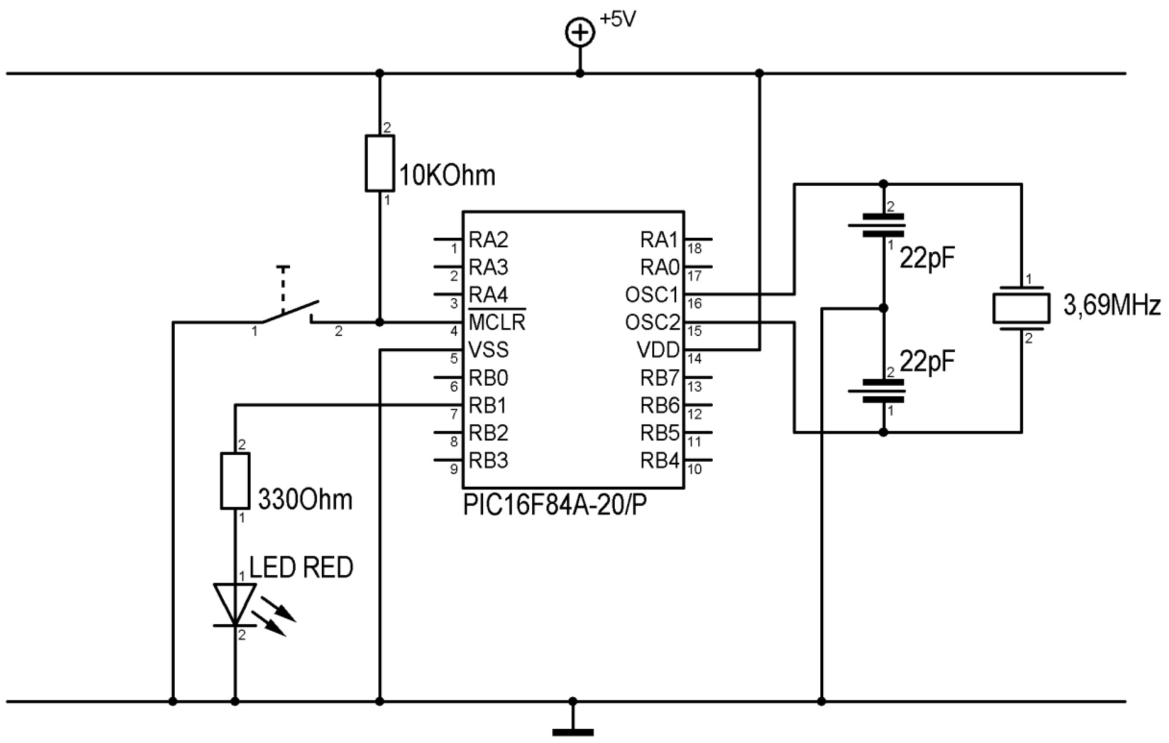
A +5V to +12V DC/DC converter (e.g., TMA0512C or ~D) can be used to generate the burn voltage. The CP signal controls this as VPP at the $\overline{\text{MCLR}}$ pin over the transistor path T1, T2. The LED serves as an indicator for ICSP mode. The $\overline{\text{CLK}}$ signal is inverted via T3 and R8 and is applied to the PIC as CLK. C2 is optional and serves as a low-pass filter to clear the CLK signal if necessary. The low voltage indicator applies only if the supply voltage falls below the LOW threshold (i.e., logical 0) during the programming phase.

To test the complete PIC programming process with the PC-G850V(S), a simple program that will flash an LED connected to pin RB1 of the PIC can be used:

```
10 #include "p16f84a.inc"
20 __config 0x3ff1 ;CP_OFF & PWRT_ON & WDT_OFF & XT_OSC
30DELAY1 equ 0x08 ;delay counter 1
40DELAY2 equ 0x09 ;delay counter 2
50 org 0
99
100start
110 bsf STATUS,RP0 ;change to bank 1
120 bcf TRISB,1 ;enable RB1 for output
130 bcf STATUS,RP0 ;back to bank 0
140loop
150 bsf PORTB,1 ;RB1=1,LED=on
160 call delay
170 bcf PORTB,1 ;RB1=0,LED=off
180 call delay
190 goto loop
299
300delay
310 movlw 255
320 movwf DELAY1
330 movwf DELAY2
340dloop
350 decfsz DELAY1,f
360 goto dloop
370 decfsz DELAY2,f
380 goto dloop
390 return
```

Enter the program in **TEXT** mode and compile it with the integrated PIC assembler. Now connect a PIC16F84A to the 11-pin interface of the PC-G850V(S) with the above PIC burner circuit (or an equivalent). Activate the PIC loader in the assembler menu (see chapter 12).

After successful programming, release the PIC from the burner and install it in the following test circuit:



This test circuit uses an external crystal <4MHz as a clock. This corresponds to the setting of XT_OSC within the configuration word (0x3FF1) of the example program (see specification of the PIC16F84A).

If the PIC has been correctly "burned" with the example program, the LED will start flashing as soon as a voltage source (+ V) is connected to the test circuit. The pushbutton is optional and, when pressed, will place the PIC in the RESET state ($\overline{\text{MCLR}}$ = LOW). The program execution will be stopped and the LED goes off. The flashing frequency is influenced on the hardware side by the quartz frequency and on the software side by the number of iterations of the external delay loop (line 310, value range = 1 … 255).

APPENDIX B: KEYBOARD COMMANDS

Keys	Description
ON	Turns on the power even if the unit is turned off by the `AUTO OFF` function. • Interrupt program execution (BREAK). • Interrupt the execution of commands such as `LOAD` or `BSAVE` during direct entry. • In **TEXT** and **C** modes, returns to the main menu or menu.
OFF	Turn off the device
BASIC	Switch to **BASIC**. Toggles between **RUN** or **PRO** mode.
SHIFT + ASBML	Switches to assembler, **CASL** or **PIC** mode
TEXT	Switch to **TEXT** mode
SHIFT + TEXT	Switch to the **C** compiler
SHIFT + ANS	Adjust screen contrast.
SHIFT	Activate second function of a key (displayed directly above the button). SHIFT must be held down to access.
CAPS	Toggle capital letters on and off. **CAPS** appears on the display when on. By default, **CAPS** is on after turning on the device. Toggle between large and small characters in Kanji mode.
カナ	Toggle Kanji mode on and off
TAB	Moves the cursor to the next tab position: • **BASIC**/**RUN**/**PRO**: moves at intervals of seven digits. • **TEXT**: moves by eight digits on the first press, six digits on the second press, and seven digits on each subsequent press.
▶	• Moves the cursor to the right. • Executes playback instructions. • Displays the cursor when it is not visible when content is displayed. • Clear an error message.
◀	• Delete a character in direct input. • Moves the cursor to the left. Otherwise, the same as the ▶ button
ANS	Retrieve results of the last calculation.
CONST	Enters a constant for calculations (**CONST** appears on the display). Pressing SHIFT + CONST (2nd F + CONST) will display the currently stored constant.
INS	Switch to insert mode. On initial startup of the computer, insert mode is off..

Keys	Description
SHIFT + DEL	Deletes the character at the cursor location.
BS	Delete the character directly to the left of the cursor.
2nd F	Activate the second function of a key (displayed directly above the key).
CLS	• Clears the screen • Clears a displayed error.
SHIFT + CA	Clears the display and resets the computer to default state. • Reset the `WAIT` time setting. • Reset the display (`USING`) format • Reset `TRON` status to `TROFF`. • Clears error conditions
↵	• Enter a line of code into the computer when writing programs. • Ask for manual calculation or direct execution of a command statement. • Resume a program that has been temporarily interrupted by the `INPUT` command.
SHIFT + P↔NP	Toggle **PRINT** mode on or off when an optional printer is connected.

The keys ▼ and ▲ have various functions, depending on the operating mode and the status of the computer as listed in the following table.

Mode	Status	▼	▲
RUN	Program execution	Not functional	
	Interrupted by `STOP` or BREAK.	Execution of following line and stop.	Hold to display executable or executed program line.
	Error condition	Not functional	Hold to display line with error.
	Trace mode ON.	Hold to run program.	Hold to display executable or executed program line.
PRO	(no program lines)		
	Program interrupted.	Display of the interrupted line.	
	Error condition	Display of line with error	
	Other condition.	Display of first line	Display of last line
	(line numbers displayed)		
		Display following program line.	Display previous program line

APPENDIX C: CALCULATION RANGES

Numerical Calculations

For a calculation involving x, the number x must fall within one of the ranges below:

$-1 \times 10^{100} < x \leq -1 \times 10^{-99}$ for negative x
$10^{-99} \leq x < 10^{100}$ for positive x
$x = 0$

The value of x displayed is limited by the number of digits on the display screen.

