MF730-05a

EPSON

CMOS 8-BIT SINGLE CHIP MICROCOMPUTER **E0C88 Family**

STRUCTURED ASSEMBLER MANUAL

ENERGY SAVING EPSON

SEIKO EPSON CORPORATION

Preface

This manual explains the software development procedure using the E0C88 Family software development tool package "E0C88 Family Structured Assembler", the functions of the software tools included in the package and how to operate them.
This manual has been divided into the following three parts depending on the explanation contents.

I User's Guide

Part I includes the explanation on program development, installation, flow of program development and the basic processing procedures of each tool within that flow when using the "E0C88 Family Structured Assembler" package.

II Creating Procedure of Assembly Source File

Part II explains details of the parts relative to an assembly source file creation such as the assembly source file format and the pseudo-instructions of which the structured preprocessor sap88 and the cross assembler asm88 are included.

III Reference

Part III describes the start-up format, all the start-up flags that can be used, error messages, etc. as reference for each software tool.

When you develop the E0C88 Family program for the first time, you should read from Part I to understand the development procedure. After understanding basic operation procedure in Part I, you will be able to develop a program by referring to the necessary section in Part II and III.

This manual and the software tools in this package can be used for all E0C88 Family models (compatible with new models in the future). Besides this package, programs for the model specific settings are prepared as a "Development Tool" package for each model. Refer to the manual attached in it.

I USER'S GUIDE

Part I includes the explanation on program development, installation, flow of program development and the basic processing procedures of each tool within that flow when using the "E0C88 Family Structured Assembler" package.

Contents

1 OUTLINE OF PACKAGE

1.1 Introduction

The "E0C88 Family Structured Assembler" package is one of the software development tools of the CMOS 8-bit single chip microcomputer E0C88 Family. It consists of a cross assembler, linker and utilities to create programs.
This package can commonly be used for all E0C88 Family models and allows for development of programs with macro function. Model dependent software tools are prepared for each model as "E0C88XXX Development Tool" package.
Therefore, the software included in this package can directly be used for future E0C88 Family models as well.

1.2 Standard Floppy Disk

This package includes the following floppy disks containing the structured assembler, linker and various common software tools.

1) 3.5" floppy disk (2HD) for IBM-PC/AT .... one
2) E0C88 Family Structured Assembler Manual (this manual) .... one

Before use of the package, read section "2.1 System Configuration" containing information on the personal computers compatible as the host for development systems.
Always make a copy the master disk either on the hard disk or on another floppy disk and store the original in a safe place.

Note: Use the DISKCOPY for backup of the original disk.

Refer to section "2.2 Installation" on how to install the floppy disk and the software development tools.

1.3 Outline of Software Tools

The software tools included in this package is responsible for the area indicated (with shading) in the overall software development program.

Figure 1.3.1 shows the flow of software development using the structured assembler.

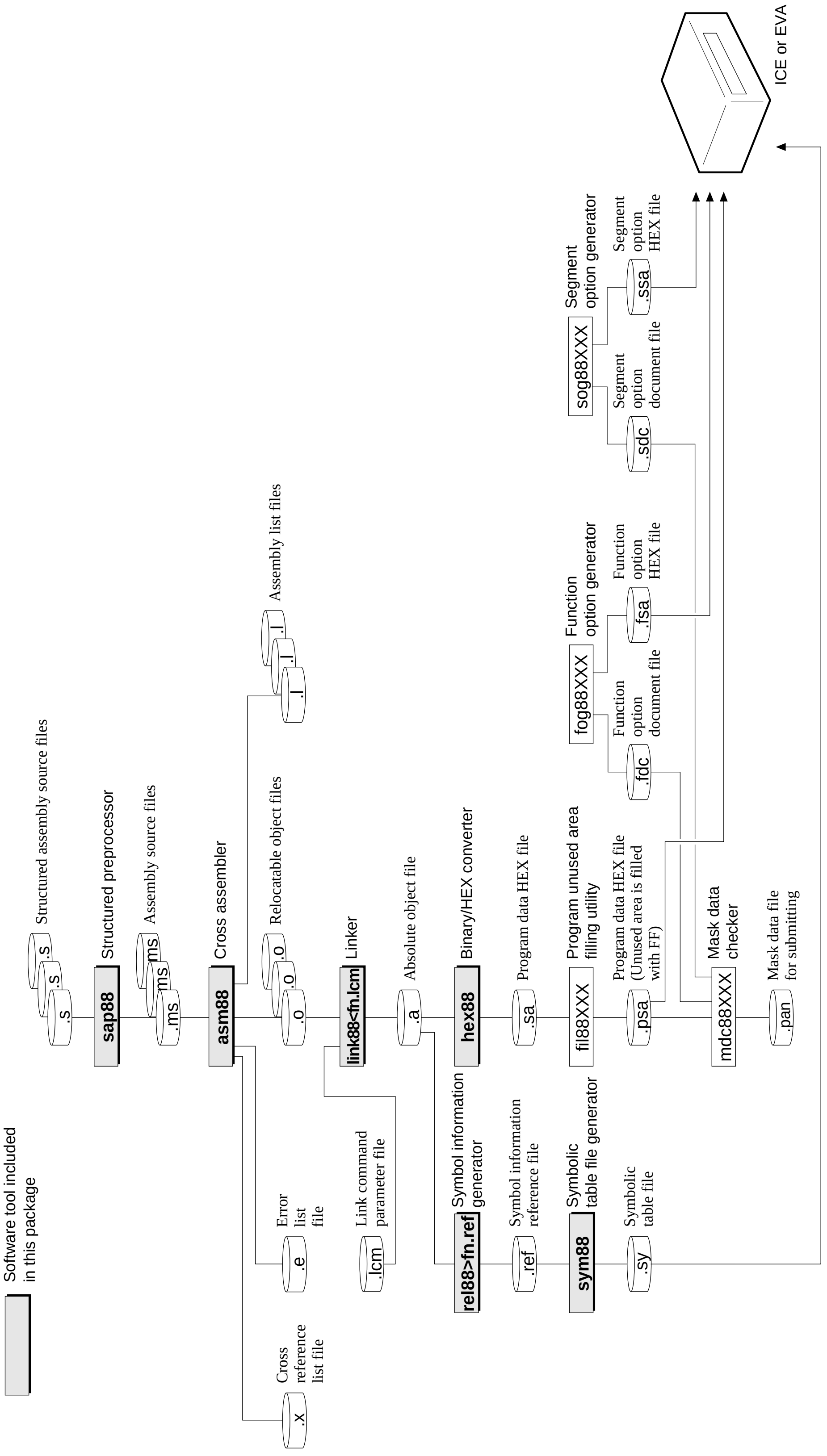


Fig. 1.3.1 Structured assembler software development flow

The basic functions of each program are as follows.

1.3.1 Structured preprocessor <sap88>

The sap88 structure preprocessor is a preprocessor used to add the macro function on the cross assembler asm88.
First create assembly source files including macro functions and process them with the sap88 to create the source files (in which macros are expanded into the E0C88 instructions) that can be assembled with the asm88.

1.3.2 Cross assembler <asm88>

The asm88 cross assembler assembles the program source file described by the E0C88 instruction set and pseudo-instruction and converts it into machine language.
The asm88 is compatible with the relocatable assembly for development by module, and creates relocatable object files used to link other modules via the linker.

1.3.3 Linker <link88>

The relocatable object file created with the asm88 is linked if there is more than one present and then converted into absolute (binary form) object file.

1.3.4 Other utilities

This package contains the following utility programs in addition to the earlier mentioned major programs.

(1) Symbol information generator <rel88>
This is a program that obtains symbolic table information of the relocatable object file.
This utility is used for preprocessing of symbolic table generations.

(2) Binary/HEX converter <hex88>
Converts the binary file into a Motorola S2 format HEX file (ASCII file).
This is basically used to convert the absolute object file output from the link88 linker into a HEX program file. The converted program data HEX file allows for debugging through hardware tools and creation of mask data.

(3) Symbolic table file generator <sym88>
The sym88 symbolic table file generator converts a symbolic information file generated in file redirect with the rel88 symbol information generator to a symbolic table file that can be referenced in the ICE88. Loading the symbolic table file and the corresponding relocatable assembly program file in the ICE88 makes symbolic debugging possible.

1.3.5 Batch files

Batch files are included to automatically process basic tools and operations to promote efficient program development. Customize the file accordingly.

Note: The batch files are installed in the root directory.
Use them after copying to your work directory if necessary.

- ra88.bat: Batch file for relocatable assembly
- lk88.bat: Batch file for linking

Details on the batch file and how to create customized files will be explained in Section 3, respectively under their titles.

2 DEVELOPMENTAL ENVIRONMENT AND INSTALL

Here we will explain the configuration of the program development system and installation procedure for the software tools.

2.1 System Configuration

The system configuration that is necessary for the program development using this package is shown below.

■ Development using IBM-PC/AT

Model:	IBM-PC/AT and the full compatible (*1)
DOS:	PC-DOS ver. 3.3 or higher (*2)
	MS-DOS ver. 3.3 or higher (*2)
RAM capacity:	640K bytes (*3)
Hard disk drive:	Free space more than 510K bytes is necessary (*4)
Floppy disk drive:	3.5 inches
	2HD (1.44M bytes)
	1 drive or more
CRT & graphic board:	EGA, VGA (text display only, fixed at 80 character × 25 lines)
Printer:	For listing (only character printing)
Other software:	Editor For source file creation/modification

Restrictions:

*1 Compatible with models that use Intel 80286/386/486 system for CPU.
When debugging using the ICE88, the interface board is compatible with XT bus (8-bit) but assumes that the CPU is 80286 or higher (IBM-PC/AT).

*2 In the case of MS-DOS 5.0, it is possible to operate unless the task swap function used.
It can be operated by DOS/V in compatible with PC/MS-DOS. (However, since multilingual mode is not supported, be sure to use English mode.)

*3 This system operates in the real mode and does not use EMS and protected memory.

*4 Assumes to use HD basically.

2.2 Installation

Here we will explain the installing of the programs included in this package. The contents of the floppy disk supplied with this package are shown in Figure 2.2.1.
Keep the original disk in a safe place as a backup after installing.

Note: Use the DISKCOPY for backup of the original disk.

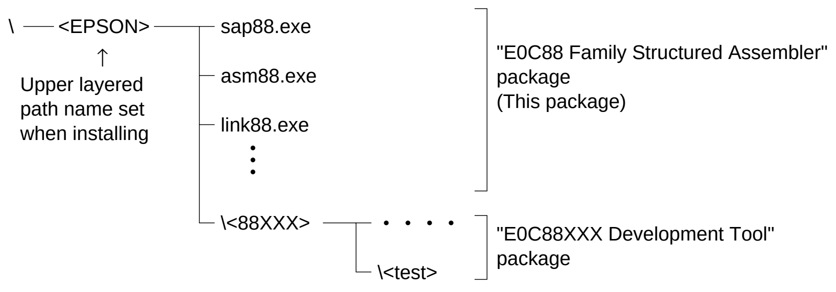
2.2.1 Installer

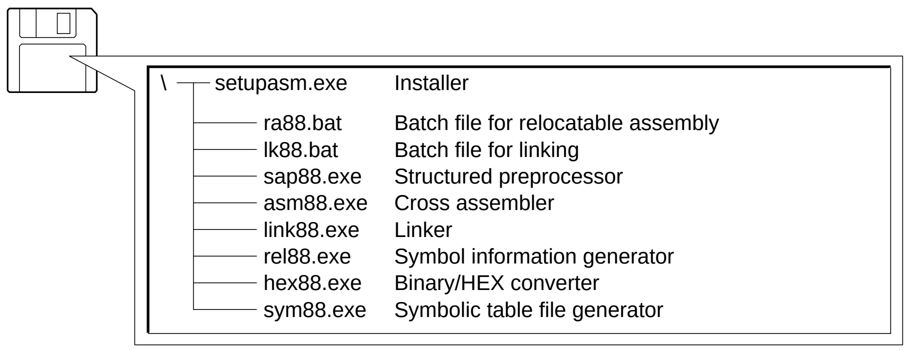
This package includes the exclusive installer "setupasm.exe" to install the software tools on the customer's disk drive. Install the software tools in this package by starting this installer.

Path name to install is fixed at the root directory (\) in the default, but the setting that appending the upper layered path name is possible.
When installing other software tools of the E0C88 Family, by setting the same upper layered path name like this, file management on the disk will be easy.

What's more, added path names will automatically be reflected in the path name to search for batch file commands.

Example: When setting to append "EPSON" in the upper layered path name.


\
<EPSON>
Upper layered path name set when installing
sap88.exe
asm88.exe
link88.exe
\<88XXX>
\<test>
"E0C88 Family Structured Assembler" package (This package)
"E0C88XXX Development Tool" package



```
\ ── setupasm.exe    Installer
     ├── ra88.bat    Batch file for relocatable assembly
     ├── lk88.bat    Batch file for linking
     ├── sap88.exe   Structured preprocessor
     ├── asm88.exe   Cross assembler
     ├── link88.exe  Linker
     ├── rel88.exe   Symbol information generator
     ├── hex88.exe   Binary/HEX converter
     └── sym88.exe   Symbolic table file generator
```

Fig. 2.2.1 Contents of floppy disk

*Note: The batch files are installed in the root directory.
Use them after copying to your work directory if necessary.*

2.2.2 Installation procedure

The software tools in this package need capacity approximate 510K bytes including a work area in case of all the program is installed. Therefore, we recommend to use HD (hard disk) basically.

The exclusive installer in this package checks the free space on the disk to be installed. If there is not enough free space on the install destination disk, reserve free space on the disk to perform file deletion and backup.

■ Operation procedure

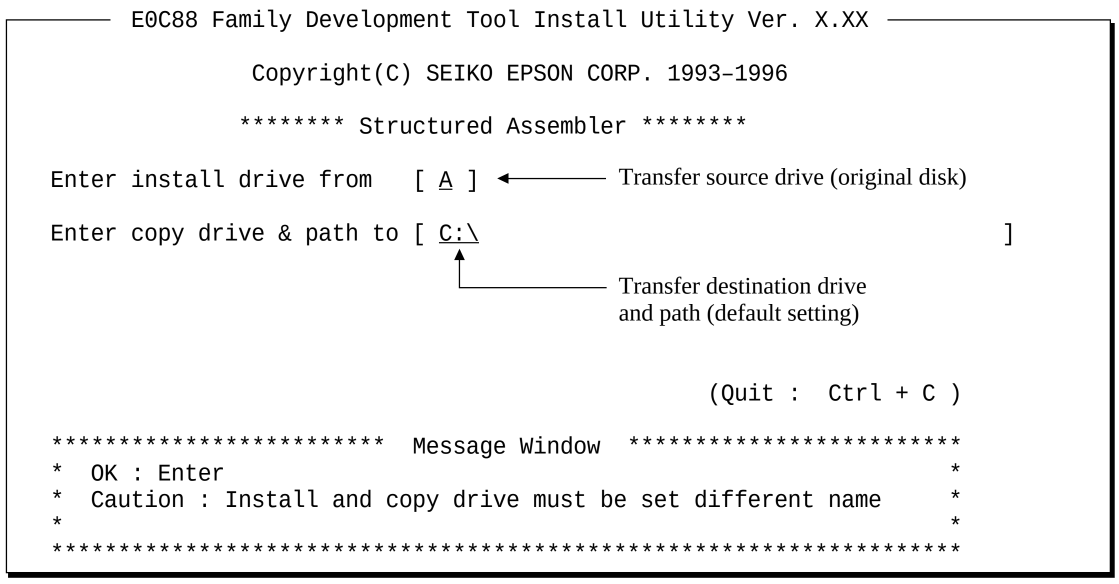
(1) Insert the floppy disk in this package into a drive.

(2) Start the installer.
When using an IBM-PC/AT, be sure to set English mode.

```
C:setupasm↵
```

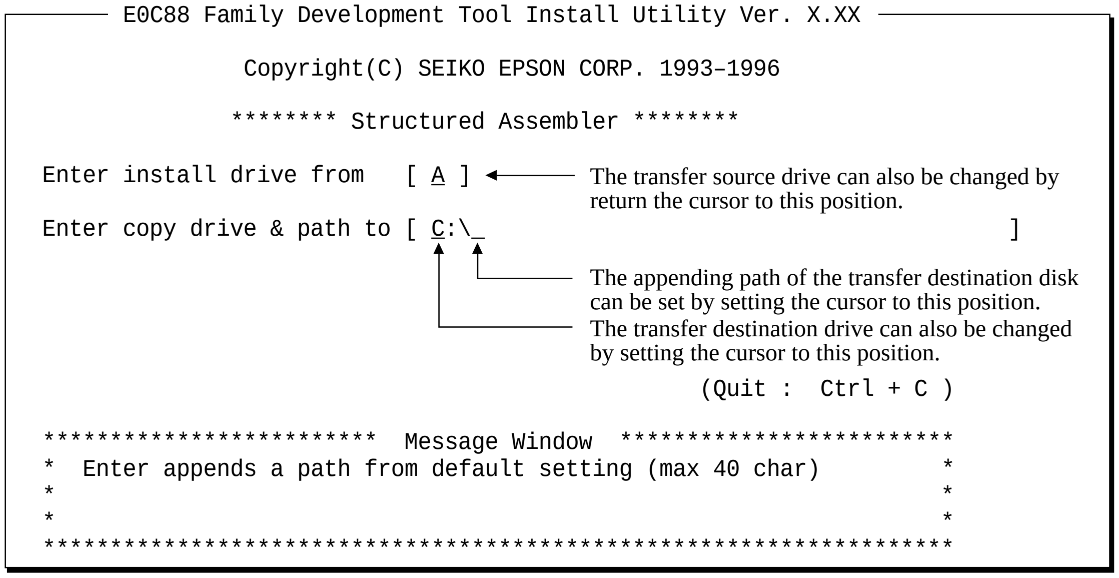
Hereafter, install according to the messages displayed as below.

1. Initial screen


Transfer source drive (original disk)
Transfer destination drive and path (default setting)

```
E0C88 Family Development Tool Install Utility Ver. X.XX

              Copyright(C) SEIKO EPSON CORP. 1993-1996

             ******** Structured Assembler ********

Enter install drive from   [ A ]

Enter copy drive & path to [ C:\                                        ]



                                                 (Quit :  Ctrl + C )

*************************  Message Window  *************************
*  OK : Enter                                                      *
*  Caution : Install and copy drive must be set different name     *
*                                                                  *
********************************************************************
```

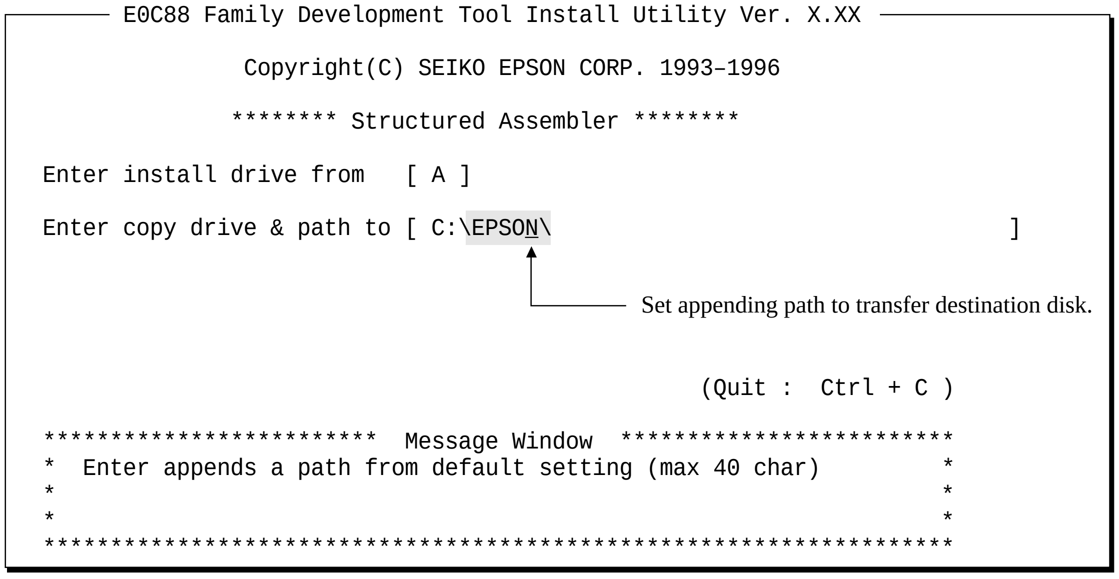
2. Changing transfer destination drive and path



```
           E0C88 Family Development Tool Install Utility Ver. X.XX

                   Copyright(C) SEIKO EPSON CORP. 1993–1996

                  ******** Structured Assembler ********

 Enter install drive from   [ A ]

 Enter copy drive & path to [ C:\_                                    ]

                                                    (Quit :  Ctrl + C )

 ************************  Message Window  ************************
 *  Enter appends a path from default setting (max 40 char)        *
 *                                                                 *
 *                                                                 *
 *******************************************************************
```

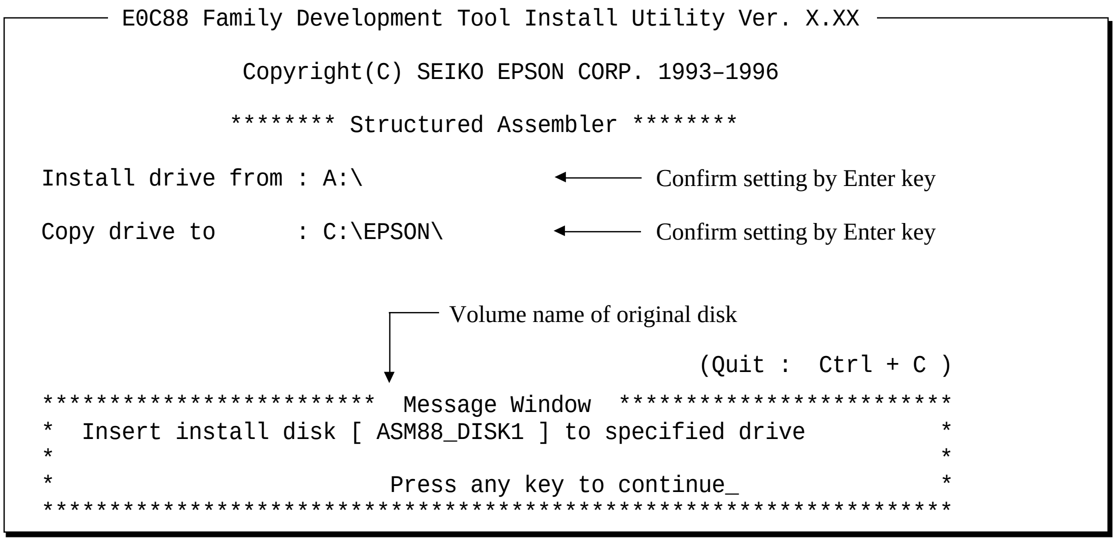
The transfer source drive can also be changed by return the cursor to this position.
The appending path of the transfer destination disk can be set by setting the cursor to this position.
The transfer destination drive can also be changed by setting the cursor to this position.

3. Registration of appending path of transfer destination



```
           E0C88 Family Development Tool Install Utility Ver. X.XX

                   Copyright(C) SEIKO EPSON CORP. 1993–1996

                  ******** Structured Assembler ********

 Enter install drive from   [ A ]

 Enter copy drive & path to [ C:\EPSON\                               ]

                                                    (Quit :  Ctrl + C )

 ************************  Message Window  ************************
 *  Enter appends a path from default setting (max 40 char)        *
 *                                                                 *
 *                                                                 *
 *******************************************************************
```

4. Confirmation of transfer source disk (original) insertion



```
        E0C88 Family Development Tool Install Utility Ver. X.XX

                Copyright(C) SEIKO EPSON CORP. 1993–1996

               ******** Structured Assembler ********

  Install drive from : A:\
  Copy drive to      : C:\EPSON\


                                                     (Quit :  Ctrl + C )
  *************************  Message Window  *************************
  *  Insert install disk [ ASM88_DISK1 ] to specified drive           *
  *                                                                   *
  *                          Press any key to continue_               *
  *********************************************************************
```

Confirm setting by Enter key
Confirm setting by Enter key
Volume name of original disk

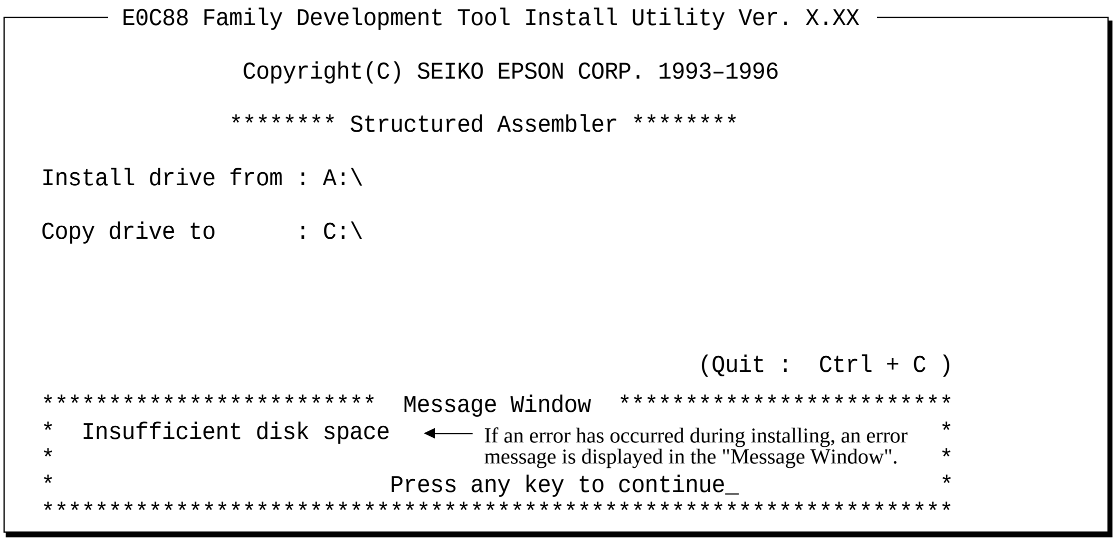
After setting the transfer source drive and transfer destination drive by Enter key respectively, the installer waits for confirmation of the transfer source disk (original) insertion.

⇩

Installation starts by inputting any key in this status.

The file name to be transferred is displayed in the "Message Window" during installing. When the installation is completed normally, an end message is displayed in the "Message Window".

5. Error



```
        E0C88 Family Development Tool Install Utility Ver. X.XX

                Copyright(C) SEIKO EPSON CORP. 1993–1996

               ******** Structured Assembler ********

  Install drive from : A:\

  Copy drive to      : C:\



                                                     (Quit :  Ctrl + C )
  *************************  Message Window  *************************
  *  Insufficient disk space                                          *
  *                                                                   *
  *                          Press any key to continue_               *
  *********************************************************************
```

If an error has occurred during installing, an error message is displayed in the "Message Window".

Note: When an error which is not fatal is generated in the installing stage, the installer returns to the initial screen and the operation can be continued. However, when "Quit" or a fatal error is generated, the installer is forcibly terminated. (In this case, delete all the file which are installed incompletely.)

<Messages of installer>

1. Start-up message

```
E0C88 Family Development Tool Install Utility Ver. X.XX
Copyright (C) SEIKO EPSON CORP. 1993–1996
```

2. Setting, confirmation and operating messages

Message	Explanation
Enter install drive from	Transfer source drive name
Enter copy drive & path to	Transfer destination drive name
Caution : Install and copy drive must be set different name	Transfer source drive name and transfer destination drive name must be set to different names.
Insert install disk [volume name] to specified drive	Insert the transfer source disk [volume name] into the specified drive.
Copy drive and path is [drive and path name]	Transfer destination drive and path is [drive name and path name].
Enter appends a path from default setting (max 40 char)	Input path name appending to the default setting.
Install drive from : [Install drive and path name]	Transfer source drive is [drive name and path name].
Copy drive to : [Copy drive and path name]	Transfer destination drive is [drive name and path name].
Copying [reading file]	Reading transfer source file [file name].
to [writing file]	Writing transfer destination file [file name].
Press any key to continue	Press any key to continue the installer.
Press any key to exit	Press any key to terminate the installer.
E0C88 Family Dev. Tools install utility has been successfully	Installation of the E0C88 Family software tools has been succeeded.
Good-bye from E0C88 Family Dev. Tools install utility	Terminates installation of the E0C88 Family software tools.

3. Error messages

Error message	Explanation
Incorrect DOS Version use DOS 3.X or later	DOS version is incorrect.
Write protect error	Transfer destination disk has been protected from writing.
Unit No. is not exist	There is no unit number.
Drive not ready	The drive is not ready.
Install disk is different	Disk to be installed is different.
Seek error	Seek error has been generated.
Media type is different	Disk is different.
Sector not found	Sector cannot be detected.
Write error	Write error has been generated.
Read error	Read error has been generated.
Other error	Other error has been generated.
Disk error on [drive]	Disk error has been generated on [drive].
Cannot create the path	Path cannot be created.
Setting path already exists enter another drive and path	Since the set path already exists, set another drive and path.
Insufficient disk space	Disk space is insufficient.
Insufficient disk space, insert other disk	Since the disk space is insufficient, insert another disk.
Bad select the install drive	Drive specification for transfer source disk is wrong.
Bad select the copy drive	Drive specification for transfer destination disk is wrong.
Error : [Reading file] Cannot open source file	Transfer source file cannot be read.
Error : [Writing file] Cannot open out file	Transfer destination file cannot be written.
E0C88 Family Dev. Tools install utility has been terminated	Installation of the E0C88 Family software tools has been terminated.
Caution : Delete temporary installed and files	Since the installation of the E0C88 Family software tools has been terminated, delete all the file which are installed incompletely.

3 PROGRAM DEVELOPMENT PROCEDURES

This section will start off by explaining the flow involved in program development and then give details on how each software tool of this package is used, in accordance with the development flow. Each software tools will be explained of its basic processing procedures and the flag settings (start-up command flag) required for the tools in terms of batch file commands. Refer to Part III, "Reference" for more information on other flags, etc.

3.1 Development Flow

The following shows the program development procedure using the asm88 cross assembler.

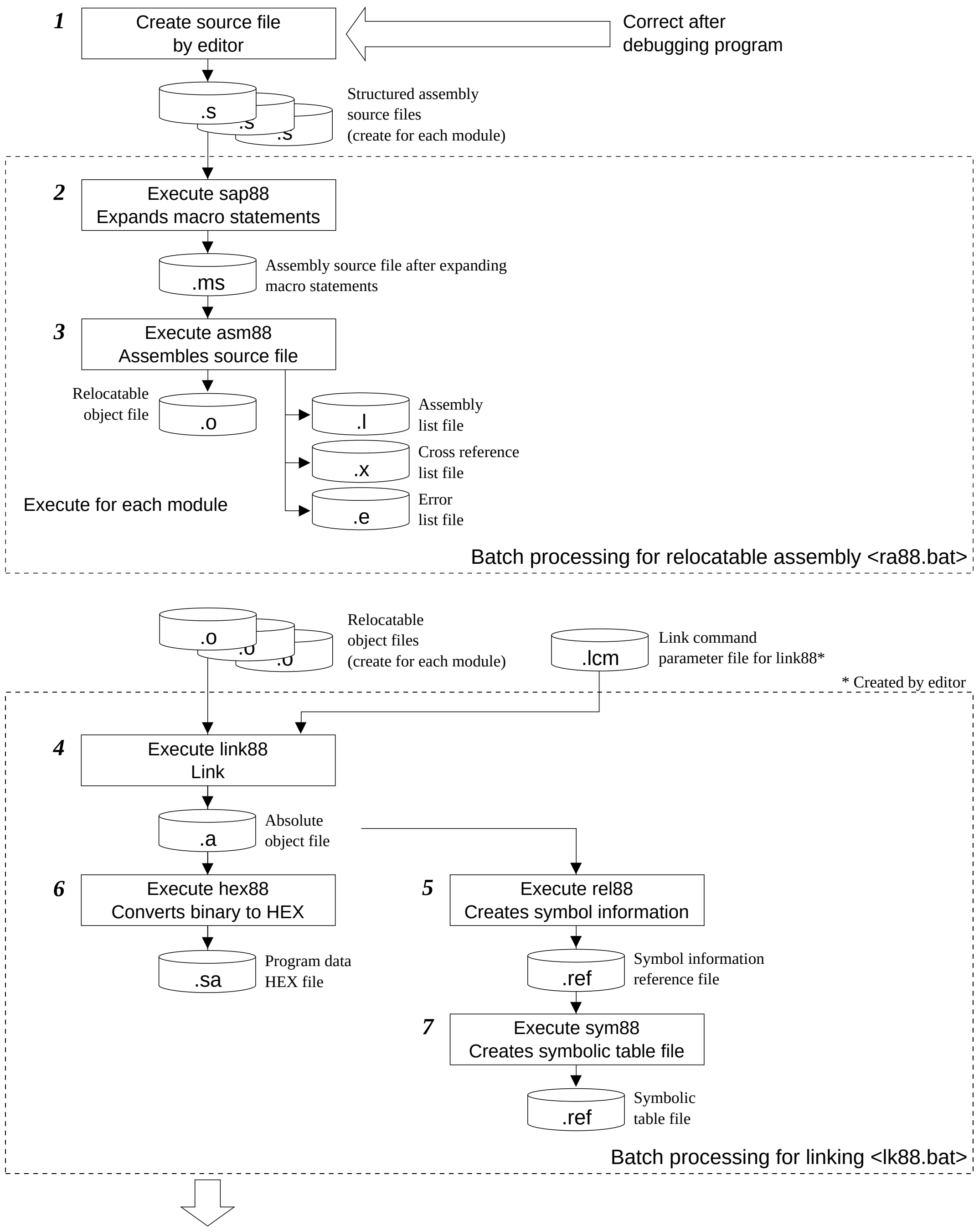
<Relocatable assembly and link>

– Create the entire program as a multiple module (development by module) –

Relocatable assembly refers to the assembling method in which programs are allocated into several parts (each allocated part is referred to as a module) according to the processing contents and then undergoing development procedures by each module.
The cross assembler can input assembly source files created with an editor and the files in which macros are expanded by the sap88.
Each module (relocatable object file) is linked via the linker after assembling and then consolidated into one program. The program memory address that allocates each module is determined through the link. Therefore, the developmental process in which the source program is created can be performed without regards to the address.
Debugging efficiency is boosted since this method allows for debugging by modules that have been allocated in small programs.

Figure 3.1.1 shows the flow of program development upon using the relocatable assembly. This package contains "ra88.bat" and "lk88.bat" that are batch files containing basic processing tools. Customize accordingly. (Refer to section "3.3.4 Batch processing the relocatable assembly" and "3.4.5 Batch processing for linking (lk88.bat)" for more information on "ra88.bat" and "lk88.bat".)

Note: *Prepare each relocatable module under 32K bytes so that they fit in one bank. Modules exceeding this capacity will result in an error message during linking. Thus, it will be necessary to allocate the program so that it is under 32K bytes. Similarly, the data size must be under 64K bytes so that it fits in one page.*
The modules cannot be reallocated so that they span across both banks. In this case, the modules will be allocated so that it starts from the head of the next bank. The program memory (usable area) will be wasted if all modules are too large. Give consideration to each module size to prevent this.


1
Create source file by editor
Correct after debugging program
.s
Structured assembly source files (create for each module)
2
Execute sap88
Expands macro statements
.ms
Assembly source file after expanding macro statements
3
Execute asm88
Assembles source file
Relocatable object file
.o
.l
Assembly list file
.x
Cross reference list file
.e
Error list file
Execute for each module
Batch processing for relocatable assembly <ra88.bat>
.o
Relocatable object files (create for each module)
.lcm
Link command parameter file for link88*
* Created by editor
4
Execute link88
Link
.a
Absolute object file
6
Execute hex88
Converts binary to HEX
.sa
Program data HEX file
5
Execute rel88
Creates symbol information
.ref
Symbol information reference file
7
Execute sym88
Creates symbolic table file
.ref
Symbolic table file
Batch processing for linking <lk88.bat>

- System code setting and FF filling in unused program area by fil88XXX.
- Program debugging using ICE88/EVA88.
- Creating mask data of program.

Fig. 3.1.1 Relocatable assembly development flow

3.2 Creating Source File

***Software used:* Editor**

Create the source file using the editor.
Small applications can be created solely in assembler language with the entire program as a single module.
What's more, source files for single module can also be allocated by using the INCLUDE pseudo-instruction of the sap88 structured preprocessor.
Generally, debugging requires appropriate consideration to module allocation since source files are each created for respective modules.

3.2.1 Development using assembler language

Create source files for assembler modules by using the E0C88 CPU instruction set or assembler pseudo-instructions.
Specify the assembly source file name with a ".s" on the extension.
Each source program statement basically comes in the following form.

Symbol field	Mnemonic field	Operand field	Comment field

- ***Symbol field:***
  This field indicates the symbol. Always put a colon (:) immediately after the symbol, other than for EQU or SET command statements.
- ***Mnemonic field:***
  This field indicates the operation code and pseudo-instruction.
- ***Operand field:***
  This field indicates the operand, constant, variable, defined symbol, symbol that indicates the memory address and formula of each instruction.
- ***Comment field:***
  A semi-colon (;) at the beginning of this field, then continued with a comment.

Refer to Part II of this manual for more information on how to create a source file.
Macro statement offered by the sap88 structured preprocessor and various pseudo-instructions of the asm88 cross assembler can be used for this assembler.
The following indicates an outlines of these statements and instructions.

<Instruction set>

All E0C88 Family models employs a E0C88 in the core CPU. Therefore, instructions are common for all models other than for CPU MODELS and mode limitations. Refer to the "E0C88 Core CPU Manual" for more information on the instructions, and refer to the "E0C88XXX Technical Manual" for control program examples of the peripheral circuit incorporated in each model.
The asm88 cross assembler is capable of converting all mnemonic instruction settings of the E0C88 into machine language.