Functions

Command	Function	Range of x
SIN x COS x TAN x	sin x cos x tan x	DEG: $\lvert x \rvert < 10^{10}$ RAD: $\lvert x \rvert < \frac{\pi}{180} \times 10^{10}$ GRAD: $\lvert x \rvert < \frac{\pi}{9} \times 10^{10}$ Also, only for tan *x*: (n = integer) DEG: $\lvert x \rvert \neq 90\ (2n-1)$ RAD: $\lvert x \rvert \neq \frac{\pi}{2}\ (2n-1)$ GRAD: $\lvert x \rvert \neq 100\ (2n-1)$
ASN x ACS x	$\sin^{-1}$ x $\cos^{-1}$ x	$-1 \leq x \leq 1$
ATN x	$\tan^{-1}$ x	$\lvert x \rvert < 10^{100}$
HSN x HCS x HTN x	sinh x cosh x tanh x	$-227.9559242 \leq x \leq 230.2585092$
AHS x	$\sinh^{-1}$ x	$\lvert x \rvert < 10^{50}$
AHC x	$\cosh^{-1}$ x	$1 \leq x < 10^{50}$
AHT x	$\tanh^{-1}$ x	$\lvert x \rvert < 1$
LN x LOG x	ln x log x	$10^{-99} \leq x < 10^{100}$
EXP x	e^x	$-10^{100} < x \leq 230.2585092$
TEN x	10^x	$-10^{100} < x < 100$
RCP x SQU x CUB x SQR x CUR x	1/x x^2 x^3 $\sqrt{x}$ $\sqrt[3]{x}$	$\lvert x \rvert < 10^{100}, x \neq 0$ $\lvert x \rvert < 10^{50}$ $\lvert x \rvert < 2.154434690 \times 10^{33}$ $0 \leq x < 10^{100}$ $\lvert x \rvert < 10^{100}$
y ^ x	$(y^x=10^{x\log y})$	when y > 0, $-10^{100} < x \log y < 100$ when y = 0, x > 0 when y < 0, $\begin{cases} x = \text{integer or } \frac{1}{x} = \text{odd integer } (x \neq 0) \\ \quad \text{and } -10^{100} < x \log\lvert y \rvert < 100 \end{cases}$

Command	Function	Range of x
&H x		$0 \leq x \leq$ 2540BE3FF (x in hexadecimal) FDABF41C01 $\leq x \leq$ FFFFFFFFFF
POL (x, y)	$r = \sqrt{x^2 + y^2}$ $\theta = \tan \frac{y}{x}$	$(x^2 + y^2) < 10^{100}$ $\frac{y}{x} < 10^{100}$
REC (r,θ)	$x = r \cos \theta$ $y = r \sin \theta$	$r < 10^{100}$ $\mid r \sin \theta \mid < 10^{100}, \mid r \cos \theta \mid < 10^{100}$
NPR (n,r)	${}_nP_r$	$\frac{n!}{(n-r)!} < 10^{100}$, $0 \leq r \leq n \leq 9999999999$ n, r integers
NCR (n,r)	${}_nC_r$	$\frac{n!}{(n-r)!r!} < 10^{100}$, $0 \leq r \leq n \leq 9999999999$ n, r integers when $n - r < r$, $n - r \leq 69$ when $n - r \geq r$, $r \leq 69$
FACT x	n!	$0 \leq x \leq 69$
DEG x	DMS → DEG	$\mid x \mid < 10^4$
DMS x	DEG → DMS	$\mid x \mid < 10^4$

Statistical Calculations

Range $\mid x \mid < 10^{50}$ $1 \leq n < 10^{100}$ $\mid y \mid < 10^{50}$

Statistics For the following calculations, the absolute value of the intermediate and final results is less than 1 x 10^{100}. The denominator (divisor) is not 0. The result of √ is a positive number.

Σx Σx^2 Σy Σy^2

$$\bar{x} = \frac{\Sigma x}{n} \qquad \bar{y} = \frac{\Sigma y}{n}$$

$$sx = \sqrt{\frac{\Sigma x^2 - n x^2}{n-1}} \qquad sy = \sqrt{\frac{\Sigma y^2 - n y^2}{n-1}}$$

$$\sigma x = \sqrt{\frac{\Sigma x^2 - n x^2}{n}} \qquad \sigma y = \sqrt{\frac{\Sigma y^2 - n y^2}{n}}$$

$$r = \bar{y} - b\bar{x} \qquad b = \frac{Sxy}{Sxx}$$

$$r = \frac{Sxy}{\sqrt{Sxx \times Syy}} \qquad Sxx = \Sigma x^2 - \frac{(\Sigma x)^2}{n}$$

$$x' = \frac{y-a}{b} \qquad Syy = \Sigma y^2 - \frac{(\Sigma y)^2}{n}$$

$$y' = a + bx \qquad Sxy = \Sigma xy - \frac{\Sigma x \times \Sigma y}{n}$$

APPENDIX D: SPECIFICATIONS

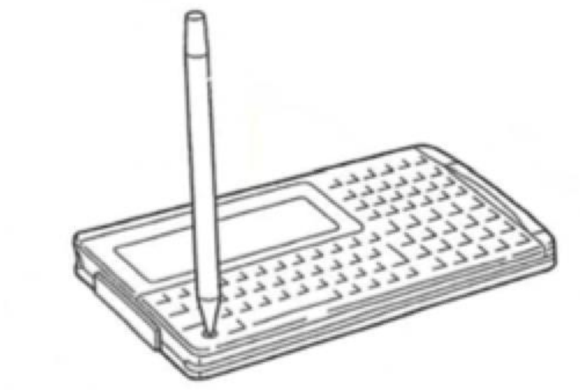
Device:	PC-G850V(S)	
Processor:	8-bit CMOS CPU (equivalent to Z80)	
Memory capacity:	System Internal:	2.3KB
	Fixed variable area:	208 bytes
	Program/data area:	30179 bytes
Stack:	Function stack:	16 level
	Data stack:	8 level
	Subroutine stack:	10 level stack for BASIC total of 90 bytes:
	REPEAT-UNTIL:	4 bytes per instance
	WHILE-WEND:	5 bytes per instance
	SWITCH-CASE:	6 bytes per instance
	FOR-NEXT:	18 bytes per instance (only one instance SWITCH-CASE allowed)
Operators:	Addition, subtraction, multiplication, division, trigonometric and inverse trigonometric functions, logarithmic and exponential functions, angle conversion, square and square root, power, sign, absolute, integers, pi, coordinate conversion, etc.	
Numerical precision:	10 digits (mantissa) + 2 digits (exponent)	
Editing functions:	Cursor right and left, line up and down, character insert, character delete. TEXT Editor, monitor for Z80 machine language.	
Interface Options:	Sharp-11Pin interface: CE-126P (printer) CE-T800 (PC data transmission cable) EA-129C (connection cable between 2 Sharp computers)	
Display:	Liquid crystal display	
	Text:	6 lines, 24 characters with 5x7 dot matrix
	Graphics:	48x144 pixels
Operating temperature:	0°C – 40°C (32° – 104°F)	

Power supply:	Four AAA batteries 6V DC 0.2W external power supply. (e.g. EA-23E)
Power consumption:	0.2W at 6.0V DC. Approximately 90 hours of continuous operation under normal conditions (based on 10 minutes of operation or program execution and 50 minutes of display per hour at a temperature of 20°C/68°F). The operating time may vary slightly depending on usage and type of battery used.
Dimensions:	196 (W) x 95 (D) x 20 (H) mm
Weight:	270g (G850VS: 260g)
Accessories:	Hard cover, 4 dry batteries. Operation Manual

APPENDIX E: RESETTING THE COMPUTER

If there is a problem with the computer, e.g. due to faulty programs, resetting the computer can help.

1. Press the ON button and then press the reset button under the SHIFT button with a ballpoint pen or similar device. Then release the reset button again.



2. Immediately after pressing the RESET button, the PC-G850V displays the following screen. If any other indication appears, the repeat the above procedure. The PC-G850V(S) asks for confirmation to clear the memory:

```
MEMORY CLEAR O.K.? (Y/N)
```

3. If you want to keep the data, press the **N** key

```
RUN MODE
>
```

If the computer still does not work properly, you can reset the computer to its factory default settings. This will delete all data on the computer. Repeat steps 1 and 2, then continue with step 4 below:

4. Press the $\boxed{\text{Y}}$ key. The following message flashes, indicating that the computer was initialized and all memory contents are deleted.

```
************************
*                      *
*      ALL CLEAR       *
*                      *
************************
```

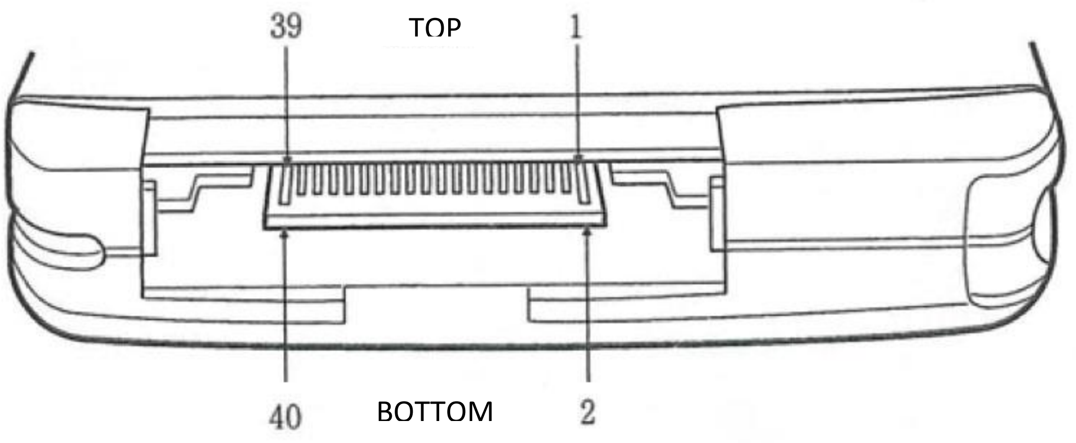
5. Press any key. The following display appears:

```
RUN MODE
>
```

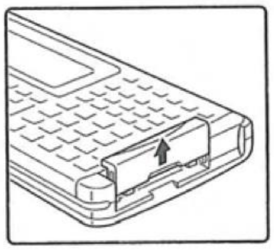
APPENDIX F: SYSTEM BUS

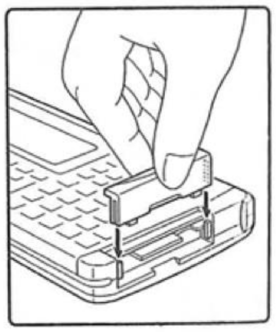
The PC-850V(S) provides a system bus for direct access to the 8-bit processor. Access to the system bus in on the right side of the computer.

Note: Depending on the battery power, the voltage at Vcc will be between 4-6V. Since the computer consists of CMOS components, the CMOS standard levels must be maintained.