<Macro statement>

Macro is used to priorly define a processing (sequence of instructions) frequently used in the program with a voluntary name to allow for it to be called out under that specific name. As a result, the need for routine procedures can be eliminated. (For more information refer to Part II of this manual.)
Macro statements are offered as pseudo-instructions of the sap88 and by putting it through the sap88 it is applied in the macro call-out portion in mnemonic form that can be assembled.

Example of macro definition

Before expanding

```
      subtitle          "example"
      public            main,work
      external          src_address,dst_address,counter
;
abc   equ               0ffh
;
      data
work: db                [1]
;
      code
;*************************************************
;**     * macro define *                        **
;*************************************************
nop3  macro
      nop
      nop
      nop
      endm
;*************************************************
;**     * example *                             **
;*************************************************
main:
      ld    a,#abc
      lb    b,[work]
      nop3                         ; macro call ***
      ld    ix,#src_address
      ld    iy,#dst_address
      ld    hl,[counter]
;***
      end
```

Macro definition

Macro call

After expanding

```
      subtitle          "example"
      public            main,work
      external          src_address,dst_address,counter
;
abc   equ               0ffh
;
      data
work: db                [1]
;
      code
;*************************************************
;**     * macro define *                        **
;*************************************************
;*************************************************
;**     * example *                             **
;*************************************************
main:
      ld    a,#abc
      lb    b,[work]
      nop
      nop
      nop
      ld    ix,#src_address
      ld    iy,#dst_address
      ld    hl,[counter]
;***
      end
```

Macro statement expanded into mnemonics

<Pseudo-instruction>

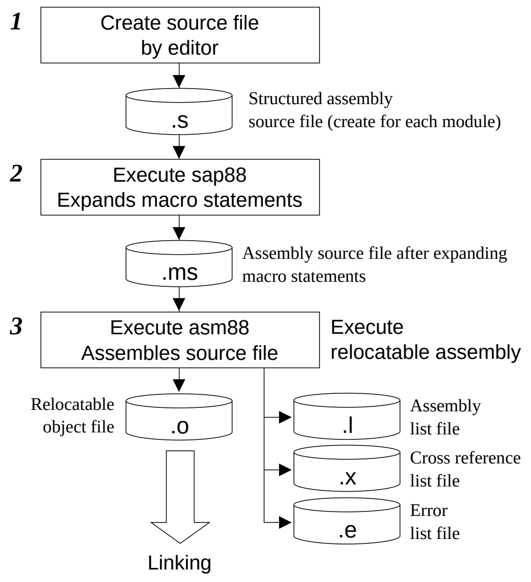
Pseudo-instruction by function	Description
Section setting pseudo-instructions (BANK, DATA)	Use to specify sections. * Specifies the program area and data area. (For more details refer to "3.3.2 Cross assembler (asm88)".)
Data definition pseudo-instructions (DB, DW, DL, ASCII, PARITY)	Specifies various data within the program memory.
Symbol definition pseudo-instructions (EQU, SET)	Allocates constant to symbols (voluntary name) used within the source program.
Location counter control pseudo-instruction (ORG)	Sets the program counter.
External definition and reference pseudo-instructions (EXTERNAL, PUBLIC)	Allows for symbols and labels to be referenced between modules.
Source file insertion pseudo-instruction (INCLUDE) sap88 only	Inserts contents of other source files in voluntary places.
Assembly termination pseudo-instruction (END)	Specified the assembly end point.
Macro related pseudo-instructions (MACRO–ENDM, DEFINE, LOCAL, PURGE, UNDEF, IRP–ENDR, IRPC–ENDR, REPT–ENDR) sap88 only	Defines the macro statement.
Conditional assembly pseudo-instructions (IFC–ENDIF, IFDEF–ENDIF, IFNDEF–ENDIF) sap88 only	Assembly or skip can be set according to the definition of the symbol.
Output list control pseudo-instructions (LINENO, SUBTITLE, SKIP, NOSKIP, LIST, NOLIST, EJECT)	Controls the output to the assembly list file.

Unlike CPU instructions, pseudo-instructions do not directly compose of application programs upon executing control instructions to the asm88.
The pseudo-instructions that can be used with this assembler are indicated above according to their functions. (Refer to Part II of this manual for more details.)

3.3 Assembly

This section will explain the method to assemble the assembly source file and the relocatable object file created by the process.

Software used: sap88, asm88



1
Create source file by editor
.s
Structured assembly source file (create for each module)
2
Execute sap88 Expands macro statements
.ms
Assembly source file after expanding macro statements
3
Execute asm88 Assembles source file
Execute relocatable assembly
Relocatable object file
.o
.l
Assembly list file
.x
Cross reference list file
.e
Error list file
Linking

Fig. 3.3.1 Flowchart of relocatable assembly

3.3.1 Structured preprocessor (sap88)

This assembler system is composed of the sap88 structured preprocessor and asm88 cross assembler.

As indicated in section 3.2.1, the sap88 is responsible in putting the macro statement in mnemonic form.
Since the asm88 cannot read the macro statement, assembly source files included these documents can not be directly input in the asm88 as a file.
The asm88 is the actual assembler responsible in converting the mnemonic language into machine language and assembling cannot be performed with sap88.
Therefore, there is a need to used both sap88 and asm88 for the structured assembly. It is advisable to process it through the sap88 even if the structured assembly is not required, since the process will not effect the source file.

The sap88 inputs an assembly source file with a ".s" extension and expands the macro statements. After that, the sap88 outputs a file for assembly. The name of the extension of the output file should be set as ".ms".

3.3.2 Cross assembler (asm88)

The asm88 cross assembler assemble the E0C88 Family CPU instructions and the pseudo-instructions of the asm88 and converts it into machine language.
The asm88 is compatible with the relocatable assembly.
The relocatable assembly creates relocatable object files (".o") that will be linked with other modules using a linker. The asm88 can input several assembly source files and thus allows for simultaneously assembly of several relocatable modules.
The asm88 can also output three lists, i.e., assembly list (".l"), error list (".e") and a cross reference list (".x") for the programmer.
The assembly list consists of the line number, target address, code that corresponds to the source and source statements. The line number is output in decimals, while the address and code are output in hexadecimals.
If in case an error takes place during assembling, an error list file containing the source file name, line number in which the error took place, error level and error message will be created. What's more, the assembly list file will also note the line in which the error took place with an asterisks "*" beside the line number. Processing will be continued regardless of an error message unless the error is fatal.
The relation of the symbol definition and reference within the file has been prepared to foster easy understanding depending on the cross reference list.
File management has been enhanced since they are prepared as separate files.

<Control of program and data memory>

This section will explain how to control the memory of the program and data.
The E0C88XXX memory map can be categorized in the program memory (ROM) for the program code and RAM and I/O memory for the data.
For example, even if a certain symbol is noted in a voluntary position in the assembly source file, the asm88 is not capable of determining whether this is within the program memory or data memory.
For this reason, there is a need to clarify which memory each line comes under by prior instruction through the section setting pseudo-instructions.
The following explains the section set methods for the relocatable assembly, and the asm88 process corresponding to the method.

Setting sections

The absolute address allocated within each module of the relocatable assembly will be specified or determined upon liking. Therefore, an absolute address cannot be specified within the assembly source file. A relative address specification can be made using an ORG pseudo-instruction, however, in this case, a standard for a relative address will be required. What's more, there is also a need to specify the segments of the program and data area for the asm88.

The entire program for this assembler is categorized into CODE and DATA. These basically indicate the following areas.

CODE section: Program data area written in the ROM
DATA section: Data memory area other than ROM

The asm88 is complete with a CODE and DATA pseudo-instruction to specify the section. The area can be set through descriptions in the assembly source file.

■ Specifying the CODE section

If a CODE pseudo-instruction is described within an assembly source file, the asm88 will assemble it to be allocated to the CODE section until the next DATA pseudo-instruction appears. The CODE pseudo-instruction can be used in several places within one module. The asm88 assumes the head of the CODE section within the module as relative address 0000H and will continuously realign them in the order that the CODE pseudo-instruction appears to consolidate it into one block. In other words, a CODE specification range of one module will be handled as one CODE section. (Refer to Figure 3.3.2.1.)

The CODE section of each module is further consolidated as a whole by the linker. The linker will link in sectional units in accordance with the bank control within the program memory area.

The CODE section consists of CODE sections with one or multiple modules and the maximum size is limited to 32K bytes as one bank is. (Details on section control will be explained in "3.4.2 Section control".) Therefore, the programmer must be careful not to use more than 32K bytes in the code when creating a module. The capacity of the CODE section can be verified by using the -ROM# flag when starting-up the asm88. Use of this feature is advised.

For example, when flag specification for "-ROM 32768" is performed, an error message will be displayed if a CODE section of one module exceeds 32K bytes.

■ Specifying the DATA section

If a DATA pseudo-instruction is described within an assembly source file, the asm88 will assemble it to be allocated to the DATA section until the next CODE pseudo-instruction appears. The DATA pseudo-instruction can be used in several places within one module. The asm88 assumes the head of the DATA section within the module as relative address 0000H and will continuously realign them in the order that the DATA pseudo-instruction appears to consolidate it into one block. In other words, a DATA specification range of one module will be handled as one DATA section. (Refer to Figure 3.3.2.1.)

The DATA section of each module is further consolidated as a whole by the linker. The linker will link in sectional units in accordance with the page control within the data memory area.

The DATA section consists of DATA sections with one or multiple modules and the maximum size is limited to 64K bytes as one page is. (Details on section control will be explained in "3.4.2 Section control".) Therefore, the programmer must be careful not to use more than 64K bytes in the code when creating a module. The capacity of the DATA section can be verified by using the -RAM# flag when starting-up the asm88. Use of this feature is advised.

For example, when flag specification for "-RAM 65535" is performed, an error message will be displayed if a DATA section of one module exceeds 64K bytes.

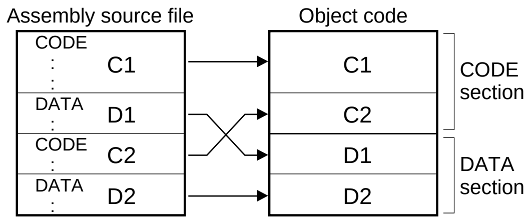


Fig. 3.3.2.1 CODE section and DATA section

■ Note

If either the CODE pseudo-instruction or DATA pseudo-instruction is missing during relocatable assembling the operation will result in an error. For this reason, it is important that the CODE pseudo-instruction is used for the program memory and the DATA pseudo-instruction is used for the data memory.

3.3.3 Starting sap88 and asm88

<sap88 operation procedure>

(1) Set the directory in which the structured assembly source file (.s) is presented as the current drive.

(2) Start-up the sap88 with the next format.

sap88_[flag]_input file↵

_ indicates a space key input.
↵ indicates a return key input.

The following indicates the flag used for batch processing of relocatable assembly (ra88.bat).

Flag	Description
-o <file name>	Specify the file name that is output. (Specify ".ms" as the extension of the file to be output.) If this flag is omitted it will be processed as a standard output.

Refer to Part III of this manual for information on other flags.

Example:
```
C:\USER>c:\EPSON\sap88 -o
                 sample.ms sample.s↵
```

Inputs the assembly source file "sample.s" created in the sub-directory USER of drive C and then creates assembly source file "sample.ms" to be input in asm88 in the same directory as the input file.
If the PATH to sap88 is set, then there is not need to specify the path before sap88.

Refer to section "3.3.9 Example of assembly execution" for more information on I/O files and messages displayed.

<asm88 operation procedure>

(1) Set the directory in which the assembly source file (.ms) created with the sap88 exsists as the current drive.

(2) Start-up the asm88 with the next format.

asm88_[flag]_input file↵

_ indicates a space key input.
↵ indicates a return key input.
Flag can be omitted.

The following indicates the flags used for batch processing of relocatable assembly (ra88.bat).

Flag	Description
-ROM#	Specify the ROM capacity in byte units. It is especially useful during relocatable assembling and is used to verify the size of the CODE area.
-RAM#	Specify the RAM capacity in byte units. It is especially useful during relocatable assembling and is used to verify the size of the DATA area.

Refer to Part III of this manual for more information on other flags.

Example 1: When continuously assembling several assembly source files through relocatable assembly.
```
C:\USER>c:\EPSON\asm88
            sample1.ms sample2.ms↵
```

Inputs the assembly source files "sample1.ms" and "sample2.ms" created in the sub-directory USER of drive C and starts the relocatable assembly process. Then creates the relocatable object files "sample1.o" and "sample2.o" in the same directory as the input file.
At the same time, the assembly list files "sample1.l" and "sample2.l", cross reference list files "sample1.x" and "sample2.x", and error list files "sample1.e" and "sample2.e" will also be created in the same directory.
If the PATH to asm88 is set, then there is not need to specify the path before asm88.

Example 2: Assembling with the relocatable assembler, including the verification of the ROM and RAM capacity.
```
C:\USER>c:\EPSON\asm88 -ROM 32768
            -RAM 65536 sample.ms↵
```

Inputs assembly source file "sample.ms" created within the sub-directory USER of drive C and starts relocatable assembly. Then creates the relocatable object file "sample.o" in the same directory as the input file.
At the same time, creates the assembly list file "sample.l", cross reference list file "sample.x" and error list file "sample.e" in the same directory.
The capacity of the CODE and DATA sections will be verified during assembling with the -ROM and -RAM flags. An error will result in this case when the CODE exceeds 32K bytes and the DATA exceeds 64K bytes.
If the PATH to asm88 is set, then there is not need to specify the path before asm88.

Refer to section "3.3.9 Example of assembly execution" for more information on I/O files and messages displayed.

3.3.4 Batch processing for relocatable assembly (ra88.bat)

The start-up procedures for sap88 and asm88 were already discussed in the earlier section, however, it must be further noted that these can be batch processed by consolidating them into a batch file. The batch file can voluntarily created by the user, however, since this package contains batch file, i.e., ra88.bat for relocatable assembly, the following will introduce the contents of the batch file and how to use them.
This batch file can be used for general processing purposes. Use it advantageously by customizing the flag settings, etc. as needed.

Figure 3.3.4.1 shows the ra88.bat processing flow.

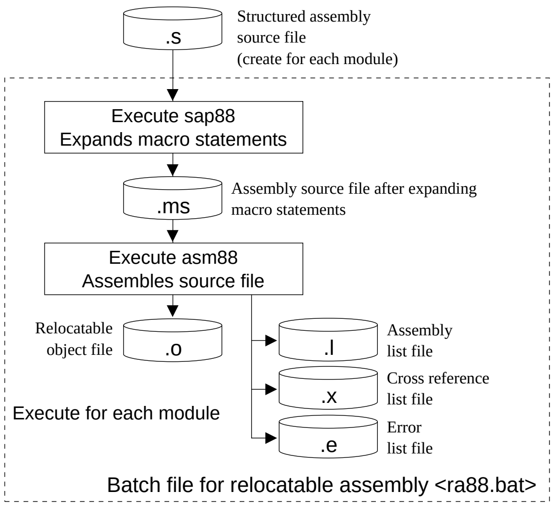

.s
Structured assembly source file (create for each module)
Execute sap88 Expands macro statements
.ms
Assembly source file after expanding macro statements
Execute asm88 Assembles source file
Relocatable object file
.o
.l
Assembly list file
.x
Cross reference list file
.e
Error list file
Execute for each module
Batch file for relocatable assembly <ra88.bat>

Fig. 3.3.4.1 ra88.bat processing flow

<Outline of process>

The ra88.bat inputs the specified assembly source file and then executes sap88 and asm88, respectively to perform relocatable assembly to create a relocatable object file. Since the sap88 does not permit input of multiple assembly source files, it is limited to assembly per module other than when several structured assembly source files are read with the INCLUDE pseudo-instruction of the sap88.

<Input/output files>

The following indicates the input/output files of the ra88.bat.

■ **Input file**

Structured assembly source file (relocatable): file_name.s
This is a structured assembly source file (relocatable) created with an editor such as EDLIN.

■ **Output files**

1. **Assembly source file: file_name.ms**
   An assembly source file in which macros are expanded will be output.
2. **Relocatable object file: file_name.o**
   This is a binary file that has been converted in machine language that can be reallocated through relocatable assembly.
   (This is also the file that inputs the lk88.bat batch file to perform linking.)
3. **Assembly list file: file_name.l**
   This is the file output as a list that corresponds to each source statement when the machine language and the relocatable address (the head of the CODE or the DATA section is assumed as relative address 000000H) converted with the assembler.
4. **Cross reference list file: file_name.x**
   This is the address list that contains the definition and references of symbols.
5. **Error list file: file_name.e**
   This is the list of error taking place during assembling.

<Operation procedure>

(1) Set the directory in which the structured assembly source file (.s) is presented as the current drive.

(2) Start-up the ra88.bat with the next format.

`ra88`_file name↵

_ indicates a space key input.
↵ indicates a return key input.

Do not input the extensions of file name. It is fixed on the ".s" extension.

Example: `C:\USER>c:\EPSON\ra88 sample`↵

Inputs structured assembly source file "sample.s" created within the sub-directory USER of drive C and starts relocatable assembly. Then creates the following files in the same directory as the input file.

```
sample.ms, sample.o, sample.l,
sample.x, sample.e
```

If the PATH to ra88 is set, then there is not need to specify the path before ra88.

Refer to section "3.3.9 Example of assembly execution" for more information on I/O files and messages displayed.

■ Customizing ra88.bat

<***Customizing ra88.bat execution parameters***>

Since the ra88.bat controls the program execution, it has a execution parameter customization field within it. General parameters are temporarily described in the default position, however, it is advised that the program is customized in accordance with the user's development method.

1. **Setting the ROM capacity (Verification of the size of the CODE section)**
   set rom = 32768 : The capacity of the ROM of the CODE section that locates errors will be specified in bytes. (default capacity 32768 = 32K bytes)

2. **Setting the RAM capacity (Verification of the size of the DATA section)**
   set ram = 65536 : The capacity of the RAM of the DATA section that locates errors will be specified in bytes. (default capacity 65536 = 64K bytes)

Note: There are basically no error checks made on these parameter settings, therefore, do not set the parameter with settings other than those specified.

<***Customizing ra88.bat execution command***>

The ra88.bat has the following command line upon execution of the program. Customize these command lines if a flag without a default setting is to be used.

sap88
```
%drv%sap88 -o %1.ms %1.s
```

asm88
```
%drv%asm88 -ROM %rom% -RAM %ram%
%1.ms
```

The %drv% is a path that locates the execution command of the ra88.bat. For this reason, it can not be altered and neither can the SET statement that is defined be altered. The %1 is a file name that is input from the command line.

The following indicates the ra88.bat program source list and the message list of the ra88.bat. Refer to it upon customizing the program.