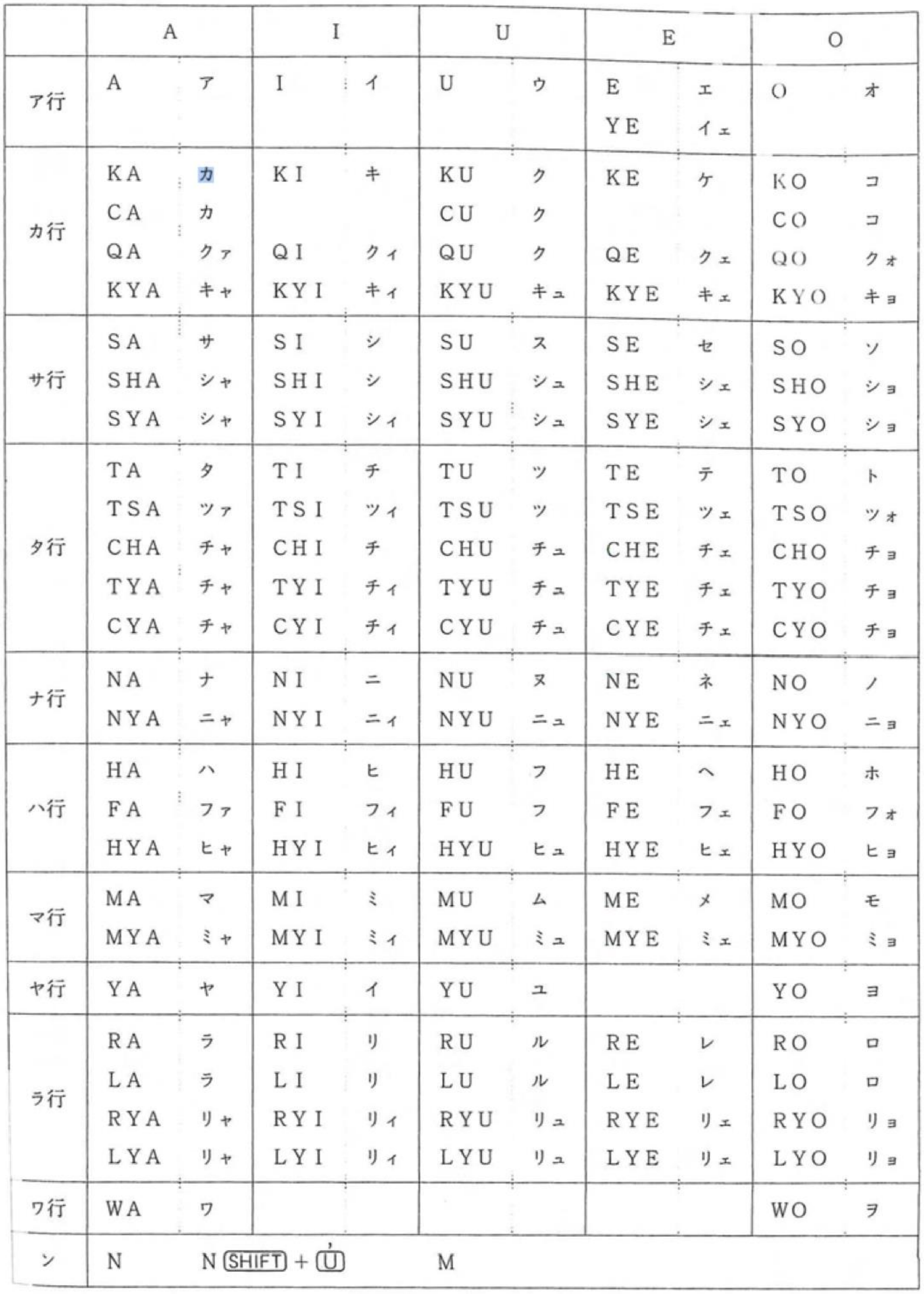


TOP		BOTTOM	
Signal	Pin	Pin	Signal
Vcc	1	2	Vcc
M1	3	4	MREQ
IORQ	5	6	IORESET
WAIT	7	8	INT1
WR	9	10	RD
BNK1	11	12	BNK0
CEROM2	13	14	CERAM2
D7	15	16	D6
D5	17	18	D4
D3	19	20	D2
D1	21	22	D0
A15	23	24	A14
A13	25	26	A12
A11	27	28	A10
A9	29	30	A8
A7	31	32	A6
A5	33	34	A4
A3	35	36	A2
A1	37	38	A0
GND	39	40	GND

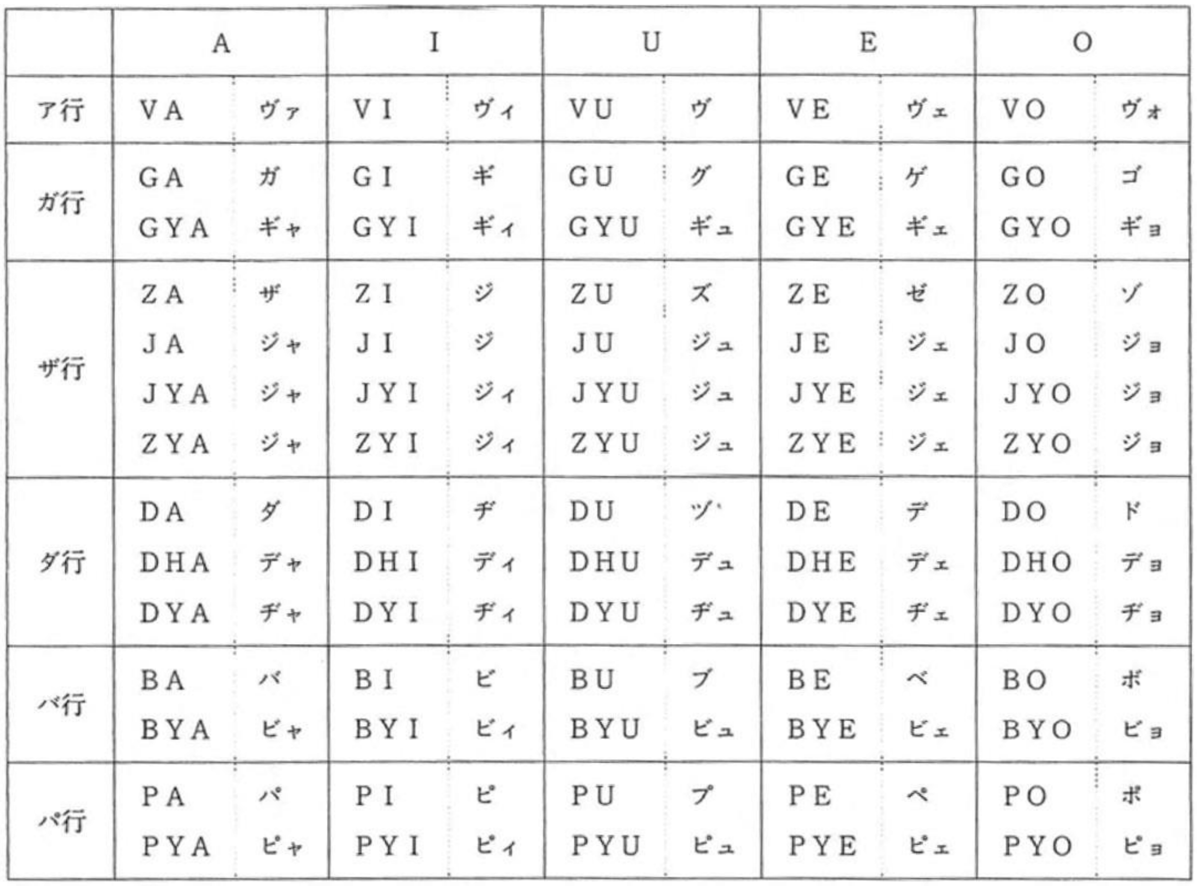




APPENDIX G: KANJI CONVERSION CHART



	A		I		U		E		O	
ア行	A	ア	I	イ	U	ウ	E	エ	O	オ
							YE	イェ		
カ行	KA	カ	KI	キ	KU	ク	KE	ケ	KO	コ
	CA	カ			CU	ク			CO	コ
	QA	クァ	QI	クィ	QU	ク	QE	クェ	QO	クォ
	KYA	キャ	KYI	キィ	KYU	キュ	KYE	キェ	KYO	キョ
サ行	SA	サ	SI	シ	SU	ス	SE	セ	SO	ソ
	SHA	シャ	SHI	シ	SHU	シュ	SHE	シェ	SHO	ショ
	SYA	シャ	SYI	シィ	SYU	シュ	SYE	シェ	SYO	ショ
タ行	TA	タ	TI	チ	TU	ツ	TE	テ	TO	ト
	TSA	ツァ	TSI	ツィ	TSU	ツ	TSE	ツェ	TSO	ツォ
	CHA	チャ	CHI	チ	CHU	チュ	CHE	チェ	CHO	チョ
	TYA	チャ	TYI	チィ	TYU	チュ	TYE	チェ	TYO	チョ
	CYA	チャ	CYI	チィ	CYU	チュ	CYE	チェ	CYO	チョ
ナ行	NA	ナ	NI	ニ	NU	ヌ	NE	ネ	NO	ノ
	NYA	ニャ	NYI	ニィ	NYU	ニュ	NYE	ニェ	NYO	ニョ
ハ行	HA	ハ	HI	ヒ	HU	フ	HE	ヘ	HO	ホ
	FA	ファ	FI	フィ	FU	フ	FE	フェ	FO	フォ
	HYA	ヒャ	HYI	ヒィ	HYU	ヒュ	HYE	ヒェ	HYO	ヒョ
マ行	MA	マ	MI	ミ	MU	ム	ME	メ	MO	モ
	MYA	ミャ	MYI	ミィ	MYU	ミュ	MYE	ミェ	MYO	ミョ
ヤ行	YA	ヤ	YI	イ	YU	ユ			YO	ヨ
ラ行	RA	ラ	RI	リ	RU	ル	RE	レ	RO	ロ
	LA	ラ	LI	リ	LU	ル	LE	レ	LO	ロ
	RYA	リャ	RYI	リィ	RYU	リュ	RYE	リェ	RYO	リョ
	LYA	リャ	LYI	リィ	LYU	リュ	LYE	リェ	LYO	リョ
ワ行	WA	ワ							WO	ヲ
ン	N	N SHIFT + U			M					



	A		I		U		E		O	
ア行	VA	ヴァ	VI	ヴィ	VU	ヴ	VE	ヴェ	VO	ヴォ
ガ行	GA	ガ	GI	ギ	GU	グ	GE	ゲ	GO	ゴ
	GYA	ギャ	GYI	ギィ	GYU	ギュ	GYE	ギェ	GYO	ギョ
ザ行	ZA	ザ	ZI	ジ	ZU	ズ	ZE	ゼ	ZO	ゾ
	JA	ジャ	JI	ジ	JU	ジュ	JE	ジェ	JO	ジョ
	JYA	ジャ	JYI	ジィ	JYU	ジュ	JYE	ジェ	JYO	ジョ
	ZYA	ジャ	ZYI	ジィ	ZYU	ジュ	ZYE	ジェ	ZYO	ジョ
ダ行	DA	ダ	DI	ヂ	DU	ヅ	DE	デ	DO	ド
	DHA	デャ	DHI	ディ	DHU	デュ	DHE	デェ	DHO	デョ
	DYA	ヂャ	DYI	ヂィ	DYU	ヂュ	DYE	ヂェ	DYO	ヂョ
バ行	BA	バ	BI	ビ	BU	ブ	BE	ベ	BO	ボ
	BYA	ビャ	BYI	ビィ	BYU	ビュ	BYE	ビェ	BYO	ビョ
パ行	PA	パ	PI	ピ	PU	プ	PE	ペ	PO	ポ
	PYA	ピャ	PYI	ピィ	PYU	ピュ	PYE	ピェ	PYO	ピョ

APPENDIX H: CHARACTER CODE TABLE

This table shows characters and their corresponding character codes used with the CHR$ and ASC commands. Each character code consists of 2 hexadecimal characters (or 8 binary bits).

Example:

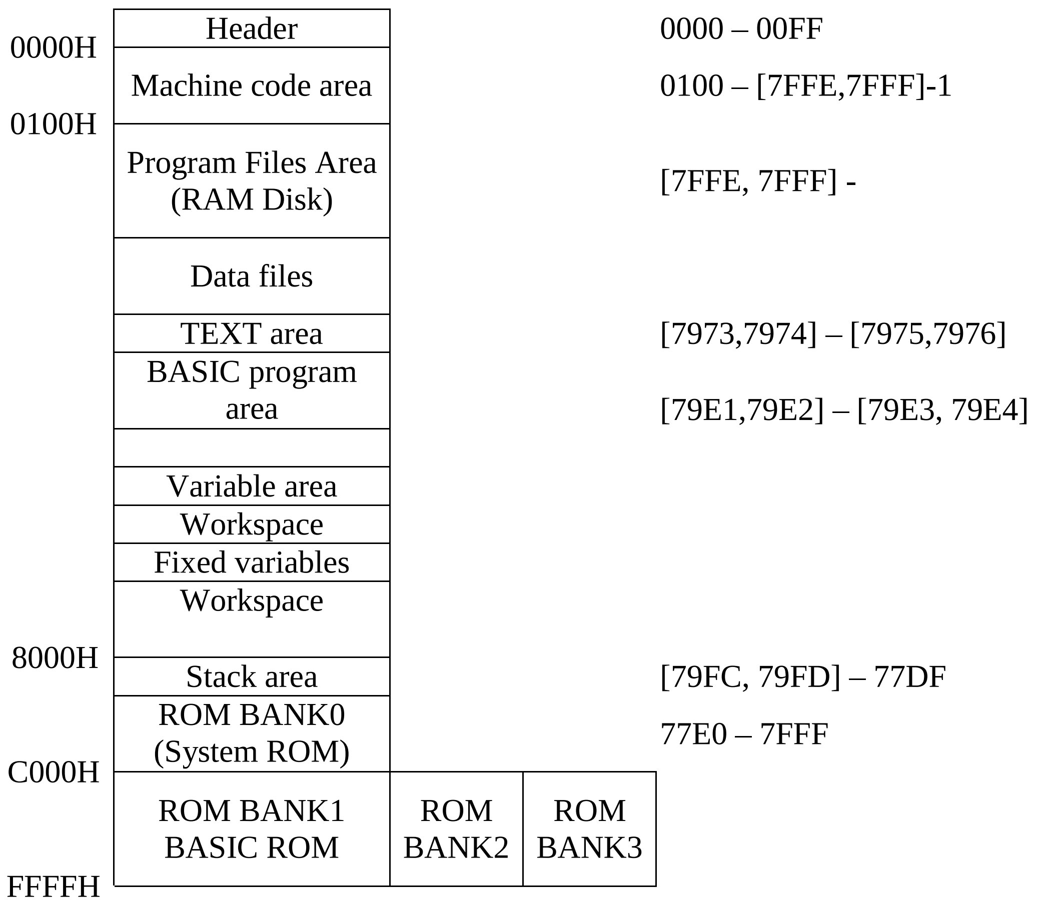
"A" = hexadecimal "41", decimal "65" and bin "01000001".
"P" = hexadecimal "50", decimal "80", and bin "01010000".

		0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	0				0	@	P	‘	p		┬		ｰ	ﾀ	ﾐ	＝	×
1	1			!	1	A	Q	a	q	▁	┴	｡	ｱ	ﾁ	ﾑ	╞	円
2	2			"	2	B	R	b	r	▂	┤	｢	ｲ	ﾂ	ﾒ	╪	年
3	3			#	3	C	S	c	s	▃	├	｣	ｳ	ﾃ	ﾓ	╡	月
4	4			$	4	D	T	d	t	▄	▔	､	ｴ	ﾄ	ﾔ	◢	日
5	5			%	5	E	U	e	u	▅	▁	･	ｵ	ﾅ	ﾕ	◣	時
6	6			&	6	F	V	f	v	▆	│	ｦ	ｶ	ﾆ	ﾖ	◥	分
7	7			'	7	G	W	g	w	▇	▕	ｧ	ｷ	ﾇ	ﾗ	◤	秒
8	8			(	8	H	X	h	x	▏	┌	ｨ	ｸ	ﾈ	ﾘ	♠	″
9	9			)	9	I	Y	i	y	▎	┐	ｩ	ｹ	ﾉ	ﾙ	♥	
10	A			*	:	J	Z	j	z	▍	└	ｪ	ｺ	ﾊ	ﾚ	♦	
11	B			+	;	K	[	k	{	▌	┘	ｫ	ｻ	ﾋ	ﾛ	♣	
12	C			,	<	L	¥	l	|	▋	╭	ｬ	ｼ	ﾌ	ﾜ	●	
13	D			-	=	M	]	m	}	▊	╮	ｭ	ｽ	ﾍ	ﾝ	○	
14	E			.	>	N	^	n	~	▉	╰	ｮ	ｾ	ﾎ	ﾞ	／	
15	F			/	?	O	_	o		┼	╯	ｯ	ｿ	ﾏ	ﾟ	＼	

Note: When printing with the CE-126P, the characters with the codes 129 (&H81) – 159 (&H9F), 224 (&HE0) – 231 (&HE7), 236 (&HEC) – 240 (&HF0), 245 (&HF5) – 248 (&HF8) are printed as spaces.

APPENDIX I: MEMORY MAP

Memory Area:



Note: Addresses in brackets represent the address in which the respective current memory position is stored.

APPENDIX J: ROM ADDRESSES

ROM Routines

Confirmed Addresses

Address	OpCode	Description
84F7		Roll the screen one line up.
871A		Initialize the serial interface (11-pin)
9249		Jumps to an address in a particular bank following the CALL statement
BCBE	STAT	Call STAT mode
BCDF		Reads a byte from the active serial interface to A (wait a short baud-rate-dependent time for the start bit)
BCE2		Reads a byte from the serial interface to A (waits for the start bit indefinitely)
BCE5		Reads a byte from the active serial interface to A (waits for the start bit indefinitely)
BCE8		Open (OPEN) the serial interface (11-pin)
BCEB		Close (CLOSE) the serial interface (11-pin)
BCEE		Add CR / LF to the file pointed to by HL
BCFI	CLRBAS	Starts the routine for “BASIC DELETE OK?”
BCF7	CLRTXT	Starts the routine for “TEXT DELETE OK?”
BCFD (88C1)	GETCHR	Reads a character from the keyboard into register A.
BD00	LDPSTR	Reads pixel string from position x, y of length B to address from HL (x position in E, y position in D). x = 0-5 and y = 0-23. 1 byte encodes 7 pixels and 5 bytes a character.
BD03	REGOUT	Displays values of all processor registers and waits for keystroke
BD09	AOUT	Displays value of the A-register and then waits for entry.
BD0F	HLOUT	Displays value of HL register and then waits for entry.
BD15		Reads an ASCII string from the serial interface to HL until EOF, EOL or an error (SCF) is detected.
BD2D	OFF	Power off (turns off the calculator)
BE53 (89BE)	INKEY	Tests if a key has been pressed and writes the key to A (INKEY function). A = 0 No key pressed. A = 52 several keys were pressed simultaneously (carry flag is set when a key was pressed)
BE62 (8440)	PUTCHR	Returns the character in register A. DE defines the x, y position (x-position in E, y-position in D)

Address	OpCode	Description
BE65	INSLN	Creates a blank line at the x, y position (in DE) (x position in E, y position in D)
BFAF		Writes the contents of register A to the serial interface
BFB2		Writes a string from HL to the serial interface. The transfer will be terminated when the character ZERO is received.
BFCD		Reads a character from the keyboard into register A. (wait until a key is pressed)
BFD0		Writes a pixel string whose address is in HL with the length B. The output starts from the x, y positions in DE (x position in E, y position in D)? X = 0-5 and y = 0-23. 1 byte encodes 7 pixels and 5 bytes a character. In contrast to the routines BFEE and BFF1, there is no line break
BFEE		Returns the character in A from position x, y in DE B times in succession. X = 0-5 and y = 0-23
BFF1		Display string of length B from address HL with x-y position in DE. If necessary, the string is wrapped at the end of the line and at the end of the display the LCD is scrolled up (the same behavior also with BFEE)
BFF4		Calling the RUN mode
C110		Power Off

BASIC Routines (Unconfirmed):

Address	Command	Description
C065		Initialize RAM (0000-003F)
C0FD		Ask if the memory should be cleared.
D7C3		HL points to the basic byte. The token string is passed in DE
F9BD		Converts the contents of register A to 2 hex numbers pointing to the HL
FFF7		Decodes Basic Byte in B. Returns the length in A and the address of the string in DE