■ ra88.bat program source list

```
echo off
rem ************************************************************************
rem * E0C88 Family Auto Relocatable Assemble Execution Utility
rem *                              (Ver. X.XX)
rem *                        Copyright(C) SEIKO EPSON CORP. 1993–1996
rem ************************************************************************
rem * customized parameter information
rem * rom=*  * : rom capacity(32768 max.)
rem * ram=*  * : ram capacity(65536 max.)
rem ************************************************************************
rem ********** customized parameter area (default) **********
rem *  caution : customized parameters value do not check, therefore
rem *            please be carefully when you set
rem **********
set rom=32768                      ← Setting the capacity of the ROM
set ram=65536                      ← Setting the capacity of the RAM
```

User customization field

Note: There are basically no error checks made on these parameter settings, therefore, do not set the parameter with settings other than those specified.

```
rem ********** command searching path **********
rem set drv=c:\
```

The drv is a path that locates the execution command of the ra88.bat. It is set to root directory by default. Customize it if necessary.

```
rem ************************************************************************
rem * main program
rem *      if you want to use another option(s), please append
rem *      option flag(s) at command line.
rem ************************************************************************
:start
     echo E0C88 Family Auto Relocatable Assemble Execution Utility Ver. X.XX
     echo Copyright (C) SEIKO EPSON CORP. 1993–1996

           if "%1"=="" goto usage
:error_chk
           if not exist %drv%nul goto exit04
           if not exist %1.s goto exit05
           if not exist %drv%sap88.exe goto exit06
           if not exist %drv%asm88.exe goto exit07

rem (sap88)
:sap88
%drv%sap88 -o %1.ms %1.s                                ← Start-up command of sap88
           if errorlevel 1 goto exit01

rem (asm88)
:asm88
%drv%asm88 -ROM %rom% -RAM %ram% %1.ms                  ← Start-up command of asm88
           if errorlevel 1 goto exit02
                 goto end

     :usage
     echo usage : ra88 needs [input file_name]
                 goto skip
     :exit01
     echo Error stop at %drv%sap88.exe
                 goto skip
     :exit02
     echo Error stop at %drv%asm88.exe
                 goto skip
     :exit03
     echo Cannot find %drv% installed E0C88 dev. tools directory
                 goto skip
     :exit04
```

```
    echo Cannot find input file
                goto skip
    :exit05
    echo Cannot find %drv%sap88.exe
                goto skip
    :exit06
    echo Cannot find %drv%asm88.exe
                goto skip
    :end
    echo ra88.bat utility has been successfully executed.
    :skip
    set rom=
    set ram=
    set drv=
```

■ Message list

1. Start-up message

```
E0C88 Family Auto Relocatable Assemble Execution Utility Ver. X.XX
Copyright (C) SEIKO EPSON CORP. 1993–1996
```

2. Message when terminated normally

```
ra88.bat utility has been successfully executed.
```

3. Error message

Error message	Explanation
usage : ra88 needs [input file_name]	Usage output.
Error stop at [drive and path name] sap88.exe	Error occurred in sap88.
Error stop at [drive and path name] asm88.exe	Error occurred in asm88.
Cannot find [drive and path name] installed E0C88 dev. tools directory	Cannot find [drive or path] in which the E0C88 Family software tools is installed.
Cannot find input file	Cannot find aa88.bat input file (.s).
Cannot find [drive and path name] sap88.exe	Cannot find sap88.
Cannot find [drive and path name] asm88.exe	Cannot find asm88.

Note: The following operations will be stopped when an error occurs.

<Precautions upon using the batch file>

(1) Some of the messages displayed during batch processing is automatically generated through the MS-DOS/PC-DOS batch processing function and command. For this reason, it may be placed under MS-DOS/PC-DOS control when an error occurs and thus force the batch processing to be interrupted.

(2) When an error occurs, the following procedures do not automatically continue. However, it may not be controllable as noted in reason (1) indicated above.

(3) The ra88.bat and the lk88.bat (mentioned hereafter) employ the MS-DOS/PC-DOS COPY command in addition to E0C88 Family tools. For this reason, it is requested that the COPY command is operable, by setting the PATH, when executing the batch file.

(4) The execution parameters (user customization field) of the batch file basically do not locate parameter setting errors. Therefore, do not set the parameters other than specified.

(5) An MS-DOS/PC-DOS environment variable will be used to execute the batch file, therefore, the size of the environment variable should be allocated with as much space as possible using the CONFIG.SYS.

3.3.5 Relocatable object file

The relocatable object file is a binary file that is created through the relocatable assembly of the asm88.
Other than when -o flag is specified the file name that is created will be the same file name input with the asm88 and the extension will be ".o".
This file consists of header information and symbol tables required for reallocation using the linker, in addition to the object (machine language) code.

3.3.6 Assembly list file

The assembly list file is an ASCII file added with an object code (hexadecimal) and code address (hexadecimal) in the assembly source file input in the asm88. It is created through asm88 assembly. Each page will have a header with the file name and date that the file is created.

The file name that is created will be the same as the file name input via the asm88 other than when -o flag is specified. The extension will be ".l".

The assembly list file consists of the following items:

LINE	The consecutive line number from the beginning.
ADDRESS	This refers to the target address of the object code.
CODE:	This is the object (machine language) code that corresponds to the source statement in the same line.
SOURCE STATEMENT ..	This is the assembly source input in the asm88.

When relocatable assembly is performed, the code address will be a relative address from the beginning of the CODE section. Similarly, the address of the data area is a relative address from the beginning of the DATA section.
If an error is occurred, an asterisks "*" will be placed at the beginning of the line in which the error occurred.

The output of assembly list file can be controlled with the following asm88 pseudo-instructions and flag specifications upon start-up.

■ Output list control pseudo-instructions

Pseudo-instruction	Description
LINENO	Changes the line number (LINE) to the voluntary value.
SUBTITLE	Inserts the subtitle line that is voluntarily set after the column explanation line.
SKIP	If any line of the code exceeds 5 bytes through ASCII, DB or DW data settings, the exceeding portion will not be output. (default setting.)
NOSKIP	Outputs all codes by canceling the SKIP setting.
LIST	The following lines are output in a list when the NOLIST setting is canceled.
NOLIST	Prevents output of the list from the line after the pseudo-instruction.
EIECT	Adds a involuntary page break.

Refer to Part II of this manual for details of the pseudo-instructions.

■ Start-up flag

Flag	Description
`-l`	Prevents creation of an assembly list file.

Refer to Part III of this manual for for details of the flag.

3.3.7 Cross reference list

The cross reference list file is created through asm88 assembly with an ASCII file. This ASCII file is defined within the module or contains a list of reference symbols.
The name of the file created will be the same as the file name input with the asm88 other than when specifying -o flag. The extension will be ".x".

The output format of the cross reference list file is as follows.

R SYMBOL A VALUE LINE No. INFORMATION

R	Reference definition G: Global L: Local
SYMBOL	Symbol name (maximum 15 characters)
A	Attribute L: Label C: Constant V: Variable U: Undefined within the module
VALUE	Symbol value (6 digit, hexadecimal expression)

LINE No. INFORMATION

This is a list in which the symbol is defined or referenced line numbers. They are output as follows.
lineno* lineno lineno . . . . lineno

lineno*:	The line number in which the target symbol is defined.
lineno:	The line number in which the target symbol is referenced.

The LINE No. INFORMATION can consist up to a maximum of 12 line numbers.

The following page header will be output at the head of each page.

The numeric labels are temporary labels. The same name can be used if they are outside the range defined by the general label. It will not be output on the cross reference list. (Refer to Part II of this manual for the numeric labels.)

The cross reference list file can prohibit output using the -x flag of the asm88.

Example of cross reference list

```
CROSS REFERENCE TABLE OF asm88  error.x  1993-06-07  17:28    PAGE  1

L  delay           L  000100H       5*      14      15
L  delay_00        L  000103H       7*       9
L  delay_3times    L  000107H      13*
```

3.3.8 Error list

The errors generated during asm88 assembling will be output as an error list file.
The name of the file created will be the same as the file name input with the asm88 other than when specifying -o flag. The extension will be ".e".

The output format of the error list is as indicated below.

SOURCE FILE LINE No.: ERROR LEVEL: ERROR MESSAGE

SOURCE FILE	Source file name
LINE No.	Line number in which the error occurred
ERROR LEVEL	Level of error
Warning	This is a warning and does not affect the output object.
Severe	This is a general error. The output object will be invalid.
Fatal	This is a fatal error. Assembly will be interrupted. Fatal errors are displayed on the CRT without output of an error list file.
ERROR MESSAGE	Error content

Refer to Part III of this manual for the error messages of the asm88.

Example of error list

```
error.s 16:  Severe:   ddelay not defined
```

When an error is not generated, nothing will be output in the error list file.

3.3.9 Example of assembly execution

The following shows example of the assembly execution.

■ Messages when ra88.bat (relocatable assembly) is executed

```
C:\USER>c:\EPSON\ra88 sample↵

C:\USER>echo off
E0C88 Family Auto Relocatable Assemble Execution Utility Ver. X.XX
Copyright (C) SEIKO EPSON CORP. 1993–1996
sap88 Structured Assembler Preprocessor Version X.XX

Copyright (c) 1993 by Advanced Data Controls, Corp.
Licenced to SEIKO EPSON CORP.
asm88 Cross Assembler Version X.XX

Copyright (c) 1993 by Advanced Data Controls, Corp.
Licenced to SEIKO EPSON CORP.

     9 Symbol(s) Used

     0 Warning Error(s)
     0 Severe Error(s)
ra88.bat utility has been successfully executed.
C:\USER>
```

3.4 Link

This section will explain the linking operations of relocatable modules.

Software used: link88

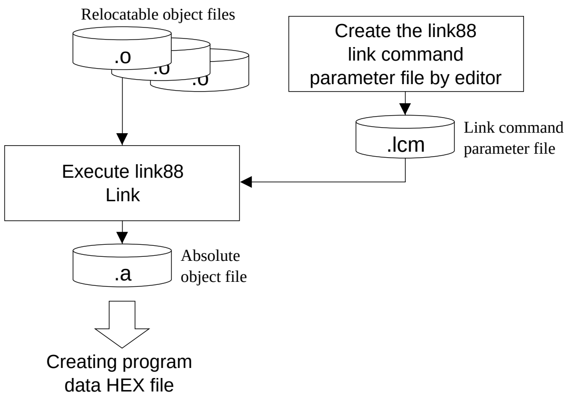

Relocatable object files
.o
.o
.o
Create the link88 link command parameter file by editor
.lcm
Link command parameter file
Execute link88 Link
.a
Absolute object file
Creating program data HEX file

Fig. 3.4.1 Link processing flow

3.4.1 Linking modules

The object codes of each module created with the relocatable assembly of the asm88 is not specified to be located in a certain portion of the ROM. The allocation address is determined by how each modules are linked. The link88 linker is the tool used for linking operations.

When linking is successfully performed the relative address for the external reference label that was undeclared up to this point will be declared and thus, create an absolute object file (.a) that consolidates all modules into one file. By processing this absolute object file with the binary/HEX converter hex88, as indicated in section 3.5, the program data HEX file to be used to create the program mask data or to debug the hardware will be created.

3.4.2 Section control

The E0C88 Family has a 24-bit width address space (maximum of 16M bytes). By using the topmost 8-bit for register control using the code bank register (CB), expand page register (EP, XP, YP) and others, the address space can be allocated into a 32K-byte bank (CODE) or 64K-byte page (DATA) unit. Access performance can be improved within those ranges. By rewriting the content of the register, the user will have access of a voluntary bank or page from a voluntary bank. As a result, large programs and data bases can easily be controlled. However, the bank and page will not automatically be changed with the execution of the program and thus it must be set in accordance with the program specifications.

Therefore a program as described in linear programs can not be created in the 16M-byte address space.
This indicates that multiple modules can not simply be linked.
For this reason, the link88 employs a multi-section method to resolve this problem by allocate voluntary modules in voluntary addresses.
Allocation in this method is undertaken by making it possible to specify addresses for block units referred to as sections.
The section is categorized into a CODE section in which the allocation site is the ROM and the DATA section which is the data memory. To resolve the aforementioned bank and page problems, the size of one CODE section can consist of up to 32K bytes and the size of one DATA section is limited to 64K bytes. It is important to note that this size is based on the fact that they are not allocated over the bank or page limit. If in case they are allocated in the middle of a bank or page, the size will be limited to the remaining size.

To create an object code for the desired multi-section using the section method, the user must define the section and supply address information on the allocation of the section to allocate the address.
The section is defined by using the linker's secondary flag (flag used to define section) +code and +data and the -p flag is used to allocate the address.
Up to a maximum of 255 sections can be defined with one link.

<Example of section definition>

Let's look at the section definition procedures through a simple example.
First, the method to actualize a memory mapping as indicated in Figure 3.4.2.1 will be explained.
It will be assumed that "prg1.s" describing C1 and D1, "prg2.s" describing C2 and "prg3.s" describing C3 is assembled and then each respective relocatable object file "prg1.o", "prg2.o" and "prg3.o" is created.
In this case, C indicates the CODE section and D indicates the DATA section.

The flag to link88 can be specified through input redirect operations.
When the following flag specification is performed and a link command parameter file (filename.1cm) that is used to allocate the address and define the section is created following by executing link88<filename.lcm, a memory mapping as indicated in Figure 3.4.2.1 will be created.

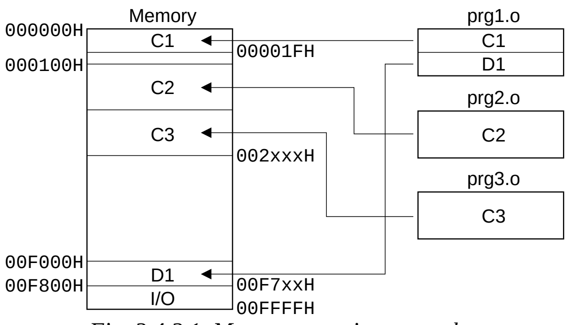


Fig. 3.4.2.1 Memory mapping example

Contents of the file transferred to link88 (link88<filename.lcm)

```
-o prg.a                   ...(1)
+code -p0x000000           ...(2)
+data -p0x00f000           ...(3)
prg1.o                     ...(4)
+code -p0x000100           ...(5)
prg2.o prg3.0              ...(6)
```

(1) Specifies the absolute object file that is output with the -o flag.
(2) Defines the CODE section that starts with a physical address from 000000H.
(3) Defines the DATA section that starts with a physical address from 00F000H.
(4) Allocates "prg1.o" to the sections defined in (2) and (3) indicated above.
In this case, the contents of the CODE section C1 in "prg1.o" will be allocated from the beginning of the CODE section defined in (2) and the contents of the DATA section D1 will be allocated at the head of the DATA section defined in (3).
(5) Defines the CODE section that starts with a physical address from 000100H. This CODE section is different from the CODE section defined in (2). The CODE section (2) will be completed when a new section is defined at this point.
(6) The "prg2.o" CODE section of C2, and "prg3.o" CODE section C3 will be continuously be allocated in respective order.
In this example, "prg2.o" and "prg3.o" does not have a DATA section. However, if there is a DATA section then it will be allocated from the address following D1 of the DATA section defined in (3).

There are three sections defined and linked in this example as indicated above. When the link is successful an absolute object file named "prg.a" will be created.
Multiple modules can be allocated in these sections defined as long as it is within the allowable capacity limit. What's more, multiple sections can be allocated within one bank as well.

<Allocation address and relocation of section>

As indicated in the earlier example, the -p flag determines the physical start address of the section defined immediately before operations.
Let's say, for example, the following settings are made for a certain section.

```
-p 0x10000
```

The start address of this section will physically be 10000H. The CODE section will be specified at the head of bank 2 and the DATA section will be specified at the head of page 1.
The following allocation (reallocation of address information) will be performed for a symbol if a symbol is defined to be positioned from the head of this section to the 1234H offset and that symbol is used to reference that address.

(1) When handled as data memory (symbol name will be indicated as "SYMBOL".)

Operand		***Relocate value***
#SYMBOL	→	#1234H
[SYMBOL]	→	[1234H]
#POD SYMBOL	→	01H
#LOD SYMBOL	→	1234H
#HIGH SYMBOL	→	12H
#LOW SYMBOL	→	34H
[BR:LOW SYMBOL]	→	[BR:34H]

(2) When handled as program memory (symbol name will be indicated as "LABEL".)

Operand		***Relocate value***
#BOC LABEL	→	02H
#LOC LABEL	→	9234H

A relative valued in accordance with the address that allocated by the branch instruction will be calculated and set for PC relative branch instructions like "JRL LABEL".

The section start address, in the above example, was specified at the head of the bank or page, however, specifications can be made for it to start in the middle of a bank or page, as indicated below.

```
-p 0x15000
```

In this case the start address will physically be 15000H and have a 5000H offset from the head of the bank or page. The link88 relocates each symbol based on the physical address, therefore, such offsets will also be properly processed.
All symbol information after reallocation will be recorded in the absolute object file. A list of these symbols can be created using the rel88 symbol information generating utility. Refer to section "3.6.1 Creating symbol information (rel88)" for more information on rel88 operations.

3.4.3 Module allocation information

As indicated in the example of section definition mentioned earlier, section definitions and command lines that specify files can be handed over to the link88 through the input redirect function.
The number of modules are limited and the link is simple, as indicated in the example, it will be possible to create a file similar to that indicated in the example and directly input into the link88.
There will be need to be conscious about the memory efficiency when increasing the number of modules.
One CODE section is limited to 32K bytes and the DATA section is limited to 64K bytes. Thus, it will be necessary to allocate each module so that it does not exceed the limit. It will be necessary to give consideration to the combination of modules in each section upon allocation. Otherwise, there will be more unused memory area and thus, require unnecessary memory extension.

3.4.4 Starting link88

<Operations of link88>

(1) Set the directory in which the relocatable object files (.o) to be linked and the link command parameter file (.lcm) including link88 command line created with the editor are existed as the current drive.

(2) Start-up the link88 with the next format.

`link88`_<_link command parameter file name↵

_ indicates a space key input.
↵ indicates a return key input.

Regardless of the input redirect function, the link command parameter file can directly be input in the command line. The procedures will be omitted since it is not practical. Refer to Part III of this manual for more information on formatting.
Details on the flags that compose the command line will also be omitted.
Refer to Part III of this manual for details of the flags.

Example: Performing linking through the link command parameter file (.lcm)

```
C:\USER>c:\EPSON\link88
                         < sample.lcm↵
```

Use the link command parameter file "sample.lcm" created in the USER of the sub-directory of drive C as the input redirect function to start-up link88 and perform linking.
The name of the absolute object file specified in the link command parameter file will be created in the same directory as the input file.
If the PATH to link88 is set, then there is not need to specify the path before link88.

Refer to Section 3.4.2 for the link command parameter file.