Other Addresses (Unconfirmed)

Address	Description
0000	Jump to BFFA
0030	Jump to BD03
0038	RET
0066	RETN
USER area + 1A	Beginning of the ram disk (in MONITOR USER is changeable (default USER = FF) The file length (always 8 + 8 bytes) in the ram disk is in the two bytes after the filename
779C	Contrast. A change does not immediately cause the display to change. Example: 10 PRINT "now:"; PEEK (& H779C) 20 INPUT "change (0-31):"; A 30 POKE & H779C, A 40 OUT & H40, & H80 + A 50 GOTO 10
77E0	Start the system RAM area
7800-78CF	Variable range A-Z: 7800 = Z, 7 bytes each
78E7-78E8	Start address of the IO buffer
78EC	SIO Transmission mode bits: Bit 7: is received char EOT Bit 6: EOL matches (is complete) Bit 5: previous what CR Bit 4: check for EOL
78ED	Baud rate: 0x1 = 300, 0x2 = 600, 0x4 = 1200, 0x8 = 2400, 0x10 = 4800, 0x20 = 9600 - highest bit starting from bit 5 is relevant. all bits 0 ==> 300 baud
78EE	Parameters of the serial interface: Bit2: add -> CR, else if Bit0 add -> LF else add -> CRLF Bit 1: (set for CR LF) Bit 3: unused Bit 4: 1 + Bit 4 stop bits Bit 5: 0 = Odd 1 = Even parity if parity enabled Bit 6: 0 = no parity 1 = parity check / generation enabled Bit 7: 7 + Bit7 Data bits
78EF	Byte for the identification of the transmission end (EOT)
78F0	Auto power-off Pointer
7900	current bank ID mapped to C000-FFFF
7901-7904	Screen display annunciators: 7901: 00000111 | | +-Always |+---CAPS +----Cana

Address	Description
790D	VRAM display start position. first LCD row offset (0-7) [enables simple scrolling]
790E	Number of the last selected file in the ram disk
7912-7913	Beginning entry first file in the Ram disk (name)
7921	Current cursor line (0-3)
7922	Current cursor column (0-23)
7932	current interrupt mask at port 17H.
7966	INKEY1, keyboard code query, see Key Matrix
7973-7974	start text area
7975-7976	end text area
79B3-79B4	basic pointer
79B5-79B6	basic line being processed
79B9	current basic byte code
79C0-79C7	password
79E1-79E2	initial executable basic program
79E3-79E4	end executable basic Program
79FC-79FD	Lower end Basic variable area RAMTOP
79FE-79FF	Start executable basic program.
7A60-7A77	last line CAL calculation result
7A80-7A98	last CAL calculation result (exact to 11 digits)
7AA0-7AA1	program pointer
7AA2-7AA3	program pointer
7AA6-7AA7	currently used variable (?)
7AB6-7AB7	FOR pointer (?)
7AB8 -7AB9	variable pointer
7AC8-7AC9	FOR pointer
7ACA-7ACB	variable pointer
7ADC-7ADD	variable pointer
7B00-7B5F	sign in the (monitor) display
7BB0-7BC7	Display line CAL
7C00-7CFF	Input buffer, evaluated
7E00-7ED5	Basic string buffer
7E00-7ED5:	INKEY2, ASCII value like INKEY $
7EE8-	Line of input
7F40-7F4B	LCD line scratch data (12 characters)
7FFD-	TOP of Stack (decreased by PUSH) at most 178 bytes
7FFE-7FFF	address of first non-USER range, i.e. here USER + 1 is stored

Display Control Ports 40h, 41h:

Low-level control of the LCD is via two ports:

40h = Control-Port
41h = Data-Port

The resolution of the LCD for cursor positioning is 144x6 (144 columns and 6 rows). The rows have text resolution, the columns have graphic resolution.

The top left corner of the LCD is assigned coordinates (0,0). A vertical bit pattern (1 byte) in `GPRINT` format can used to a set cursor position through port 41h (bit 0 = lowest pixel, bit 7 = highest pixel). Cursor position automatically changes after output with shift of one column to the right.

This is the fastest way to access the LCD - it is controlled directly by the hardware LCD driver without accessing VRAM (as with the PC-1600). The latter, however, is easy to implement.

The control port can also be read. Bit 7 indicates whether the LCD hardware is BUSY. In this case, you must wait for the next `OUT` command.

The following values can be written to port 40h:

Value (hex)	Description	Notes
0n	Sets the lower-order nibble of the x-axis	Value range: $0 \leq n \leq F$
1n	Sets the higher-order nibble of the x-axis	Value range: $0 \leq n \leq 8$

2n	n = 4 LCD off
	n = 5 LCD on	

3n	n = CursorBlinkRate	30-3F: from fastest to slowest. Slowest blink rate still faster than the standard rate.
40-7F	VRAM display start position	The LCD has 144x48 dots but there are 144x64 dots in VRAM. 16 vertical points are always hidden. Example of scrolling: FOR A = 0 TO 63: OUT &H40, &H40 + A: NEXT
80-9f	LCD contrast	80–9F: from brightest to darkest. Usable values are from 80-8F. Outside this range, no difference in contrast seen.

Value (hex)	Description	Notes

An	n = 0 mirror mode off
	n = 1 mirror mode on
	n = 4 all pixels (mask) off
	n = 5 all pixels (mask) on
	n = 6 inverse off
	n = 7 Inverse on
	n = 8 Voltage on
	n = 9 Reduce voltage
	n = E all active pixels off
	n = F all active pixels on
		Sets display mode

Bn	Sets the y-axis	Value range: $0 \leq n \leq 5$

Cn	Partially switches on the display.
	n = 0 normal display
	n = 1 left 16 pixels
	n = 2 right 10 pixels, including mode
	n = 3 left 32 pixels
	n = 4 1 + 2
	n = 5 right 42 pixels + mode
		Turning on a display draws a line from the center to the When the display is turned on, a line ascending to the center appears, depending on the left and right. If bit 8 is set, it is a line descending to the middle

Dn	No function	
En	Unknown	
Fn	No function	

Example:

Assembler program for setting the cursor position and writing the 8-bit pattern:

```
DI
LD A, 0 <colLow>
OUT (40H), A
LD A, 1 <colHigh>
OUT (40H), A
LD A, B <row>
OUT (40H), A
LD A, <8bit -pattern>
OUT (41H), A
EI
```

The corresponding BASIC program:

```
10 GCURSOR (<colHigh> * 16 + <colLow>, 7 + <row> * 8)
20 GPRINT "<8bit-pattern>"
```

Example for inverting the display in BASIC:

```
OUT &H40,&HA7
```

Key Matrix

Output 11h, input 10h

Output / Input	7	6	5	4	3	2	1	0
7	)	R–CM	M+	Enter	↑	.	K	U
6	/	*	-	+	↓	M	J	Y
5	9	6	3	=	SPACE	N	H	T
4	8	5	2	.	TAB	B	G	R
3	7	4	1	0	Cana	V	F	E
2	π	INS	CON	ON	CAPS	C	D	W
1	BS	0	;	→	TXT	X	S	Q
0	P	I	L	←	BASIC	Z	A	OF

Output 12h, input 10h

Output / Input	7	6
7	CLS	MDF
6	F↔E	1 / x
5	tan	(
4	log	yx ^
3	ln	x2
2	cos	Ì
1	sin	→DEG
0	2ndF	nPr

Example:

```
10 CLS
20 LINE (7,16) - (18,7), B
30 A = 1
40 GCURSOR (8,15)
50 FOR B = 1 TO 7
60 OUT &H11, A
70 A = A * 2
80 GPRINT INP &H10;
90 NEXT
100 OUT &H11.0
110 FOR B = 1 TO 2
120 OUT &H12, B
130 GPRINT INP &H10;
140 NEXT
150 OUT &H12,0
160 GOTO 30
```

BIOS Key Values

Low \ High	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		OFF	Q	W	E	R	T	Y	U	A	S	D	F	G	H	J
1	K	Z	X	C	V	B	N	M	,	BASIC	TEXT	CAPS		TAB	SPACE	↓
2	↑	←	→	ANS	0	.	=	+	RETURN	L	;	CONST	1	2	3	-
3	M+	I	O	INS	4	5	6	*	R-CM	P	BS	π	7	8	9	/
4	)	nPr	→DEG	x2	yx ^	(	1 / x	MDF	2ndF	sin	cos	ln	log	tan	F⇔E	
5	CLS	ON														

+80h when pressing the shift key
52h When two or more keys are pressed

Conversion of BEEP Command Values to Tones:

Format: `BEEP` *`repeat`* `[,` *`level`*`] [,` *`length`*`]`

Repeat : number of beep tones. 0 to 65535
Level : Frequency of the buzzer. 230Hz ~ 8kHz (0 ~ 255). Optional.
Length : Duration of the sound. 0 to 65535, optional. Length can be calculated by the following equation: 1300000 / (166 +22 * level) Hz

C	C+	D	D+	E	F	F+	G	G+	A	A+	B
7											
21		18		15	14		12		10		8
49	46	43	40	37	35		30		26		22
105	99	93	87	82	77	72	68	64	60	56	52
219	200	194	182	172	162	152	143	135	127	119	112
										246	232

Example:

```
10 DATA 105,93,82,77,68,60,52,49
20 FOR A = 1 TO 8
30 READ B
40 C = 650000 / (166 + 22 * B)
50 BEEP 1, B, C
    60 EXT
```

Self-Test Mode

The following menu appears when you execute `OUT &H69,6`:

```
* PC-G850V V1.03 CHECK *
1:TOTAL      2:RAM
3:ROM        4:11PIN
5:LCD        6:KEY
7:SHOCK      8:AGING
9:L.B,ESD    0:CURRENT
```

Caution: Functions may clear the computer memory.