3.4.5 Batch processing for linking (lk88.bat)

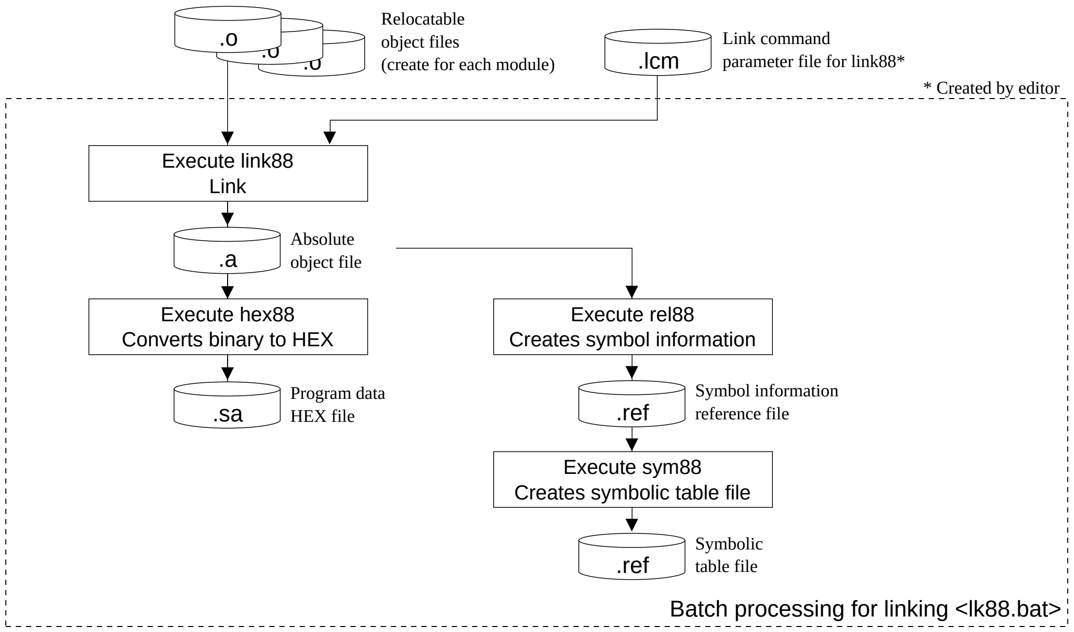

.o
.o
.o
Relocatable object files (create for each module)
.lcm
Link command parameter file for link88*
* Created by editor
Execute link88
Link
.a
Absolute object file
Execute hex88
Converts binary to HEX
.sa
Program data HEX file
Execute rel88
Creates symbol information
.ref
Symbol information reference file
Execute sym88
Creates symbolic table file
.ref
Symbolic table file
Batch processing for linking <lk88.bat>

Fig. 3.4.5.1 lk88.bat processing flow

As so with the assembler, this package contains the lk88.bat batch file for linking.
This batch file is prepared so that it can process the procedures from linking to creation of the program data HEX file. (Details on processing procedures after linking will be noted later.)
Figure 3.4.5.1 shows the processing flow of lk88.bat.

<Outline of processing procedures>

The lk88.bat reads the link command parameter file for the link88 and executes linking operations. When an absolute object file is created using the link88, it will then use the rel88 symbol information generator. After reallocation operations are complete a symbolic table information file will be created. After that, the sym88 will be executed to generate a symbolic table file that is necessary for symbolic debugging using the ICE88.
Then a program data HEX file will be created with the hex88 binary/HEX converter from the absolute object file.

<Input/output files>

■ Input files

1. **Link command parameter file: file_name.lcm**
   This is a command parameter file for the link88. It indicates the information to reallocate the relocatable object of the E0C88 memory space.
2. **Relocatable object file: file_name.o**
   This is a relocatable file in machine language that can be output through relocatable assembly with the cross assembler.

■ Output files

1. **Absolute object file: file_name.a**
   This is the multi-section object file created with the linker.
2. **Program data HEX file: file_name.sa**
   This is a Motorola S2 format ASCII record file consisting of an absolute object file that was converted with the binary/HEX converter.

3. **Symbol information reference file: file_name.ref**
   This is the symbol information reference file of the absolute object file that was reallocated by the physical address.

4. **Symbolic table file: file_name.sy**
   This file contains symbol names and the address list information for symbolic debugging.

<Operation procedure>

(1) Set the directory including the relocatable object files (.o) to be linked as the current drive. Put the command parameter file handed over to the link88 in the same directory.

(2) Start-up the lk88 with the next format.

```
lk88↵
```

↵ indicates a return key input.

Example: `C:\USER>c:\EPSON\lk88↵`

Use the link command parameter file "sample.lcm" created in the USER of the sub-directory of drive C to start batch processing. Batch processing will create the absolute object file (.a), symbol information reference file (.ref), program data HEX file (.sa) and symbolic table file (.sy) in the same directory as the input file. If the PATH to lk88 is set, then there is not need to specify the path before lk88.

■ Customizing lk88.bat

<*Customizing ra88.bat execution parameters*>

Since the lk88.bat controls the program execution, it has a execution parameter customization field within it. General parameters are temporarily described in the default position. Always customize the batch files according to your development method since the parameter will vary depending on your application style.

1. **Parameter file name to be input**

set	parfn	= file_name :	Link command parameter file name (.lcm) input to link88

2. **Output file name**

set	outfn	= file_name :	File name of absolute object file and program data HEX file

3. **Use of the symbol information generator (rel88)**

set	rel88	= y :	rel88 is used (default) A symbol information reference file (.ref) will be created.
		= n :	rel88 is not used.

4. **Use of +sec flag (information on individual section) of the symbol information generator (rel88)**

set	secf	= y :	+sec flag is added to rel88 (default)
		= n :	+sec flag is not added to rel88

Note: This parameter will be ignored when rel88 is not used.

Note: There are basically no error checks made on these parameter settings, therefore, do not set the parameter with settings other than those specified.

<*Customizing lk88.bat execution command*>

The lk88.bat has the following command line upon execution of the program. Customize these command lines if a flag without a default setting is to be used.

link88
```
%drv%link88<%parfn%.lcm
```

rel88 (when +sec flag is used)
```
%drv%rel88 -v +sec
%outfn%.a>%outfn%.ref
```

rel88 (when +sec flag is not used)
```
%drv%rel88 -v %outfn%.a>%outfn%.ref
```

hex88
```
%drv%hex88 -o %outfn%.sa %outfn%.a
```

sym88
```
%drv%sym88 %outfn%.ref
```

The %drv% is a path that locates the execution command of the lk88.bat. For this reason, it can not be altered and neither can the SET statement that is defined be altered.
Use the same name for the customized parameter outfn as the name described in the link command parameter (.lcm).

The following indicates the lk88.bat program source list and the message list of the lk88.bat. Refer to it upon customizing the program.

■ lk88.bat program source list

```
echo off
rem ************************************************************************
rem * E0C88 Family Auto Link Execution Utility
rem *                              (Ver. X.XX)
rem *                        Copyright(C) SEIKO EPSON CORP. 1993–1996
rem ************************************************************************
rem * customized parameter information
rem * parfn=          : input parameter file_name
rem *                   (file_name_lcm) for link88.exe      i.e. c8316xxx.lcm
rem * outfn=          : output file_name which is written
rem *                   in the input parameter file_name    i.e. c8316xxx
rem * rel=y  y : use rel88 for absolute symbol map generation
rem *    =n  n : do not use rel88
rem *
rem * secf=y y : show physical address and module size with absolute
rem *            symbolic table after link procedure
rem *     =n n : do not show physical address and module size just
rem *            symbolic table after link procedure
rem ************************************************************************
rem ********** customized parameter area (default) **********
rem *  caution : customized parameters value do not check, therefore
rem *       please be carefully when you set
rem **********
set parfn=sample                ← Name of link command parameter file to be input
set outfn=sample                ← Name of file to be output
set rel=y                       ← Use of not of rel88
set secf=y                      ← Use or not of the rel88 + sec flag
```

User customization field

Note: There are basically no error checks made on these parameter settings, therefore, do not set the parameter with settings other than those specified.

```
rem ********** command searching path **********
rem set drv=c:\
```

The drv is a path that locates the execution command of the lk88.bat. It is set to root directory by default. Customize it if necessary.

```
rem ************************************************************************
rem * main program
rem *      if you want to use another option(s), please append
rem *      option flag(s) at command line
rem ************************************************************************
:start
     echo E0C88 Family Auto Link Execution Utility Ver. X.XX
     echo Copyright (C) SEIKO EPSON CORP. 1993–1996

:error_chk
           if not exist %drv%nul goto exit05
           if not exist %parfn%.lcm goto exit06
           :chk00
           if not exist %drv%link88.exe goto exit07
           if not exist %drv%rel88.exe goto exit08
           if not exist %drv%hex88.exe goto exit09
           if not exist %drv%sym88.exe goto exit10

:link88
%drv%link88<%parfn%.lcm                                ← Start-up command of link88
           if errorlevel 1 goto exit01

rem (rel88 no sec option)
:rel88_01
           if "%rel%"=="n" goto hex88
           if "%secf%"=="y" goto rel88_02
%drv%rel88 -v %outfn%.a>%outfn%.ref                    ← Start-up command of rel88 (no +sec flag)
           if errorlevel 1 goto exit02
                 goto hex88
```

```
rem (rel88 with sec option)
:rel88_02
%drv%rel88 -v +sec %outfn%.a>%outfn%.ref                    ← Start-up command of rel88 (with +sec flag)
            if errorlevel 1 goto exit02

:hex88
%drv%hex88 -o %outfn%.sa %outfn%.a                          ← Start-up command of hex88
            if errorlevel 1 goto exit03

:sym88
%drv%sym88 %outfn%.ref                                      ← Start-up command of sym88
            if errorlevel 1 goto exit04
                  goto end

      :exit01
      echo Error stop at %drv%link88.exe
            goto skip
      :exit02
      echo Error stop at %drv%rel88.exe
            goto skip
      :exit03
      echo Error stop at %drv%hex88.exe
            goto skip
      :exit04
      echo Error stop at %drv%sym88.exe
            goto skip
      :exit05
      echo Cannot find %drv% installed E0C88 dev. tools directory
            goto skip
      :exit06
      echo Cannot find %parfn% input parameter file
            goto skip
      :exit07
      echo Cannot find %drv%link88.exe
            goto skip
      :exit08
      echo Cannot find %drv%rel88.exe
            goto skip
      :exit09
      echo Cannot find %drv%hex88.exe
            goto skip
      :exit10
      echo Cannot find %drv%sym88.exe
:end
      echo lk88.bat utility has been successfully executed.
:skip
      set parfn=
      set outfn=
      set rel=
      set secf=
      set drv=
```

■ Message list

1. Start-up message

```
E0C88 Family Auto Link Execution Utility Ver. X.XX
Copyright (C) SEIKO EPSON CORP. 1993–1996
```

2. Message when terminated normally

```
lk88.bat utility has been successfully executed.
```

3. Error message

Error message	Explanation
Error stop at [drive and path name] link88.exe	Error occurred in link88.
Error stop at [drive and path name] rel88.exe	Error occurred in rel88.
Error stop at [drive and path name] hex88.exe	Error occurred in hex88.
Error stop at [drive and path name] sym88.exe	Error occurred in sym88.
Cannot find [drive and path name] installed E0C88 dev. tools directory	Cannot find [drive or path] in which the E0C88 Family software tools is installed.
Cannot find [file_name] input parameter file	Cannot find input parameter file (.lcm) that is used with the lk88.bat.
Cannot find [drive and path name] link88.exe	Cannot find link88.
Cannot find [drive and path name] rel88.exe	Cannot find rel88.
Cannot find [drive and path name] hex88.exe	Cannot find hex88.
Cannot find [drive and path name] sym88.exe	Cannot find sym88.

Note: The following operations will be stopped when an error occurs.

<Precautions upon using the batch file>

(1) Some of the messages displayed during batch processing is automatically generated through the MS-DOS/PC-DOS batch processing function and command. For this reason, it may be placed under MS-DOS/PC-DOS control when an error occurs and thus force the batch processing to be interrupted.

(2) When an error occurs, the following procedures do not automatically continue. However, it may not be controllable as noted in reason (1) indicated above.

(3) The execution parameters (user customization field) of the batch file basically do not locate parameter setting errors. Therefore, do not set the parameters other than specified.

(4) An MS-DOS/PC-DOS environment variable will be used to execute the batch file, therefore, the size of the environment variable should be allocated with as much space as possible using the CONFIG.SYS.

3.4.6 Absolute object file

The absolute object file is a binary file created by link88.
The name of the file name created will be the same as that specified with the -o flag.
The files come in a multi-section object format.
This file is composed of an object (machine language) code and various reallocation information.

3.4.7 Execution example of linking

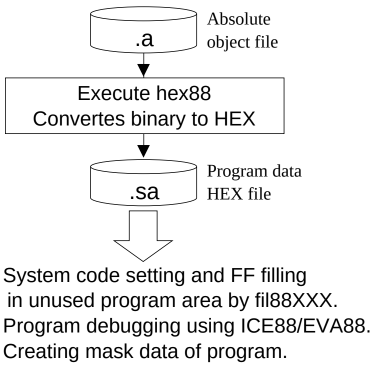
The following shows examples of the lk88 execution.

```
C:\USER>c:\EPSON\lk88

C:\USER>echo off
E0C88 Family Auto Link Execution Utility Ver. X.XX
Copyright (C) SEIKO EPSON CORP. 1993–1996
link88 Linker Version X.XX

Copyright (c) 1993 by Advanced Data Controls, Corp.
Licenced to SEIKO EPSON CORP.
lk88.bat utility has been successfully executed.
C:\USER>
```

3.5 Creating Program Data HEX File

This section will explain the program data HEX file and how they can be created using the hex88 binary/HEX converter.

Software used: hex88



.a
Absolute object file
Execute hex88
Convertes binary to HEX
.sa
Program data HEX file
• System code setting and FF filling in unused program area by fil88XXX.
• Program debugging using ICE88/EVA88.
• Creating mask data of program.

Fig. 3.5.1 Program data HEX file generation flow

3.5.1 Program data HEX file

The program data HEX file is an ASCII file in which the binary object codes were converted in HEX data.
The Motorola S2 format is generally employed at the HEX file format since the E0C88 Family has a 16M-byte address space. (Refer to section 3.5.3 for more information.)
This file will be required to mask program data or to debug program with the ICE88 and EVA88.
When development is undertaken for modules according to relocatable assembly, the absolute object file created by the linker will be converted into HEX data through the hex88 binary/HEX converter and then create a program data HEX file.
The program data HEX file created through such procedures will set system codes according to each model and fill FF of the unused built-in ROM area.
This is done with the fil88XXX software tools according to the model and comes with the "E0C88XXX Development Tool" package.

3.5.2 Creating program data HEX file using hex88

The following indicates the direction in creating a program data HEX file using the hex88.

(1) Set the directory in which the absolute object file (.a) is presented as the current drive.

(2) Start-up the hex88 with the next format.

hex88_[flag]_file name↵

_ indicates a space key input.
↵ indicates a return key input.

The following indicates the flag employed during batch processing (lk88.bat) of links.

Flag	Description
`-o` <file name>	Specify the file name that is output. (Specify ".sa" as the extension of the file to be output.) If this flag is omitted it will be processed as a standard output.

Example: Converting sample.a to create program data HEX file

```
C:\USER>c:\EPSON\hex88 -o
           sample.sa sample.a↵
```

"sample.sa" will be created in the same directory as the input file by inputting the absolute object file "sample.a" created in the USER of the sub-directory of drive C and converting it into HEX data format.
If the PATH to hex88 is set, then there is not need to specify the path before hex88.

The batch file can allow for hex88 to be executed after linking. Refer to section "3.4.5 Batch processing for linking (lk88.bat)" for more details on such batch processing methods.

3.5.3 Motorola S2 format

The HEX file in the Motorola S2 format is a collection of records composed of fields like the following.

<S FIELD><COUNT><ADDR><DATA BYTES><CHECKSUM>

All information will be indicated in hexadecimal pairs and each pair will indicate a 1-byte value.

<S FIELD>	Indicates the format of that line. "S2" will appear in this field.
<COUNT>	Indicates the total number of bytes of <ADDR>, <DATA BYTES> and <CHECKSUM> in hexadecimal form.
<ADDR>	Indicates the address of the first data byte of that line. The <ADDR> field in S2 format is 3-byte.
<DATA BYTES>	Data will be allocated in 1 byte units in order of the increase in address. This field generally includes the 32-byte (maximum) data.
<CHECKSUM>	This is the complement of 1 of the total number of bytes allocated to that line (excluding S field).

Motorola S2 format

```
S224000380788812CF7C8812CFC0CFC1CFC2CFC3CFC4CFC5CFC6CFC7CFD0CFD1CFD2CFD3CF7C
S2240003A0D4CFD5CFD6CFD7CFD8CFD9CFDACFDBCFDCCFDDCFDECFDFCFE0CFE1CFE2CFE3CF90
S2240003C0E4CFE5CFE6CFE7CFE8CFE9CFEACFEBCFECCFEDCFEECFEFCFF0CFF1CFF2CFF3CE71
S2240003E0F4CEF5CEF8CEF9CFFACFFEDD8812C8C8C9C9CACACCCCCCCCDCDA8A9AAABACAD28
S224000400AEAFCFB4CFB5CFB6CFB7CFBCCFBDA0A1A2A3A4A5A6A7CFB0CFB1CFB2CFB3CFB8AC
S224000420CFB9F6F7CE94CE95CE9688CE97CE90CE91CE9288CE93CE9CCE9DCE9E88CE9FCE22
S22400044098CE99CE9A88CE9BCE80CE81CE8288CE83CE84CE85CE8688CE87CE88CE89CE8A9E
S22400046000CE8BCE8CCE8DCE8E88CE8FE438E536E634E732CEE02FCEE12CCEE229CEE326CE
└┘└┘└──────┘└─────────────────────────────────────────────────────────────┘└┘
 │ │    └ <ADDR>                      <DATA BYTES>                       <CHECKSUM>
 │ └───── <COUNT> 32-byte
 └─────── <S FIELD>
```

3.6 Symbol Information

3.6.1 Creating symbol information (rel88)

The rel88 is a utility used to create symbol information. It will obtain symbol information from the specified object file and then create its list. The target object files are the relocatable object file created with asm88 and the absolute object file created with link88.

Generally, this tool is used for two purposes: one for checking the symbol list after linking and second for generating a file to be input to the sym88.

The rel88 outputs a list in accordance with the standard output.
The following explains the operations to obtain the symbol list of an absolute object file.

<rel88 operation procedure>
When creating a symbol list for the absolute object file

(1) Set the directory in which the absolute object file (.a) is presented as the current drive.

(2) Start-up the rel88 with the next format.

`rel88_[flag]`_input file name_>_output file name↵

_ indicates a space key input.
↵ indicates a return key input.

General flags

Flag	Description
+sec	Outputs the start address and size of each section.
-v	Sorts the sections contents according to the symbol value.

Refer to the following examples for information on the flag effects. Refer to Part III of this manual for more details on the flag.

Since the rel88 output corresponds to the standard output, a file will be created according to the output redirect.

Example: `C:\USER>c:\EPSON\rel88 -v +sec sample.a > sample.ref`↵

Inputs the absolute object file "sample.a" created in the USER of the sub-director of drive C and then creates the symbol list file "sample.ref" in the same directory as the input file.
If the PATH to rel88 is set, then there is not need to specify the path before rel88.