BASIC Code Table

High / Low	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																MON
1	RUN	NEW	CONT	PASS	LIST	LLIST	CLOAD	RENUM	LOAD			DELETE	FILES			LCOPY
2	CSAVE	OPEN	CLOSE	SAVE		RANDOMIZE	DEGREE	RADIAN	GRAD	BEEP	WAIT	GOTO	GRON	TROFF	CLEAR	USING
3	DIM	CALL	POKE	GPRINT	PSET	PRESET					ERASE	LFILES	KILL			
4						OUT			PIOSET	PIOPUT	SPOUT	SPINP	HDCOPY	ENDIF	REPEAT	UNTIL
5	CLS	LOCATE	TO	STEP	THEN	ON	IF	FOR	LET	REM	END	NEXT	STOP	READ	DATA	
6	PRINT	INPUT	GOSUB	LNINPUT	LPRINT	RETURN	RESTORE		GCURSOR	LINE						CIRCLE
7	PAINT	OUTPUT	APPEND	AS			ELSE				WHILE	WEND	SWITCH	CASE	DEFAULT	ENDSWITCH
8	MDF	REC	POL				TEN	RCP	SQR	CUR	HSN	HCS	HTN	AHS	AHC	AHT
9	FACT	LN	LOG	EXP	SQR	SIN	COS	TAN	INT	ABS	SGN	DEG	DMS	ASN	ACS	ATN
A	RND	AND	OR	NOT	PEEK	XOR	INP		PIOSET					POINT	PI	FRE
B	EOF		LOF				NCR	NPR								CUB
C							MOD	FIX								
D	ASC	VAL	LEN	VDEG												
E										INKEY$	MID$	LEFT$		RIGHT$		
F	CHR$	STR$	HEX$	DMS$												

APPENDIX K: ERROR MESSAGES

If an error in BASIC occurs, one of the following codes will be displayed. For errors that occur during program execution, the line number where the error occurred is also displayed.

Error code	Description
10	Syntax error.
12	Illegal command for specified mode (RUN/PRO).
13	Illegal CONT command.
14	Program does not exist for PASSWORD.
15	Illegal address for BSAVE M.
20	Overflow error (>10^{100}).
21	Divide by zero.
22	Illegal operation.
30	Array already assigned.
31	Undimensioned array.
32	Array index overflow.
33	Out of range error.
40	Name/line number nonexistent.
41	Illegal line number.
43	Error with RENUM/LCOPY
44	End line < start line
50	Stack depth exceeded (for GOSUB, FOR, REPEAT, WHILE, and SWITCH)
51	Missing GOTO.
52	Missing NEXT.
53	Missing READ.
54	Buffer overflow.
55	String/line > 255 bytes.
60	Out of memory.
61	Missing ENDIF
62	Missing REPEAT.
63	Missing WEND.
64	Missing WEND.
66	Extra CASE/DEFAULT statement.
68	Missing ENDSWITCH.
69	Missing SWITCH.
70	Cannot print with current USING format.
71	Illegal USING format.
72	I / O error.

Error code	Description
77	File overflow.
80	Checksum error.
81	Timeout error.
82	Type mismatch error with BLOAD
83	Type mismatch error with INPUT#
84	Printer error
85	Device/file not opened before use
86	Device/file number already in use.
87	End of file.
90	Type mismatch error.
91	Type mismatch error.
92	Incorrect password.
93	Password protected
94	File not found
95	Illegal file name.
96	File type mismatch (BASIC vs. TEXT)
97	Greater than 255 files

APPENDIX L: Z80 PROGRAMMING REFERENCE

This chapter summarizes the instructions for the Z80 processor. It does not replace a manual for Z80 programming. It should only serve as a reference.

Z80 registers and flags

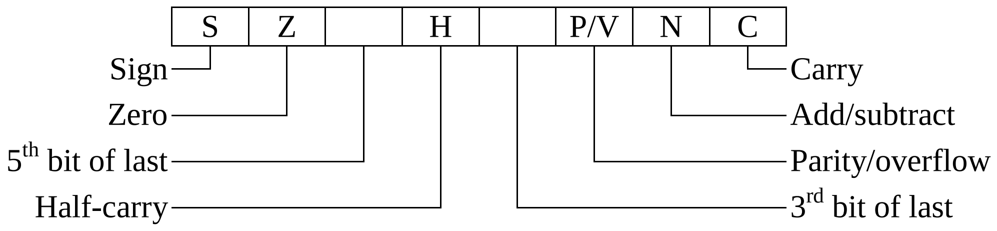
The Z80 has various 8-bit and 16-bit registers. During execution, some 8-bit registers merge into 16-bit registers (which can easily be recognized by the register names).

8-bit registers	: A, B, C, D, E, H, L
16-bit registers	: IX, IY, BC, DE, HL

The Z80 processor provides duplicates of the 8-bit registers so you have a pair of registers available. With one command you can exchange register assignments.

The Z80 also has a number of flags. Flags are 1-bit registers in which current states can be displayed. The flags are held in the F-register (from bit 7-0):

S	: sign flag (1 if negative)
Z	: zero flag (1 if result 0)
H	: auxiliary flag (also called half-carry flag)
P	: parity flag (1 if overflow)
N	: subtraction flag (1 if subtraction in accumulator)
C	: carry flag (carry flag or CY, 1 if overflow)



Z80 Instruction set

Abbreviations

r, r’	8-bit registers A, B, C, D, E, H, L
dd	16-bit registers BC, DE, HL, SP
qq	16-bit registers AF, BC, DE, HL
pp	16-bit registers BC, DE, SP
n	8-bit constant
nn	16-bit constant, address
d	Offset for indirect addressing in the range -128 to 127
b	Bit to be used in single-bit instructions $0 \leq b \leq 7$
m, M	Contents of memory addressed by HL (L contains bits 0-7; H bits 8-15
p	Value of 00h, 08h, 10h, 18h, 20h, 28h, 30h, or 38h
CY	Carry flag
T	Number of clock cycles

8-bit Load Instructions

These instructions move 8-bit data between registers or between registers and memory. The first argument in the operand field is the destination address and the second is the source. The contents of the source address are not changed.

Instruction	T	Operation	SZHPNC
LD r, r’	4	load register r’ into register r	------
LD r, n	7	load constant n into register r	------
LD r, m	7	load memory addressed by HL into register r	------
LD r, (IX + d)	19	load memory addressed by IX + offset d into register r	------
LD r, (IY + d)	19	load memory addressed by IY + offset d into register r	------
LD m, r	7	load register r into memory addressed by HL	------
LD (IX + d), r	19	load register r into memory addressed by IX + offset d	------
LD (IY + d), r	19	load register r into memory addressed by IY + offset d	------
LD m, n	10	load constant n into memory addressed by HL	------
LD (IX + d), n	19	load constant n into memory addressed by IX + offset d	------
LD (IY + d), n	19	load constant n into memory addressed by IY + offset d	------
LD A, (BC)	7	load memory addressed by register BC into register A (accumulator)	------
LD A, (DE)	7	load memory addressed by register DE into register A (accumulator)	------
LD A, (nn)	13	load memory addressed by nn into register A (accumulator)	------
LD (BC), A	7	load register A (accumulator) into memory addressed by register BC	------
LD (DE), A	7	load register A (accumulator) into memory addressed by register DE	------
LD (nn), A	13	load register A (accumulator) into memory addressed by nn	------
LD A, I	9	load register I (interrupt) into register A (accumulator)	**0F0-
LD A, R	9	load register R (refresh) into register A (accumulator)	**0F0-
LD I, A	9	load register A (accumulator) into register I (interrupt)	------
LD R, A	9	load register A (accumulator) into register R (refresh)	------

16-bit Load Instructions

These instructions move 16-bit data between registers or between registers and memory. The first argument in the operand field is the destination address and the second is the source. The contents of the source address are not changed.

Special 16-bit instructions include the PUSH and POP. 16-bit data from double registers can be pushed into the stack or taken back to the double register.

All 16-bit data is stored in Intel (little-endian) order (least significant byte first).

Instruction	T	Operation	SZHPNC
LD dd, nn	10	load constant nn into double register	------
LD IX, nn	14	load constant nn into index register IX	------
LD IY, nn	14	load constant nn into index register IY	------
LD HL, (nn)	16	load memory addressed by nn, nn+1 into double register HL (nn→L, nn+1→H)	------
LD pp, (nn)	20	load memory addressed by nn, nn+1 into double register pp (nn→L, nn+1→H)	------
LD IX, (nn)	20	load memory addressed by nn, nn+1 into double register IX (nn→X, nn+1→I)	------
LD IY, (nn)	20	load memory addressed by nn, nn+1 into double register HL (nn→Y, nn + 1→I)	------
LD (nn), HL	16	load contents of double register HL into addresses nn, nn + 1 (L→nn, H→nn+1)	------
LD (nn), pp	20	load contents of double register pp into addresses nn, nn + 1 (L→nn, H→nn+1)	------
LD (nn), IX	20	load contents of double register IX into addresses nn, nn + 1 (X→nn, I→nn+1)	------
LD (nn), I	20	load contents of double register IY into addresses nn, nn + 1 (Y→nn, I→nn+1)	------
LD SP, HL	6	load double register HL into SP (stack pointer)	------
LD SP, IX	10	load double register IX into SP (stack pointer)	------
LD SP, IY	10	load double register IY into SP (stack pointer)	------
PUSH qq	11	load double register qq into the stack DEC SP; LD (SP), H; DEC SP; LD (SP), L	------
PUSH IX	15	load double register IX into the stack DEC SP; LD (SP), I; DEC SP; LD (SP), X	------
PUSH IY	15	load double register IY into the stack DEC SP; LD (SP), I; DEC SP; LD (SP), Y	------
POP qq	10	load last value on the stack into double register qq LD L, (SP); INC SP; LD H, (SP); INC SP	------
POP IX	14	load last value on the stack into double register IX LD X, (SP); INC SP; LD I, (SP); INC SP	------
POP IY	14	load last value on the stack into double register IY LD Y, (SP); INC SP; LD I, (SP); INC SP	------

8-bit Arithmetic and Logic Instructions

These instructions work with the accumulator (A register) and other registers or memory addresses. The result of these instructions is stored in the accumulator (A register).

Instruction	T	Operation	SZHPNC
ADD r	4	add register r to the accumulator	***V0*
ADD m	7	add memory addressed by register HL to the accumulator	***V0*
ADD n	7	add constant n to the accumulator	***V0*
ADD (IX + d)	19	add memory addressed by register IX + offset d to the accumulator	***V0*
ADD (IY + d)	19	add memory addressed by register IX + offset d to the accumulator	***V0*
ADC r	4	add register r + carry flag to the accumulator	***V0*
ADC m	7	add memory addressed by m + carry flag to the accumulator	***V0*
ADC n	7	add constant n + carry flag to the accumulator content	***V0*
ADC (IX + d)	19	add contents of memory addressed by register IX + offset d and carry flag to the accumulator	***V0*
ADC (IY + d)	19	add contents of memory addressed by register IY + offset d and carry flag to the accumulator	***V0*
SUB r	4	subtract contents of register r from the accumulator	***V1*
SUB m	7	subtract memory addressed by register HL from the accumulator	***V1*
SUB n	7	subtract constant n from the accumulator	***V1*
SUB (IX + d)	19	subtract memory addressed by register IX + offset d from the accumulator	***V1*
SUB (IY + d)	19	subtract memory addressed by register IY + offset d from the accumulator	***V1*
SBC r	4	subtract register r + carry flag from the accumulator	***V1*
SBC m	7	subtract memory addressed by m + carry flag from the accumulator	***V1*
SBC n	7	subtract constant n + carry flag from the accumulator	***V1*
SBC (IX + d)	19	subtract memory addressed by IX + offset d and carry flag from the accumulator	***V1*
SBC (IY + d)	19	subtract memory addressed by IY + offset d and carry flag from the accumulator	***V1*
AND r	4	logical AND of register r and the accumulator	**1p00
AND m	7	logical AND of memory addressed m and the accumulator	**1p00
AND n	7	logical AND of constant n and the accumulator	**1p00
AND (IX + d)	19	logical AND of memory addressed by register IX + offset d and the accumulator	**1p00
AND (IY + d)	19	logical AND of memory addressed by register IY + offset d and the accumulator	**1p00
OR r	4	logical OR of register r and the accumulator	**0p00
OR m	7	logical OR of memory addressed m and the accumulator	**0p00
OR n	7	logical OR of constant n and the accumulator	**0p00

Instruction	T	Operation	SZHPNC
OR (IX + d)	19	logical OR of memory addressed by register IX + offset d and the accumulator	**0p00
OR (IY + d)	19	logical OR of memory addressed by register IY + offset d and the accumulator	**0p00
XOR r	4	logical XOR of register r and the accumulator	**0p00
XOR m	7	logical XOR of memory addressed m and the accumulator	**0p00
XOR n	7	logical XOR of constant n and the accumulator	**0p00
XOR (IX + d)	19	logical XOR of memory addressed by register IX + offset d and the accumulator	**0p00
XOR (IY + d)	19	logical XOR of memory addressed by register IY + offset d and the accumulator	**0P00
CP r	4	compare register r with accumulator Zero-Flag: 1 → contents are identical 0 → contents are different Carry-Flag: 1 → accumulator smaller 0 → accumulator equal or greater	***V1*
CP m	7	compare memory addressed by register m with accumulator	***V1*
CP n	7	compare constant n with accumulator	***V1*
CP (IX + d)	19	compare memory addressed by register IX + offset d with accumulator	***V1*
CP (IY + d)	19	compare memory addressed by register IY + offset d with accumulator	***V1*
INC r	4	increase value of register r by one	***V0-
INC m	11	increase value of memory addressed by m by one	***V0-
INC (IX + d)	23	increase value of memory addressed by register IX + offset by one	***V0-
INC (IY + d)	23	increase value of memory addressed by register IY + offset by one	***V0-
DEC r	4	decrease value of register r by one	***V1-
DEC m	11	decrease value of memory addressed by m by one	***V1-
DEC (IX + d)	23	decrease value of memory addressed by register IX + offset by one	***V1-
DEC (IY + d)	23	decrease value of memory addressed by register IY + offset by one	***V1-
DAA	4	BCD correction of accumulator using flags	***p-*
CPL	4	bitwise 1's complement of accumulator	--1-1-
NEG	8	subtract the accumulator from zero (2's complement, bitwise negate, increase by 1)	***V1*
CCF		invert carry flag	--x-0*
SCF	4	Set carry flag to 1	--0-01

16-bit Arithmetic Instructions

These instructions work similarly to the 8-bit arithmetic instructions, but with double registers. As the accumulator is not a 16-bit register, HL, IX or IY is used.

Instruction	T	Operation	SZHPNC
ADD HL, dd	11	add register dd to register HL	--x-0*
ADD IX, IX	15	add register IX to itself (doubling)	--x-0*
ADD IY, IY	15	add register IY to itself (doubling)	--x-0*
ADD IX, pp	15	add register pp to register IX	--x-0*
ADD IY, pp	15	add register pp to register IY	--x-0*
ADC HL, dd	15	add register dd + carry flag to register HL	**xV0*
SBC HL, dd	15	subtract register dd + carry flag to register HL	**xV1*
INC dd	6	increment register dd by one	------
INC IX	10	increment register IX by one	------
INC IY	10	increment register IY by one	------
DEC dd	6	decrement register dd by one	------
DEC IX	10	decrement register IX by one	------
DEC IY	10	decrement register IY by one	------

Register Exchange Instructions

These instructions are used to exchange 16-bit register contents. It also allows backup of primary registers with their corresponding “shadow” registers.

Instruction	T	Operation	SZHPNC
EX DE, HL	4	Exchange registers DE and HL	------
EX AF, AF’	4	Exchange registers AF and AF’	------
EXX	4	Exchange registers with shadow registers BC↔BC’ DE↔DE’ HL↔HL’	------
EX (SP), HL	19	Exchange contents of register HL with last value in the stack SP+1→H, SP→L	------
EX (SP), IX	23	Exchange contents of register IX with last value in the stack SP+1→I, SP→X	------
EX (SP), IY	23	Exchange contents of register IY with last value in the stack SP+1→I, SP→Y	------

Branch Instructions

These include conditional and unconditional jumps. The destination of jumps can be specified using absolute or relative addressing. Range of relative address is restricted from -126 to +129 bytes. For conditional jumps, a flag must be specified as an operands and the corresponding flag bit is tested. Depending on this test, the jump is either executed or ignored.

Instruction	T	Operation	SZHPNC
JP nn	10	jump to address nn	------
JP NZ, nn	10	jump to address nn if zero bit cleared (0)	------
JP Z, nn	10	jump to address nn if zero bit set (1)	------
JP NC, nn	10	jump to address nn if carry bit cleared (0)	------
JP C, nn	10	jump to address nn if carry bit set (1)	------
JP PO, nn	10	jump to address nn if parity/overflow bit cleared (0)	------
JP PE, nn	10	jump to address nn if parity/overflow bit set (1)	------
JP P, nn	10	jump to address nn if sign bit cleared (0)	------
JP M, nn	10	jump to address nn if sign bit set (1)	------
JR nn	10	jump to relative address nn	------
JR NZ, nn	12	jump to relative address nn if zero bit cleared (0)	------
JR Z, nn	12/7	jump to relative address nn if zero bit set (1)	------
JR NC, nn	12/7	jump to relative address nn if carry bit cleared (0)	------
JR C, nn	12/7	jump to relative address nn if carry bit set (1)	------
JP m	4	jump to address specified by register HL	------
JP (IX)	8	jump to address specified by register IX	------
JP (IY)	8	jump to address specified by register IY	------
DJNZ nn	13/8	decrement register B and jump to relative address nn if B≠0	------

Subroutines

As with the jump instructions, there are conditional and unconditional instructions. The subroutine call operates by storing the return address following the CALL command on the stack. If the subroutine is terminated with the RET command, the return address is loaded from the stack and execution continues from the return address.

Instruction	T	Operation	SZHPNC
CALL nn	17	call subroutine at address nn	------
CALL NZ, nn	17/10	call subroutine at address nn if zero flag cleared (0)	------
CALL Z, nn	17/10	call subroutine at address nn if zero flag set (1)	------
CALL NC, nn	17/10	call subroutine at address nn if carry flag cleared (0)	------
CALL C, nn	17/10	call subroutine at address nn if carry flag set (1)	------
CALL PO, nn	17/10	call subroutine at address nn if parity/overflow flag cleared (0)	------
CALL PE, nn	17/10	call subroutine at address nn if parity/overflow flag set (1)	------
CALL P, nn	17/10	call subroutine at address nn if sign flag cleared (0)	------
CALL M, nn	17/10	call subroutine at address nn if sign flag set (1)	------
RST p	11	call subroutine at restart address (valid addresses: 00h, 08h, 10h, 18h, 20h, 28h, 30h, 38h.)	------
RET	10	unconditional return from a subroutine	------
RET NZ	11/5	return from subroutine if zero flag cleared (0)	------
RET Z	11/5	return from subroutine if zero flag set (1)	------
RET NC	11/5	return from subroutine if carry flag cleared (0)	------
RET C	11/5	return from subroutine if carry flag set (1)	------
RET PO	11/5	return from subroutine if parity/overflow flag cleared (0)	------
RET PE	11/5	return from subroutine if parity/overflow flag set (1)	------
RET P	11/5	return from subroutine if sign flag cleared (0)	------
RET M	11/5	return from subroutine if sign flag set (1)	------
RETI	14	return from interrupt	------
RETN	14	return from non-maskable interrupt	------

Shift Instructions

These instructions allows bitwise shifting of values in the accumulator (A-register), register, or in memory. The bit shifted out of the byte is stored in the carry flag.