The following indicate the list of symbols that are created.

■ Correlation with flag

```
*** rel88 (default) format ***

0x8000c         acia.o
0x80b8d         acia.o
0x8000C  n_getch
0x80bcD  _buffer
0x8059C  n_recept
0x8045C  n_outch
0x80baD  _ptlec
0x80b8D  _ptecr
0x8082C  n_main

*** rel88 -v format ***

SECTION 1
0x008000 c      acia.o
0x008000 C  n_getch
0x008045 C  n_outch
0x008059 C  n_recept
0x008082 C  n_main

SECTION 2
0x0080b8 d      acia.o
0x0080b8 D  _ptecr
0x0080ba D  _ptlec
0x0080bc D  _buffer

*** rel88 +sec format ***

SECTION 1:  code
       address = 0x008000  size = 0x000b8

SECTION 2:  data
       address = 0x0080b8  size = 0x00000
```

(For reference)

```
*** -a format ***

0x000000 c  sec: 1      acia.o
0x0000b8 d  sec: 2      acia.o
0x0000bc D  sec: 2 _buffer
0x0000b8 D  sec: 2 _ptecr
0x0000ba D  sec: 2 _ptlec
0x000000 C  sec: 1 n_getch
0x000082 C  sec: 1 n_main
0x000045 C  sec: 1 n_outch
0x000059 C  sec: 1 n_recept

*** -d format ***

0x000000 c      acia.o
0x0000b8 d      acia.o
0x000000 C  n_getch
0x0000bc D  _buffer
```

```
0x000059 C  n_recept
0x000045 C  n_outch
0x0000ba D  _ptlec
0x0000b8 D  _ptecr
0x000082 C  n_main

*** -g format ***

0x000000 C  n_getch
0x0000bc D  _buffer
0x000059 C  n_recept
0x000045 C  n_outch
0x0000ba D  _ptlec
0x0000b8 D  _ptecr
0x000082 C  n_main

*** +dec format ***

       0 c      acia.o
     184 d      acia.o
       0 C  n_getch
     188 D  _buffer
      89 C  n_recept
      69 C  n_outch
     186 D  _ptlec
     184 D  _ptecr
     130 C  n_main
```

3.6.2 Creating symbolic table file (sym88)

The sym88 symbolic table file generator converts symbol information reference (.ref) output from the rel88 symbol information generator into an information file that contains a symbolic table for symbolic debugging in the ICE88.

<sym88 operation procedure>

(1) Set the directory in which the symbol information reference file (.ref) is presented as the current drive.

(2) Start-up the sym88 with the next format.

sym88_input file name↵

_ indicates a space key input.
↵ indicates a return key input.

Example: `C:\USER>c:\EPSON\sym88 sample.ref↵`

Inputs the symbol information reference file "sample.ref" created in the USER of the sub-director of drive C and then creates the symbolic table file "sample.sy" in the same directory as the input file.
If the PATH to sym88 is set, then there is not need to specify the path before sym88.

II CREATING PROCEDURE OF ASSEMBLY SOURCE FILE

Part II explains details of the parts relative to an assembly source file creation such as the assembly source file format and the pseudo-instructions of which the structured preprocessor sap88 and the cross assembler asm88 are included.

Contents

1 OUTLINE

When you develop a program using the assembly language, first create an assembly source file using the CPU instructions and the pseudo-instructions included with the cross assembler. The assembly source file should be created according to the contents and rules to be explained hereafter, using EDLIN or an editor you have.

1.1 File Name

As explained in Section 3.3 of Part I, this assembler is separated into two programs: the structured preprocessor sap88 which expands macro instructions into the format that can be assembled by the asm88, and the cross assembler asm88 which actually executes assembly. Files to be handled in this series of procedures are an assembly source file. However, since there are some difference in each file, extensions of the file names are specified as below.

- **Structured assembly source file: file_name.s**
  This is an assembly source file which includes macro instructions, etc., and is input into the structured preprocessor sap88. When you create programs using the assembler language, create assembly source files to make the file name with the extension ".s".
- **Assembly source file: file_name.ms**
  This is an assembly source file in which the macro instructions have been expanded, and is generated from the structured preprocessor sap88.

In the structured preprocessor sap88 and the cross assembler asm88, files with other extensions can be input, but generally use the above mentioned extension.

1.2 Source File Differences Depending on sap88 and asm88

As explained in the previous section, format of the file to be input to the cross assembler asm88 is different from that of the structured preprocessor sap88 as to contents.
The statement (line) such as macro instruction and sap88 pseude-instruction, which can be used in the structured preprocessor sap88, cannot be distinguished in the cross assembler asm88, and will cause an error. Consequently, when using the macro instructions, be sure to expand it to the format which can be input into the cross assembler asm88, using the structured preprocessor sap88.
In particularly, attention should be paid when modifying the source file ".ms" being input into the asm88 directly.
The pseudo-instructions which are incorporated in the cross assembler asm88 functions will not cause an error in the structured preprocessor sap88.
In the pseudo-instructions explained later, details for only the structured preprocessor sap88 are indicated by [sap88 only] or the notes are described. Take care when reading.

1.3 Macro Instructions

Macro instruction allows the user to define virtual instructions with instruction sequences. The structured preprocessor sap88 expands the defined instructions into the source format that can be assembled by the cross assembler asm88.
The following describes the outline of it.

When using the same statement block in multiple parts of a program, previous define the statement block with an optional name, after this the statement block can be called using the defined name. The defined statement block is Macro.
Describe the macro name that has been defined and necessary parameters in program, to call the macro. That part is expanded in the contents of the statement block that have been defined as a macro by the structured preprocessor sap88, and at that point the changing of the specified parameters is also to be done.
In addition to the macro-definition and the macro-call, some pseudo-instructions related to the macro have been provided. For details, see Section 3.8.

2 GENERAL FORMAT OF SOURCE FILE

Assembly source file is composed of statements (lines) such as the CPU instruction set, pseudo-instructions which are incorporated in the sap88 and asm88, and comments, and is completed by END pseudo-instruction (pseudo-instruction to terminate assembly). (Statements can be described after the END pseudo-instruction, however, that part will not be assembled.)
The following explains the asm88 fundamentally. (Functions permitted on the asm88 will not cause an error on the sap88.)

Example of source file

```
      subtitle    "assembly source file example (sample.s)"
      public      main
      external    src_address, dst_address, counter
;
      code
main:
      ld    ix,[src_address]
      ld    iy,[dst_address]
      ld    hl,[counter]
      ret
;***
      end
```

The following explains the general particulars such as the composition of the statement and characters and notation for numerical values which can be used.

Each source program statement should be written using the following format.

Symbol field	Mnemonic field	Operand field	Comment field

Example:

```
    on                      equ                     1000h
    start:                  jrl                     init                    ;to initialize
    flag:                   db                      [1]
    value:                  db                      080h
```

In the above sort of format line, the line end normally is the termination, however, the operand may be described over several lines.

Symbol field: In this field, describe a symbol. A colon (:) must be used following the symbol except for the statement of the EQU or SET instruction.
Use symbols properly in accordance with the following definition.

Symbol
- •Label (Colon must follow)
- •Name (Constant definition by EQU or SET instruction)

Mnemonic field: In this fild, describe an operation code or a pseudo-instruction.

Operand field: In this field, describe an operand or constant of each instruction, a variable, a defined symbol, a symbol that indicates memory address, or an operational expression.

Comment field: Put semicolon (;) at the beginning of this field, and describe a comment following it.

2.1 Symbol

Symbol is the name in which the specific value is defined. The following two ways are to define a symbol.

(1) Label

The symbol that is put at the beginning of statement of CPU instructions or data definition is defined as a label. The value that is defined to the symbol is the address of the CPU instruction or data area.

(2) Name

It is defined using the EQU or SET pseudo-instruction. The value that is defined to the symbol is the value of <expression> that is specified using the EQU or SET pseudo-instruction.

The symbol definition is in accordance with the following rules.

- Although the symbol length is not restricted, a maximum of 15 characters from the front will be distinguished as a symbol.
- In the case of a label, it can be described from any column, however, a colon (:) must be used at the end of a label.
- In the case of a name, it must begin from column 1.
- The characters that can be used for symbols are as follows:
  Alphabetic characters (A–Z, a–z), Arabic numerals (0–9), _
- To input symbol it does not matter whether capital letters or small letters are used. In the default setting, capital letters and small letters are not distinguished, therefore symbols ABC and abc are handled identically. However, when the -c flag is used, they are distinguished.
- A symbol cannot begin with a number. Symbol names must begin with an alphabetic character or "_".

2.2 Mnemonic

A CPU instruction or a pseudo-instruction is placed in the mnemonic field. These are normally composed of character-strings that end with a blank space.These are discussed later.

In the default setting of the asm88 and sap88, capital letters and small letters are not distinguished. In such cases, even if inputting the following, they will all be considered as correct and the same.

Examples: `byte  BYTE  bYtE`

In the default setting, it is also permissible for a CPU instruction set to be written either in capital letters or small letters. When writing programs, it is better to write them with the standard method. However, when handling the symbol name to distinguish between capital letters and small letters using -c flag, be sure to describe the CPU instruction set and register name in small letters.

Example:

```
      jrl   ABC   ;jump to label ABC
      ld    a,b   ;A register <- B register
```

2.3 Operand

0 or more operands can be placed in accordance with the content of the mnemonic field. These operands are allocated by the parameter strings. They begin from a blank character indicating the termination of the mnemonic field, are delimited by a comma and end with a blank character or semicolon.

2.4 Comment

Comments are disregarded in the process of assembly. The comment begins with a ";" (semicolon) and ends at the termination of the line end (line feed code).

2.5 Numerical Expression

Bit control is frequently executed in a microcomputer built into the equipment. For this reason, asm88 and sap88 can handle binary, octal, hexadecimal and decimal expressions as the radix of numerical expression.

The radix is recognized by placement of the following characters after the number.

B:	Binary
O, Q:	Octal
H:	Hexadecimal
None:	Decimal (D can be used.) (These may also be written as small letters.)

The numbers must begin with Arabic numerals (0–9). For example, the number "10" can appear as follows.

10:	Decimal
1010B:	Binary
12Q:	Octal
0AH:	Hexadecimal (To distinguish from names all hexadecimal numbers using letters A to F must have a "0" in front. eg. 0AH = HEX number, AH = name)

2.6 Characters

The sap88 and asm88 have adopted the notation that has been normally called ASCII (American Standard Code for Information Interchange) for expression of characters and character strings.

2.7 ASCII Character Set

The ASCII character set code is composed of two parts: 7 bits data according to the characters and 1 bit parity to check whether there is an error during transfer. The ASCII character set is classified into the following four types.

Table 2.7.1 ASCII character code table

L \ H	00	01	02	03	04	05	06	07
00	NUL	DEL	SP	0	@	P	`	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EOT	DC4	$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	'	7	G	W	g	w
08	BS	CAN	(	8	H	X	h	x
09	HT	EM	)	9	I	Y	i	y
0a	LF	SUB	*	:	J	Z	j	z
0b	VT	ESC	+	;	K	[	k	{
0c	FF	FS	'	<	L	\(¥)	l	|
0d	CR	GS	-	=	M	]	m	}
0e	SO	RS	.	>	N	^	n	~
0f	SI	US	/	?	O	_	o	DEL
	00		01		10		11	

Section

In the asm88, the notation characters can be handled as a character constant by enclosing them with single quotation marks such as 'A', 'Z' and 'X'. '\'' is particularly used for the single quotation marks themselves.

To express a character which can not be displayed such as a control code, the asm88 permits the following notations for control characters thought to have a particularly high usage frequency.

'\a'	Bell	(07H)
'\n'	New-line	(0AH)
'\r'	Return	(0DH)
'\t'	Tab	(09H)
'\b'	Back space	(08H)
'\e'	Escape	(1BH)
'\i'	Shift-in	(0FH)
'\o'	Shift-out	(0EH)

The notation, \nnn (nnn is an octal), can also be used. When this notation is used, bell, for example, can be written '\007'.

These descriptions by escape sequences are only permitted in character strings. The character string can be handled by ASCII instruction, and they can also be expressed by sets of characters enclosed by single quotation marks.

2.8 Expressions

Constants are set at many points within programs, for example, the operands for CPU instruction set and the parameters for pseudo-instructions. Moreover, constants can be shown using expressions. The cross assembler asm88 evaluates expressions and can make the result value into the constant. A variable of the same size as the numbers used by the CPU or a larger one may be used for the expression evaluation during assembly.

NOTE:

(1) When a relocatable code is made, the address can only be used within the expression of which the result will be a quantity that becomes relocatable or a constant.

Consequently, the following expressions may be used.

```
label1 - label2 ;When two labels are in the
                 same program selection
label1 + <constant>
label1 - <constant>
```

The following expressions may not be used because the result will not be a relocatable quantity or a constant.

```
label1 + label2
label1 & label2
label1 * <constant>
label1 / <constant>
label1 % <constant>
label1 * label2
label1 / label2
label1 % label2
<constant> + label2
label1 - label2 ;When two labels are in the
                 different program selection
```

(2) Since the results do not become relocatable quantity, logic operations using a relocatable address become errors during assembly.

Expressions are composed of several terms linked by binary operators (for example, +). In the evaluation, these expressions are calculated with 16-bit precision.
The following terms may be used within the expressions.

1 Numbers
2 Variables which have been defined by the user to use the EQU and SET instructions, and declared labels
3 Location counters $

When $ is used as the operand for the CPU instructions, the address immediately preceding the instruction is applied.

The asm88 is a two pass assembler and the values for several variables which are used in program are not defined in the pass 1 stage. When variables for which values are undefined appear within expressions during the pass 1 execution, 0 is assigned for them. And if there are variables for which values are still undefined in pass 2 execution, an error results. Also, if variables which were undefined when used for the expression in pass 1 are used in pass 2, it causes a phase error. Consequently, you should define the values for variables prior to using them in an expression.

2.9 Operators

The asm88 accepts the following operators.

Table 2.9.1a Unary operator (1)

Operator	Function
+a	Positive sign Example: `ld a,#+25h`
-a	Negative sign Example: `add b,#-13h`
~a	Assigns the values reversing each bit. Example: `and a,#~10h`
LOW a	Assigns a lower 8-bit value of an expression. Example: `or b,#low 1234h`
HIGH a	Assigns a lower 8-bit value of an expression after the expression value is shifted 8-bit to the right. This is the same as that to return the upper 8-bit of a 16-bit expression. Example: `ld h,#high 1020h`
BOC	Calculates a bank value from a physical address. This operator is effective for a physical address. (Bank Of Code) Example: `ld a,#boc label` `ld nb,a`

LOC	Calculates a logical address within the logical space from a physical address. This operator is effective for a physical address.
	(Logical address Of Code)
	Example: `ld hl,#loc label`
	`jp hl`
	`:`
	`label:`

Table 2.9.1b Unary operator (2)

Operator	Function
POD	Calculates a page value from a physical address. This operator is effective for a physical address. (Page Of Data) Example: `ld a,#pod label` `ld ep,a`

LOD	Calculates a logical address within the page from a physical address. This operator is effective for a physical address.
	(Logical address Of Data)
	Example: `ld ix,#lod label`
	`ld a,[ix]`
	`:`
	`label:`

Table 2.9.1c Binary operator

Operator	Function
a+b	Addition (32-bit signed integer) Example: `sbc [hl],#25h+10h`
a-b	Subtraction (32-bit signed integer) Example: `sub a,#63h-03h`
a*b	Multiplication (32-bit signed integer) Example: `xor l,#48h*5h`
a/b	Integer division (32-bit signed integer) Example: `cp ba,#1256h/31h`
a%b	Remainder. Divides the left operand by the right operand, and returns the remainder. Example: `add a,#0d7h%4fh`
a&b	Logical AND. Returns true if both operands are true. Returns false if either of the operands is false or both operands are false. Example: `ld sp,#04a1h&2030h`
a|b	Logical OR. Returns true if either operand is true or both operands are true. Example: `ld ix,#3026h|1000h`
a^b	Exclusive OR. Returns true if one operand is true and the other is false. Returns false if both operands are true or false. Example: `ld [iy],#44h^10h`
a<<b	Shift to left. Shifts b (integer) bits to the left. Example: `adc hl,#5000h<<3`
a>>b	Shift to right. Shifts b (integer) bits to the right Example: `cp ba,#8130h>>10h`

• Priority for operators

An expression is evaluated from left to right, however, an operator with higher priority is evaluated earlier than the other operators immediately in front of or behind it. If there are two or more continued operators equal in priority, the operators are evaluated from the left side.
Every left parenthesis "(" must have a corresponding right parenthesis ")".
The following table shows the priority for operators.

Table 2.9.2 Priority for operators

Operators	Priority
|, ^, &	Low
+ (addition), - (subtraction)	↑
*, /, %, <<, >>	
BOC, LOC, POD, LOD	↓
HIGH, LOW, ~, -, +	High

• Operation rules for BOC, LOC, POD and LOD

In the unary operators, four operators BOC, LOC POD and LOD are peculiar to the E0C88, and possesses original rules for operation as the below.

```
BOC   (physical address & 0x7f8000) >> 15
LOC   If (physical address & 0x7f8000)
              (physical address & 0x7fff) | 0x8000
      else
              (physical address & 0x7fff) | 0x0000
POD   (physical address & 0xff0000) >> 16
LOD   (physical address & 0xffff)
```

In the above, the value indicates the physical value possessed by the operand. During assembly, the asm88 only generates special relocation information corresponding to each operator and the actual address calculation is done by the link88 during linking.

2.10 Instruction Set

The asm88 accepts each of the following instructions as CPU instruction set.

E0C88 Family instruction list

```
adc  cp   inc  neg  rete sep  swap
add  cpl  int  nop  rets sla  upck
and  dec  jp   or   rl   sll  xor
bit  div  jrl  pack rlc  slp
call djr  jrs  pop  rr   sra
carl ex   ld   push rrc  srl
cars halt mlt  ret  sbc  sub
```

2.11 Register Name

The CPU register names indicated in the following have been reserved as keywords in the asm88. Refer to the "E0C88 Core CPU Manual" for information on the respective register functions.

a	Data register A
b	Data register B
ba	A and B register pair
h	Data register H
l	Data register L
hl	Index register HL
ix	Index register IX
iy	Index register IY
sp	Stack pointer SP
br	Base register BR
sc	System condition flag SC
pc	Program counter PC
nb	New code bank register NB
cb	Code bank register CB
ep	Expand page register EP
xp	XP expand page register for IX
yp	YP expand page register for IY
ip	XP and YP register

2.12 Addressing Mode

The E0C88 determines the execution address according to the following 12 types of addressing modes.

Table 2.12.1 List of E0C88 addressing modes

No.	Addressing mode
1	Immediate data addressing
2	Register direct addressing
3	Register indirect addressing
4	Register indirect addressing with displacement
5	Register indirect addressing with index register
6	8-bit absolute addressing
7	16-bit absolute addressing
8	8-bit indirect addressing
9	16-bit indirect addressing
10	Signed 8-bit PC relative addressing
11	Signed 16-bit PC relative addressing
12	Implied register addressing

Refer to the "E0C88 Core CPU Manual" for details on each addressing mode. The notation rules for the operands corresponding to these addressing modes are as follows.