Instruction	T	Operation	SZHPNC
RLCA	4	rotate accumulator one bit to the left, bit 7 becomes bit 0	--0-0*
RRCA	4	rotate accumulator one bit to the right, bit 0 becomes bit 7	--0-0*
RLA	4	rotate accumulator one bit to the left, bit 7 becomes the carry flag and the carry flag becomes bit 0	--0-0*
RRA	4	rotate accumulator one bit to the right, bit 0 becomes the carry flag and the carry flag becomes bit 7	--0-0*
RLC r	8	rotate register r one bit to the left, bit 7 becomes bit 0	**0p0*
RLC m	15	rotate memory addressed by register m one bit to the left, bit 7 becomes bit 0	**0p0*
RLC (IX + d)	23	rotate memory addressed by register IX + offset d one bit to the left, bit 7 becomes bit 0	**0p0*
RLC (IY + d)	23	rotate memory addressed by register IY + offset d one bit to the left, bit 7 becomes bit 0	**0p0*
RRC r	8	rotate register r one bit to the right, bit 0 becomes bit 7	**0p0*
RRC m	15	rotate memory addressed by register m one bit to the right, bit 0 becomes bit 7	**0p0*
RRC (IX + d)	23	rotate memory addressed by register IX + offset d one bit to the right, bit 0 becomes bit 7	**0p0*
RRC (IY + d)	23	rotate memory addressed by register IY + offset d one bit to the right, bit 0 becomes bit 7	**0p0*
RL r	8	rotate register r one bit to the left, bit 7 becomes the carry flag and the carry flag becomes bit 0	**0p0*
RL m	15	rotate memory addressed by register m one bit to the left, bit 7 becomes the carry flag and the carry flag becomes bit 0	**0p0*
RL (IX + d)	23	rotate memory addressed by register IX + offset d one bit to the left, bit 7 becomes the carry flag and the carry flag becomes bit 0	**0p0*
RL (IY + d)	23	rotate memory addressed by register IY + offset d one bit to the left, bit 7 becomes the carry flag and the carry flag becomes bit 0	**0p0*
RR r	8	rotate register r one bit to the right, bit 0 becomes the carry flag and the carry flag becomes bit 7	**0p0*
RR m	15	rotate memory addressed by register m one bit to the right, bit 0 becomes the carry flag and the carry flag becomes bit 7	**0p0*
RR (IX + d)	23	rotate memory addressed by register IX + offset d one bit to the right, bit 0 becomes the carry flag and the carry flag becomes bit 7	**0p0*

Instruction	T	Operation	SZHPNC
RR (IY + d)	23	rotate memory addressed by register IY + offset d one bit to the right, bit 0 becomes the carry flag and the carry flag becomes bit 7	**0p0*
SLA r	8	arithmetic shift left register r one bit, bit 7 becomes the carry flag, bit 0 is 0.	**0p0*
SLA m	15	arithmetic shift left memory addressed by register m one bit, bit 7 becomes the carry flag, bit 0 is 0.	**0p0*
SLA (IX + d)	23	arithmetic shift left memory addressed by register IX + offset d one bit, bit 7 becomes the carry flag, bit 0 is 0.	**0p0*
SLA (IY + d)	23	arithmetic shift left memory addressed by register IY + offset d one bit, bit 7 becomes the carry flag, bit 0 is 0.	**0p0*
SRA r	8	arithmetic shift right register r one bit, bit 0 becomes the carry flag, bit 7 is unchanged.	**0p0*
SRA m	15	arithmetic shift right memory addressed by register HL one bit, bit 0 becomes the carry flag, bit 7 is unchanged.	**0p0*
SRA (IX + d)	23	arithmetic shift right memory addressed by register IX + offset d one bit, bit 0 becomes the carry flag, bit 7 is unchanged.	**0p0*
SRA (IY + d)	23	arithmetic shift right memory addressed by register IY + offset d one bit, bit 0 becomes the carry flag, bit 7 is unchanged.	**0p0*
RLD	18	4-bit leftward rotation of a 12-bit number whose 4 most significant bits are the 4 least significant bits of register A (accumulator) and its 8 least significant bits are in register HL	**0p0*
RRD	18	4-bit rightward rotation of a 12-bit number whose 4 most significant bits are the 4 least significant bits of register A (accumulator) and its 8 least significant bits are in register HL	**0p0*

Bit Commands

These instructions allow the testing, setting or deletion individual bits in registers or in memory.

Instructions	T	Operation	SZHPNC
BIT b, r	8	test bit b in register r, the inverse of bit b is written to the Z flag.	x*1x0-
BIT b, m	12	test bit b in memory addressed by m, the inverse of bit b is written to the Z flag.	x*1x0-
BIT b, (IX+d)	20	test bit b in memory addressed by IX + offset d, the inverse of bit b is written to the Z flag.	x*1x0-
BIT b, (IY+d)	20	test bit b in memory addressed by IY + offset d, the inverse of bit b is written to the Z flag.	x*1x0-
SET b, r	8	set bit b in register r	------
SET b, m	12	set bit b in memory addressed by m	------
SET b, (IX+d)	20	set bit b in memory addressed by IX + offset d	------
SET b, (IY+d)	20	set bit b in memory addressed by IY + offset d	------
RES b, r	8	clear bit b in register r	------
RES b, m	12	clear bit b in memory addressed by m	------
RES b, (IX+d)	20	clear bit b in memory addressed by IX + offset d	------
RES b, (IY+d)	20	clear bit b in memory addressed by IY + offset d	------

CPU Commands

These instructions control the CPU interrupts.

Instructions	T	Operation	SZHPNC
NOP	4	no operation	------
STOP	4	executes NOP instructions until interrupt or RESET	------
DI	4	disables interrupts in mode 1 and mode 2	------
EGG	4	enables interrupts in mode 1 and mode 2	------
IM 0	8	set interrupt mode 0 (external)	------
IM 1	8	set interrupt mode 1 (OS)	------
IM 2	8	set interrupt mode 2 (user)	------

Copy/Compare

These instructions can copy blocks of memory or search for a particular byte. The search ends when the byte is found or the end of the memory area has been reached.

Instruction	T	Operation	SZHPNC

LDI	16	copies from memory addressed by HL to memory addressed by DE, increments HL and DE, decrement BC
		if BC = 0 → P = 0
		if BC ≠ 0 → P = 1
			--0*0-

LDIR	21	repeats LDI until BC = 0	--000-

LDD	16	copies from the memory addressed by HL to memory addressed by DE, decrement registers DE, HL and BC
		if BC = 0 → P = 0
		if BC ≠ 0 → P = 1
			--0*0-

LDDR	21	repeats LDD until BC = 0	--000-

CPI	16	compare memory addressed by HL with register A (accumulator)
		if A = (HL) → Z = 1
		if A ≠ (HL) → Z = 0, increment HL, decrement BC
		if BC = 0 → P = 0
		if BC ≠ 0 → P = 1
			****1-

CPIR	21	repeats CPI until BC = 0 or A = (HL)	****1-

CPD	16	compare memory addressed by HL with register A (accumulator)
		if A = (HL) → Z = 1
		if A ≠ (HL) → Z = 0, decrement HL, BC
		if BC = 0 → P = 0
		if BC ≠ 0 → P = 1
			****1-

CPDR	21	repeat CPD until BC = 0 or A = (HL)	****1-

Input/Output

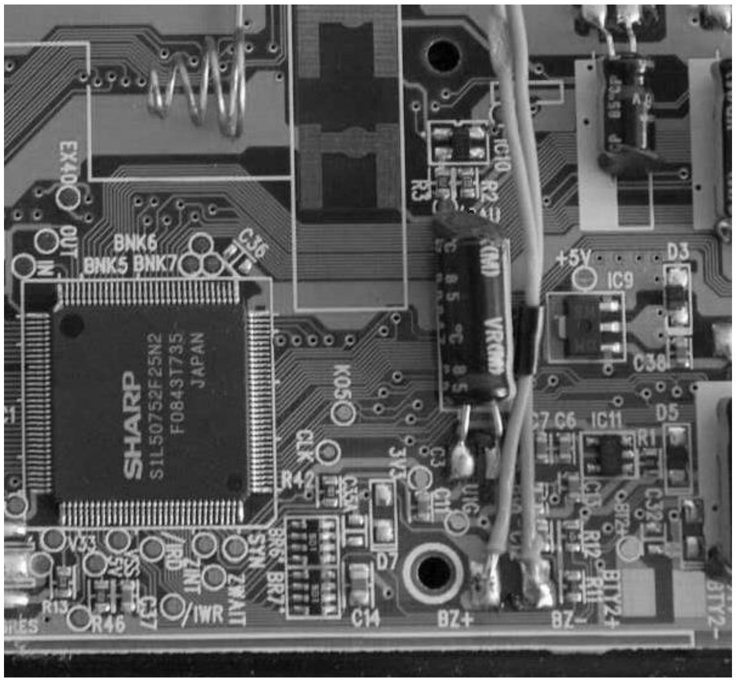
These instructions allow for the exchange of data between registers/memory with external blocks. The external block is accessed via a port address (8-bit value). Depending on the instruction, this port address is either specified directly (as a constant) or located in register C. Similar to copy, instructions for transferring blocks of memory are available.

If the C register is used for addressing, the B register is used to hold the more significant bits of the address bus.

Instructions	T	Operation	SZHPNC
IN A, (n)	11	reads from hardware port (n) to the accumulator	------
IN r, (C)	12	reads from hardware port in register C to register r	**0P0-
INI	16	reads hardware port (C) and writes the result to memory addressed by HL, increment HL, decrement B if B = 0 → Z = 1 otherwise Z = 0	x*xx1-
INIR	21	repeat INI instruction until register B = 0	x1xx1-
IND	16	reads hardware port (C) and writes the result to memory address HL, decrement HL, B if B = 0 → Z = 1 otherwise Z = 0	x*xx1-
INDR	21	repeat IND instruction until register B = 0	x1xx1-
OUT (n), A	11	writes accumulator to hardware port (n)	------
OUT (C), r	12	writes memory addressed by r to hardware port (C)	------
OUTI	16	reads memory addressed by HL and writes the result to hardware port (C), increment HL, decrement B if B = 0 → Z = 1 otherwise Z = 0	x*xx1-
OTIR	21	repeat OUTI instruction until register B = 0	x1xx1-
OUTD	16	reads memory addressed by HL and writes the result to hardware port (C), decrement HL, B if B = 0 → Z = 1 otherwise Z = 0	x*xx1-
OTDR	21	repeat OUTD instruction until register B = 0	x1xx1-

APPENDIX M: INSTALLING A SPEAKER

The PC-G850V (S) has a connector for attaching a speaker. These connections are marked with BZ+ and BZ-. Here the piezo is soldered and attached to the housing with double-sided tape.



Note: The previous models do not have these two connections. In this case, the cables must be connected directly to the 11-pin interface. (Pin 3 (FL3) and pin 7 (FL7).