Table 2.12.2 Notation rules for operands

No.	Notation rule
1	A "#" is to be placed in front of numeric expressions and symbols
2	Register name is to be written directly
3	Index register is to be enclosed by brackets ([ ])
4	Index register and displacement are to be enclosed by brackets ([ ])
5	Index register + L is to be enclosed by brackets ([ ])
6	A "BR:" is to be placed in front of numeric expres-sions and enclosed by brackets ([ ])
7	Numeric expressions and symbols are to be enclosed by brackets ([ ])
8	Numeric expressions and symbols are to be enclosed by brackets ([ ])
9	Numeric expressions and symbols are to be enclosed by brackets ([ ])
10	Numeric expressions and symbols are to be written directly
11	Numeric expressions and symbols are to be written directly
12	None

2.13 Example for Mnemonic Notation

The examples for mnemonic notation in each addressing mode are shown in the below.

Addressing	Constant	Name name equ 50h	Label (default) label: address 00ffh	Default definition
#nn 0 to 255	eg.) ld a,#0ffh	eg.) ld a,#name	eg.) ld a,#label	-----
#mmnn 0 to 65535	eg.) ld ba,#1000h	eg.) ld ba,#name	eg.) ld ba,#label	-----
[br:ll] 0 to 255	eg.) ld b,[br:0ffh]	eg.) ld b,[br:name]	eg.) ld b,[br:label]	[br:low lod label]
[hhll] 0 to 65535	eg.) ld 1,[1000h]	eg.) ld l,[name]	eg.) ld l,[label]	[lod label]
[ix+dd] [iy+dd] [sp+dd] -128 to 127	eg.) ld [ix+10h],a	eg.) ld [ix+name],a	-----	-----
#hh 0 to 255	eg.) ld br,#0ffh	eg.) ld br,#name	eg.) ld br,#label	high lod label
#pp 0 to 255	eg.) ld ep,#05h	eg.) ld ep,#name	eg.) ld ep,#label	pod label
#bb 0 to 255	eg.) ld nb,#05h	eg.) ld nb,#name	eg.) ld nb,#label	boc label
rr -128 to 127	eg.) jrs 10h	eg.) jrs name	eg.) jrs label	loc label
[kk] 0 to 255	eg.) jp [10h]	eg.) jp [name]	eg.) jp [label]	[low lod label]
qqrr -32768 to 32767	eg.) jrl 1000h	eg.) jrl name	eg.) jrl label	loc label

- Meaning of the above mentioned default definitions are as follows:
  For example, when "jrl label" has been described, the cross assembler asm88 judges as "jrl loc label".

  ```
  jrl  label  →  jrl  loc label
  ```

  The program sequence is long jumped to the logical address converted from the physical address.

- An error occurs when the operand exceeding the above mentioned addressing range has been specified, or when it is judged to exceed it.

- In programming, pay attention to the following points when using the short branch or long branch instruction.

`jrs(l)  10H`	Jumps to the address at a distance of (10+1)H from current address
`jrs(l)  $+10H` ...	Jumps to the address at a distance of 10H from current address

Except for the above, notations described in the "E0C88 Core CPU Manual" can be used as is.

3 PSEUDO-INSTRUCTIONS

In this chapter the usage of each type of pseudo-instruction supported by the asm88 and sap88 is explained in the form classified by function. The format as explained below has been adopted for each explanation to permit reference to it at any time.

View of the explanation

The explanation contents of each pseudo-instruction have been configured as the following format.

1) Name

Name of the pseudo-instruction . . . Function of the instruction

2) Format

Here the instruction format is described. The format is explained using notations according to the following rules.
The explanations of the respective terms used in the operand notations are as follows.

<Expression>
General expression composed of symbols and constants including operators

<Numerical expression>
Constant expression using a numerical value expression (including name which has been defined as constant by EQU instruction)

<Label>
Symbols having a definition within the self-module that has a relocatable property

<Name>
Symbols defined by EQU and SET instructions

<Symbol>
Name to be defined for the specific value

<Character string>
Character strings enclosed by double quotation marks

The following symbols have been given special meanings.

{ } ... The enclosed part indicated an optional selection.
{ }* .. This option may be placed repeatedly any number of times.
| | | .. When different parameters of a number of different types can be adopted, one among them that is delimited by this symbol must necessarily be used as a parameter.

Other symbols
Commas ","s, brackets "[" and "]", and parentheses "(" and ")" may be input as assembler sources.

3) Functions

Here the operations of the instruction are explained in detail.

4) Examples

Here usage examples are indicated. Several types may be written depending on the instruction.

5) Related items

Here instructions that function in a similar manner and instructions that assist in understanding are indicated.

6) Restriction

Here restrictions for use are provided. Also, causes of errors that occur in the use of an instruction (forgetting the separator, for example) are explained.

3.1 Section Setting Pseudo-Instructions

The section setting pseudo-instructions set each section (code section and data section) and decides program area. The section setting pseudo-instructions are as follows:

CODE DATA

The section setting pseudo-instruction of the cross assembler asm88 has been defined on assumption that the code section should be allocated into ROM and data section into RAM. It aims that the non-volatile data such as program codes and constant data should not be assigned into RAM, since the microcomputer to built into an equipment has RAM area that the initial values become undefined. Therefore, when the non-volatile data such as program codes and constant data are described, it must be described within code section to set the code section by CODE pseudo-instruction. When the volatile data such as work area and stack area are described, it must be described within data section to set the data section by DATA pseudo-instruction.

Correspondence of each pseudo-instruction, setting section, area used, and contents to be described are shown in table below.

Section name	Area used	Contents to be described
Code section (CODE)	ROM	Data allocation that is necessary to decide from the power on, such as program code, constant data, and table.
Data section (DATA)	RAM	Reservation for data area that does not matter if the initial value is undefined at power on, such as work area, stack area, flags, and buffers.

Name:

CODE.....Definition of program section

Format:

CODE

Functions:

This instruction is used to allocate the program and constants in the CODE section (ROM area). An optional number of CODE sections may be defined within one module and resumed during assembly. Since this instruction specifies the section with the same function as the DATA pseudo-instruction, be sure to specify which when in the assembly. When it has not been specified, an error message is output.

Example:

Defines the program and constants in the code section.

```
       code
trans: ld     [iy],[ix]
       inc    ix
       inc    iy
       djr    nz,trans
       ret
       db     01h, 02h, 03h, 04h, 05h
```

Related items:

DATA, ORG

Name:

DATA.....Definition of data section

Format:

DATA

Functions:

This instruction is used to reserve and allocate the data area in the DATA section (RAM area). An optional number of DATA sections may be defined within one module and resumed during assembly. Normally, the data section definition performs only area reservation, and it is not output to the object as a result of the assembly. However, this section is a RAM area. When using equipment with built in microcomputer, pay attention that the RAM area is undefined at the power on and the initial values are invalidated.
Since this instruction specifies the section with the same function as the CODE pseudo-instruction, be sure to specify which when in the assembly for the data section. When it has not been specified, an error message is output.

Example:

Reserves an area for flag and buffer table in the data section.

```
         data
flag:    db    [1]
buffer:  db    [256*8]
```

Related items:

CODE, ORG

3.2 Data Definition Pseudo-Instructions

Data definition pseudo-instruction is the pseudo-instruction to define data to be stored into the memory. The data definition pseudo-instructions are as follows:

DB DW DL ASCII PARITY

Name:

DB .... Reserve/constant setting of the byte unit data area

Format 1:

DB <expression> {,<expression>}*

Format 2:

DB <expression> (<numeric expression>)
{,<expression> (<numeric expression>)}*

Format 3:

DB [<numeric expression>]
{,[<numeric expression>]}*

Functions:

This instruction is used to reserve the 1 byte unit data area and to set the constant. The setting of constants are done according to a string of numeric values delimited by a comma or the specification for the repeat number. The parameters for this instruction can be described over several lines, but you should take care that the relocation information for linking are not included. Further when this instruction is used, it should be described within the DATA (RAM) area when reserving data area, and within the CODE (ROM) area when setting constant. The code generation rules for each format are as follows.

- Format 1
  This format defines the optional constant as the optional number of object codes in 1 byte unit and multiple expressions can be specified for an operand field. The expression is handled as constant value of 1 byte and when multiple specifications are made, the object codes are generated in the order of specification.
- Format 2
  This format repeat defines the optional constant in 1 byte units and sets the repeat number in a <numeric expression> enclosed by parentheses.
- Format 3
  This format reserves the area for the number of bytes that have been assigned by the <numeric expression> enclosed by brackets. The code generated within the object at this time is 0.

Integer numeric constants, character constants and symbols can be used as the expressions for formats 1 and 2, but they must necessarily have an absolute numeric attribute. The value of the expression must also be within the range of -128 to 255. When an operation result is outside the above range, it will be made an error and the value of the lower 1 byte will be made the evaluation value. Each format can be premixed for one instruction.

Examples:

```
buffer: db   [50]      ;Reserves 50 bytes area
tratbl: db   '0','1','2','3','4','5','6','7',
             '8','9','A','B','C','D','E','F'
                       ;Reserves 16 bytes data as the
                        constant
xhrbuf: db   ' '(64)   ;Reserves 64 bytes and
                        initializes at the character code
                        for the space
        db   '*'(64)   ;Reserves 64 bytes '*' as the
                        constant
```

Related items:

DW, DL

Name:

DW .... Reserve/constant setting of the word unit data area

Format 1:

DW <expression> {,<expression>}*

Format 2:

DW <expression> (<numeric expression>)
{,<expression> (<numeric expression>)}*

Format 3:

DW [<numeric expression>]
{,[<numeric expression>]}*

Functions:

This instruction is used to reserve the word (2 bytes) unit data area and to set the constant. The setting of constants are done according to a string of numeric values delimited by a comma or the specification for the repeat number. The parameters for this instruction can be described over several lines. Further when this instruction is used, it should be described within the DATA (RAM) area when reserving data area, and within the CODE (ROM) area when setting constant. The code generation rules for each format are as follows.

- Format 1
  This format defines the optional constant as the optional number of object codes in word (2 bytes) units and multiple expressions can be specified for an operand field. The expression is handled as a long word constant value or symbol value and when multiple specifications are made, the object codes are generated in the order of specification.
- Format 2
  This format repeat defines the optional constant in word units and sets the repeat number in a <numeric expression> enclosed by parentheses.
- Format 3
  This format reserves the area for the number of words that have been assigned by the <numeric expression> enclosed by brackets. The code generated within the object at this time is 0.

Integer numeric constants, character constants and symbols can be used as the expressions for formats 1 and 2. When the expression has a relocatable quality, the logical address of the location where the concerned symbol has been allocated is rearranged during linking. The value of the expression must also be within the range of -32766 to 65535. When an operation result is outside the above range, it will be made an error and the value of the lower 2 bytes will be made the evaluation value. Each format can be premixed for one instruction.

Examples:

```
array:   dw   [10]     ;Reserves 10 word size area

         external      func1,func2,func3,
                       func4,func5
jmptbl:  dw   func1,func2,func3,func4,func5
                       ;Jump table of the functions
```

Related items:

DB, DL

Name:

DL ..... Reserve/constant setting of the long word unit data area

Format 1:

```
DL   <expression> {,<expression>}*
```

Format 2:

```
DL   <expression> (<numeric expression>)
     {,<expression> (<numeric  expression>)}*
```

Format 3:

```
DL   [<numeric expression>]
     {,[<numeric expression>]}*
```

Functions:

This instruction is used to reserve the long word (4 bytes) unit data area and to set the constant. The setting of constants are done according to a string of numeric values delimited by a comma or the specification for the repeat number. The parameters for this instruction can be described over several lines. Further when this instruction is used, it should be described within the DATA (RAM) area when reserving data area, and within the CODE (ROM) area when setting constant. The code generation rules for each format are as follows.

- Format 1
  This format defines the optional constant as the optional number of object codes in long word (4 bytes) units and multiple expressions can be specified for an operand field. The expression is handled as a long word constant value or symbol value and when multiple specifications are made, the object codes are generated in the order of specification.
- Format 2
  This format repeat defines the optional constant in long word units and sets the repeat number in a <numeric expression> enclosed by parentheses.
- Format 3
  This format reserves the area for the number of long words that have been assigned by the <numeric expression> enclosed by brackets. The code generated within the object at this time is 0.

Integer numeric constants, character constants and symbols can be used as the expressions for formats 1 and 2. When the expression has a relocatable quality, the lower 16 bits value is rearranged as a valid value during linking. Each format can be premixed for one instruction.

Examples:

```
lubarr:  dl  [10]     ;Reserves 10 4 byte size areas
lonum:   dl  13768    ;Sets the constant lonum with a
                       long word size integer
```

Related items:

DB, DW

Name:

ASCII.....ASCII text storing in memory

Format:

ASCII character expression {, character expression}*
character expression
= character string | character constant | byte constant

Functions:

This instruction is used to store the ASCII character code in memory.
For the area reserved by this instruction, the ASCII text assigned by the parameter must be stored in the memory. The character string for the parameter is decoded and stored in the memory sequentially from low-order addresses.
The area size becomes the number of bytes for the decoded parameter. The operand is a character string of one or more characters enclosed by double quotation marks.
The ASCII instruction stores the character code of each character of the character string in the memory, however, since the information showing the length and the termination of the character string is not output, the character strings may be set without a limitation.

Examples:

```
ascii "E0C88 Family"
ascii "bell",'\a'          ; bell and BELL code
ascii "bell\07"            ; Other format example
ascii "bell",'\07'         ; Other format example
ascii 62h,65h,6ch,6ch,07h  ; Other format example
```

Related item:

Table of ASCII character set

Name:

PARITY.....Setting/resetting of parity bit

Format:

PARITY <operand>

Functions:

The alphabet that has been adopted in the cross assembler asm88 is an ASCII character set. The ASCII character data are indicated with 7 bits and the most significant bit shows the parity. This bit can be optionally set or reset either always 0 or always 1 using the PARITY instruction. In addition, the total number for 1 bit can be made odd or even. The following parities can be specified for an <operand>.

PARITY 7	Sets the parity bit at 0 (default)
PARITY 8	Sets the parity bit at 1
PARITY ODD	It is set such that "1" within the 8 bits becomes odd
PARITY EVEN	It is set such that "1" within the 8 bits becomes even

Related item:

Table of ASCII character set

3.3 Symbol Definition Pseudo-Instructions

Symbol definition pseudo-instruction is the pseudo-instruction to define an expression with a name. The symbol definition pseudo-instructions are as follows:

EQU SET

Name:

EQU.....Name value setting

Format:

<name> EQU <expression>

Functions:

This instruction is used to define the <expression> with a <name>. The value of a name that has been defined by this instruction may not be changed later. Nor may an EXTERNAL declared symbol be placed on the right side of the equals sign.
Length of the expression is not restricted, but up to a 6 character hexadecimal number can be output to the assembly list. When a 7 or more character hexadecimal number has been defined, a warning is output.

In the sap88, the name defined by the EQU can be used in the conditional expression of the IFC statement that hereafter occurs, or it can be used as the parameter for the IFDEF/IFNDEF statements. [sap88 only]

Examples:

```
false  equ  0              ;Initialization
true   equ  -1
tablen equ  TABFIN-TABSTA  ;Calculation of table
                            length
nul    equ  00h            ;Defines a character
                            string indicating
                            ASCII characters
soh    equ  01h
stx    equ  02h
etx    equ  03h
eot    equ  04h
enq    equ  05h
```

Related items:

SET, IFC, IFDEF, IFNDEF, REPT

Limitation:

The <name> description must begin from the 1st column.

Name:

SET.....Name value setting

Format:

<name> SET <expression>

Functions:

This instruction is the same as the EQU instruction, it is intended, among others, to improve maintenance of the assembler source code and it serves to link <numeric expressions> with the <names>.
Unlike in the case of the EQU instruction, a name defined by the SET instruction can be redefined any number of times for other values and can be treated as an assembler variable. Among the the attributes of the cross-reference list, which is one of the output lists of the assembler, those are defined as variables take this symbol. The right side of the equals sign must be defined before this instruction.
The main object of this instruction is to use the name as a conditional assemble or macro variable and it serves as a valuable function in the structured preprocessor sap88. However, it does not have too much application in the cross assembler asm88 itself, other than functioning to permit the redefining of names.
Length of the expression is not restricted, but up to a 6 character hexadecimal number can be output to the assembly list. When a 7 or more character hexadecimal number has been defined, a warning is output.
In the sap88, the name defined by the SET can be used in the conditional expression of the IFC statement that hereafter occurs, or it can be used as the parameter for the IFDEF/IFNDEF statements. [sap88 only]

Examples:

```
abc   set  1
      ld   a,#abc
abc   set  2
      ld   a,#abc
```

Related items:

EQU, IFC, IFDEF, IFNDEF, REPT

Limitation:

The <name> description must begin from the 1st column.

3.4 Location Counter Control Pseudo-Instruction

The location counter control pseudo-instruction is as follows:

ORG

Name:

ORG.....Changing of location counter value

Format:

ORG <expression>

Functions:

This instruction is used to specify addresses where program has been placed. <expression> must be a relative value from a label within the current program section. At this time, an attempt to insert an absolute address into the program counter results as an error.
Length of the expression can be defined up to a 6 digit hexadecimal number, and an error occurs if 7 digits or more has been defined.

Examples:

```
sizstk    equu   200h        ;The stack size is 512 bytes
topstk:                      ;Reserves space for the
                              stack
          org   topstk+sizstk
```

Related items:

CODE, DATA

3.5 External Definition and External Reference Pseudo-Instructions

External definition and external reference pseudo-instructions are the pseudo-instructions to define and refer symbols which are commonly used between modules.

- External reference pseudo-instruction .....**EXTERNAL**
- External definition pseudo-instruction .....**PUBLIC**

Name:

EXTERNAL..Symbol external definition declaration

Format:

EXTERNAL <symbol> {,<symbol>}*

Functions:

EXTERNAL and PUBLIC instructions are used so that the same symbol will be used between multiple modules. Declaration must be done with an EXTERNAL instruction to reference symbols not defined within the self-module, but rather defined within other modules. If a declaration is made in EXTERNAL, it will simultaneously be made in PUBLIC as well.

Example:

```
external sqrt
carl     sqrt
```

Related item:

PUBLIC

Name:

PUBLIC.....Global declaration of symbol

Format:

PUBLIC <symbol> {,<symbol>}*

Functions:

When optional symbols are used in multiple modules, they are declared with the PUBLIC and EXTERNAL instructions. PUBLIC is used for declaration of symbols, such that there is a definition within the self-module that permits reference from other modules.

Example:

```
    public  sqrt ;SQRT permits reference from other
                  modules
sqrt:            ;Routine that computes the square
                  root of an integer
    .....
    etc.
```

Related item:

EXTERNAL

3.6 Source File Insertion Pseudo-Instruction [sap88 only]

Source file insertion pseudo-instruction is a pseudo-instruction to read and insert other files into the optional location of source file.

INCLUDE

* This instruction can only be used in the structured preprocessor sap88. The sap88 expands this instruction and creates the source file in which the specified file is inserted. In the cross assembler asm88, this instruction cannot be used and will cause an error if used.

Name:

INCLUDE.....Another file insertion

Format:

INCLUDE <file name>

Functions:

This instruction reads the specified file in the following an INCLUDE statement.
Including can be nested to optional depths. Another file can be further included into a file that is already included.

The sap88 analyses this pseudo-instruction and creates the output file in which the specified file is inserted. This pseudo-instruction is not transferred to the asm88 as is.

Examples:

```
include  chargen.s    ;Character generator
include  utilsub      ;General purpose
                       subroutine group
```

Limitation:

This instruction can only be used in the structured preprocessor sap88. In the cross assembler asm88, it cannot be used and will cause an error if used.

3.7 Assembly Termination Pseudo-Instruction

Assembly termination pseudo-instruction terminates each source program.

END

Name:

END.....Assembly stop

Format:

END {<Label>}

Functions:

This instruction is used to stop the assembly. A list for the portion following this instruction is output, but not assembled.

3.8 Macro-Related Pseudo-Instructions [sap88 only]

The following pseudo-instructions are related to the macro functions, and they perform a macro definition, a macro deletion, a repeat definition, and the like.

MACRO ~ ENDM
DEFINE
LOCAL
PURGE
UNDEF
IRP ~ ENDR
IRPC ~ ENDR
REPT ~ ENDR

* These pseudo-instructions can only be used in the structured preprocessor sap88. The sap88 outputs the source file in which the setting contents of these pseudo-instructions are expanded into a form that can be assembled by the cross assembler asm88. Further these macro-related pseudo-instructions cannot be accepted in the asm88 and will cause an error if used.

Name:

MACRO.....Macro definition

Format:

```
<macro name>   MACRO   [<parameter> [, <parameter>] * ]
               <statement string>
               [EXITM]
               <statement string>
[<macro name>] ENDM
```

Functions:

This instruction performs a macro definition. If the specified macro name is already used, the previous definition will be overridden and this current definition will redefine the macro. Names including any characters except blank characters, brackets "(" , ")", "{" , "}", "[" , "]" and a colon ":" can be used as macro names. It is not necessary to define the macro name for the ENDM line except the case that the macro definition was nested. Moreover, there is no limitation as to the number of parameters. Arguments delimited by a comma "," can be specified by the number of your choice at the time of a macro call. The number of arguments should not necessarily be equal to the number of parameters at the time of a macro definition. If a character string identical to one parameter exists in the macro body, it will be replaced with the corresponding argument character string at the time of a macro call. If any corresponding argument does not exit it will be replaced with a blank character string. It is also possible to specify a blank character string on arguments. In this case, specification should be done using the characters which are not included in the blank character string. For example, if it is specified as shown below at the time of a certain macro "xmac" call :

```
xmac   1,,2
```

The second argument will become a blank character string. At the same time, the number of argument at the time of the call will be replaced with the sap88 system parameters NARG and narg. The blank character string arguments at this time will also be counted.

All the parameters are not necessarily independent as tokens. Some will be replaced with arguments even when they occur inside character strings. In order to reduce substitution, it is advisable to use special symbols so that too much substitution can be evaded. All symbols except a comma "," and brackets "(" , ")", "{" , "}", "[" . "}" can be used for parameters and arguments.
For example :

```
sum  macro   c,d
     ld      a,[c]
     add     a,d
     ld      [c],a
     endm

     sum     total,#20
```

The above will be interpreted as follows :

```
     l#20    a,[total]
     a#20#20 a,#20
     l#20    [total],a
```

If you redefine your macro definition as shown below, your input will be correctly replaced :

```
sum  macro   c,&d
     ld      a,[c]
     add     a,&d
     ld      [c],a
     endm
```

The blank characters before and after parameters and arguments will be discarded. The blank characters inside parameters and arguments, however, are valid. Please take caution in this respect. A macro call from inside the body of the macro for a macro definition can also be done. In this case, a macro call should be initiated at the time the macro call generates.
For example :

```
 macamacro   x,y
     add     x,y
     endm

 macbmacro   x,y
     maca    x,y
     endm

     macb    a,#2    →    add  a,#2

 macamacro   x,y
     sub     x,y
     endm

     macb    a,#2    →    sub  a,#2
```

A macro call from the body of the macro can be executed according to the depth of your choice. However, if the call enters a loop, the macro call will be suspended. Take a simple example for instance:

```
add  macro   x,y
     ld      a,x
     add     a,y
     ld      x,a
     endm
```

When the macro defined as above is called, it is expanded as follows:

```
                              ld   a,b
     add     b,#2      →      add  a,#2
                              ld   b,a
```

"add a,y" in the third line will call itself. The macro call, therefore, will not occur. It will turn out to be a simple "add" instruction. If we take a look at a little more complicated example :

```
maca macro   x,y
     macb    x,y
     macc    x,y
     endm

macb macro   x,y
     macc    x,y
     maca    x,y
     endm

macc macro   x,y
     maca    x,y
     macb    x,y
     endm
```

$$\begin{tikzcd}
 & & & \mathtt{maca\ r0,\#2} \\
 & & \mathtt{macc\ r0,\#2} \arrow[ru] \arrow[rd] & \\
 & \mathtt{macb\ r0,\#2} \arrow[ru] \arrow[rd] & & \mathtt{macb\ r0,\#2} \\
 & & \mathtt{maca\ r0,\#2} & \\
\mathtt{maca\ r0,\#2} \arrow[ruu] \arrow[rdd] & & & \\
 & & \mathtt{maca\ r0,\#2} & \\
 & \mathtt{macc\ r0,\#2} \arrow[ru] \arrow[rd] & & \mathtt{macc\ r0,\#2} \\
 & & \mathtt{macb\ r0,\#2} \arrow[ru] \arrow[rd] & \\
 & & & \mathtt{maca\ r0,\#2}
\end{tikzcd}$$

When performing a conditional assembly using the IFC statement inside the body of the macro, the judgment will be made at the time of the macro call. If an EXITM line occurs at this time, the macro expansion will be suspended and the macro call will end at that moment.

For example :

```
xmac  macro x,y
      ...
      ifc   MODE == 2
            exitm
      endif
      ...
      endm
```

```
MODE  set   2
      xmac  #3,#4
```

When called as shown above, the macro expansion will end at the EXITM line.

```
MODE  set   1
      xmac  #3,#4
```

When called as shown above, the macro expansion will be executed to the last.

It is possible to include a macro definition in the body of the macro. In this case, however, the macro name of the MACRO line corresponding to the ENDM line will be required :

```
x     macro
      ...
      y     macro
            ...
            z     macro
                  ...
            z     endm
            ...
      y     endm
      ...
      endm
```

With the case shown above, the macro "y" definition will be executed at the time the macro "x" is called. In this case, however, it is not necessary to specify a macro name for the outermost macro definition ("x" in the above example) of the ENDM line. Nesting can be done to the depth of your choice.

Related items:

EQU, IFC, IFDEF, IFNDEF, IRP, IRPC, PURGE, SET

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

DEFINE.....Character-string macro definition

Format:

DEFINE <character-string macro name> [<substitute character-string>]

Functions:

This instruction performs a character-string macro definition. The token identical to the character-string macro name in the source after the DEFINE statement will be replaced with a macro instruction in the specified substitute character-string prior to the evaluation of all the statements except the IFDEF and IFNDEF statements. In the case that a substitute character-string is not specified, it will be replaced with a blank character-string. In addition, a character-string macro name will be subject to be evaluated in the IFDEF or IFNDEF statements.

Example:

```
define  XMAX    #128

        cp      a,XMAX
        ↓
        cp      a,#128
```

Related items:

IDEF, IFNDEF, UNDEF

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

LOCAL.....Definition of local label

Format:

```
LOCAL    [<local label name>
         [,<local label name>] * ]
```

Functions:

This instruction declares a local label. When a token with the name identical to that of a local label occurs inside a macro definition, it will be replaced in macros by a different label name, which will be automatically generated at each macro expansion. According to the rule of local label generation, the numerals in four digits starting with 0001 should follow the front character string "L". The front character string can be changed if specified at the start-up of the sap88.

Example:

```
macl  macro
      local x
      cp    a,#3
      jr    c,x
      ld    d,r0
x:
      endm

      macl
      macl
      ↓
      cp    a,#3
      jr    c,L001
      ld    d,a
L001:
      cp    a,#3
      jr    c,L002
      ld    d,a
L002:
```

Related item:

MACRO

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

PURGE.....Macro deletion

Format:

```
PURGE    [<macro name>]
```

Functions:

Once this instruction is executed, the macro definition of specified name that occur thereafter will be deleted. When name is not specified, all the macro definitions will be deleted. It is also possible to specify undefined macro name.

Example:

```
purge add       ;delete the macro add
add   ba,#10    ;use the add instruction
```

Related item:

MACRO

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

UNDEF.....Deletion of a character string macro

Format:

```
UNDEF    <character string macro name>
```

Functions:

The character-string macro definition will be deleted of the specified name that occur after this instruction is executed. It is also possible to specify undefined character-string macro name.

Example:

```
undef XMAX   ;delete the character string macro XMAX
```

Related items:

DEFINE, IFDEF, IFNDEF

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

IRP.....Repetition using character strings

Format:

```
IRP     <parameter>, <argument> [, <argument>] *
        <statement string>
ENDR
```

Functions:

With this instruction, arguments will be assigned to parameters in sequence from the left and expansion will be repeatedly performed up to the ENDR line by the times equal to the number of the arguments. If, at this time, a character string identical to the parameter exists between the IRP line and the ENDR line, such a character string will be replaced with the character string keyed by the argument.

All the parameters are not necessarily independent as tokens. Even when they occur inside character strings, they will be replaced with arguments. In order to reduce substitution, it is advisable to use special symbols for parameters so that too much substitution can be evaded. All except a comma "," and brackets " (" , ") ", " { " , " } ", " [" , "] " can be used as special symbols.
For example :

```
irp  w,10,20,30
     dw   w
endr
```

The above will be interpreted as :

```
     d10  10
     d20  20
     d30  30
```

If you modify the symbols as follows, your input will be correctly replaced:

```
irp  &w,10,20,30
     dw   &w
endr
```

The blank characters before or after parameters or arguments can be discarded. However, the blank characters located inside parameters and arguments are valid. Please take caution in this regard.

Each statement of IRP, IRPC and REPT can be nested to the depth of your choice. The ENDR line at this time will correspond to the inside IRP/IRPC/REPT lines.

Example:

```
irp     char,30,31,32,33,34,35,36,37,38,39
c_char: dw    charh
endr
        ↓
c_30:   dw    30h
c_31:   dw    31h
c_32:   dw    32h
c_33:   dw    33h
c_34:   dw    34h
c_35:   dw    35h
c_36:   dw    36h
c_37:   dw    37h
c_38:   dw    38h
c_39:   dw    39h
```

Related items:

IRPC, MACRO, REPT

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

IRPC.....Repetition by characters

Format:

```
IRPC  <parameter>, <argument character string>
      <statement string>
ENDR
```

Functions:

With this instruction, the characters of argument character strings will be assigned to parameters one by one in sequence from the left. The expansion will be repeatedly performed till the ENDR line by the times equal to the number of characters of arguments. If, at this time, the character strings identical to the parameters exist between the IRPC line and the ENDR line, such strings will be replaced with the characters keyed by the arguments.

All the parameters are not necessarily independent as tokens. Even when they occur inside character strings, they will be replaced with arguments. In order to reduce substitution, it is advisable to use special symbols so that excessive substitution can be prevented. All symbols except a comma "," and brackets "(" , ")", "{" , "}", "[" , "]" can be used as special symbols for parameters and arguments. For example :

```
irpc w,abc
     dw   'w'
endr
```

The above will be interpreted as :

```
     da   'a'
     db   'b'
     dc   'c'
```

If you modify the symbols as follows, your input will be correctly replaced :

```
irpc &w,abc
     dw   '&w'
endr
```

The blank characters before or after the parameters or arguments will be discarded. However, the blank characters inside the parameters and arguments are valid. Please take caution in this respect. Each statement of IRP, IRPC and REPT can be nested to the depth of your choice. The ENDR line at this time will correspond to the inside IRP/IRPC/REPT lines.

Example:

```
irp   char,Hello, world!
      dw    'char'
endr
      ↓
      dw    'H'
      dw    'e'
      dw    'l'
      dw    'l'
      dw    'o'
      dw    ','
      dw    ' '
      dw    'w'
      dw    'o'
      dw    'r'
      dw    'l'
      dw    'd'
      dw    '!'
```

Related items:

IRPC, MACRO, REPT

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

REPT .... Repetition by the specified number of times

Format:

```
REPT   <operation expression>
       <statement string>
ENDR
```

Functions:

The portion between the REPT line and the ENDR line will be repeatedly expanded by the number of times equal to the value of the operation expression. If there is any undefined name in the operation expression, the value of such a name will be evaluated as "0".

Each statement of IRP, IRPC and REPT can be nested to the depth of your choice. The ENDR line at this time will correspond to the inside IRP/IRPC/REPT lines.

Example:

```
rept  4             ;4-bit shift
      sll   a
endr
```

Related items:

EQU, IRP, IRPC, SET

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

3.9 Conditional Assembly Pseudo-Instructions [sap88 only]

The conditional assembly pseudo-instructions decide whether or not to perform the assembly within the specified range by the evaluation result of the conditional expression or whether the name has been defined or not. The conditional assembly pseudo-instructions are as follows:

IFC	**~**	**ENDIF**
IFDEF	**~**	**ENDIF**
IFNDEF	**~**	**ENDIF**

* These pseudo-instructions can only be used in the structured preprocessor sap88. The sap88 outputs the source file in which the statements subject for assembly are included. Further these conditional assembly pseudo-instructions cannot be accepted in the asm88 and will cause an error if used.

Name:

IFC ..... Conditional assembly by conditional expression

Format:

```
IFC     <conditional expression>
        <statement string> [
ELESEC
        <statement string> ]
ENDIF
```

Functions:

This instruction evaluates a conditional expression. If an expression is evaluated as "true", the statements following the IFC line will become a subject to be assembled until either an ELSEC line or an ENDIF line appears. If it is evaluated as "false", the statements following the IFC line will not be considered a subject to be assembled. In the case that there is an ELSEC line, the portion between the ELSEC and ENDIF lines will become a subject to be assembled if the conditional expression of the IFC line is "false". If it is "true", the ELSEC line through the ENDIF line will not become a subject for assembly.

Each statement of IFC, IFDEF and IFNDEF can be nested to the depth of your choice. The ELSEC line and the ENDIF line at this time will correspond to the inside IFC/IFDEF/IFNDEF lines.

As explained in the following, the conditional expression comes in three cases :

1) <operation expression>
When only an operation expression is used, a decision will be made as to whether the value of the expression is "0" or not "0". If it is "0", the value will be considered as "false". If it is not "0", the value will be considered as "true". In the case that there is any undefined name in the operation expression, the value of such a name will be evaluated as "0". For instance :

```
IFC  ee
```

will be decided as equivalent to

```
IFC  ee != 0
```

2) <operation expression> <relational operator> <operation expression>
The values of each operation expression are compared. If, at this time, there is any undefined name in the operation expressions, the value of the undefined name will be evaluated as "0".
The following relational operators are available :

==	"true" if the value of the left side is equal to that of the right side
!=	"true" if the value of the left side is not equal to that of the right side
<	"true" if the left side is smaller than the right side
>	"true" if the left side is larger than the right side
<=	"true" if the left side is smaller than, or equal to the right side
>=	"true" if the left side is larger than, or equal to around the right side

3) [<conditional expression>] <logical operator> <conditional expression>
A complex conditional expression can be expressed using a logical operator. The logical operation expressions include the following :

Unary operator:

!	"true" if the conditional expression is "false"

Binary operator:

&&	"true" if the left side is "true" and the right side is also "true"
||	"true" if the left side is "true" or the right side is "true"

The operators will be classified as follows from high to low precedence : either an operation expression or a conditional expression enclosed by a round bracket > a unary operator > an operator of an ordinary operation expression > a relational operator > &&> ||

The same operator precedence will take effect inside a round bracket. A unary operator is defined as a unary operator of an ordinary operation expression and "!" of a logical operator.

In addition, "character string" can be used as an operation expression.

When such character strings occurs on both sides of a relational operator, a character string will be compared to another character string. Otherwise, the value of the length of character strings will be compared.

Example:

```
table macro &1,&2
    ifc  narg == 1
         ifc! USE_DEFAULT || DEFAULT_SIZE<64
            &1:  db  0(64)
         elsec
            &1:  db  0(DEFAULT_SIZE)
         endif
      elsec
         &1: db  0(&2)
      endif
endm
```

Related items:

EQU, IFDEF, IFNDEF, SET

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

IFDEF .... Conditional assembly by the name either defined or undefined

Format:

```
IFDEF     <name>
          <statement string> [
ELSEC
          <statement string> ]
ENDIF
```

Functions:

If the name is defined by either the EQU statement or the SET statement, or is a character-string macro name which is defined by the DEFINE statement, the statements following the IFDEF line will become a subject to be assembled until either the ELSEC line or ENDIF line occurs. If the name is undefined, the statements following the IFDEF line will not become a subject to be assembled. In the case that there is an ELSEC line, the portion between the ELSEC line and the ENDIF line corresponding to the IFDEF line will become a subject to be assembled if the name of the IFDEF line is not defined. If the name is defined, the ELSEC line through the ENDIF line will not become a subject to be assembled.

Each statement of IFC, IFDEF and IFNDEF can be nested to the depth of your choice. The ELSEC line and the ENDIF line at this time corresponds to the inside IFC/IFDEF/IFNDEF lines.

Example:

```
ifdef EXTRA_MEMORY
stack_start equ   4000h
stack_size  equ   1000h
elsec
stack_start equ   3800h
stack_size  equ   800h
endif
```

Related items:

DEFINE, EQU, IF, IFNDEF, SET

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

Name:

IFNDEF ..... Conditional assembly by the name either undefined or defined

Format:

```
IFNDEF    <name>
          <statement string> [
ELSEC
          <statement string> ]
ENDIF
```

Functions:

If the name is not defined neither by the EQU statement or SET statement, nor defined by the DEFINE statement as a character-string macro name, the statements following the IFNDEF line will become a subject to be assembled until either the ELSEC line or the ENDIF line occurs. If the name is defined, the statements following the IFNDEF line will not be processed as a subject to be assembled. In addition, in the case that there is an ELSEC line, the portion between the ELSEC line and the ENDIF line corresponding to the IFNDEF line will become a subject to be assembled if the name of the IFNDEF line is defined. If not defined, the portion will not become a subject to be assembled.

Each statement of IFC, IFDEF and IFNDEF can be nested to the depth of your choice. The ELSEC line and the ENDIF line at that time will correspond to the inside IFC/IFDEF/IFNDEF lines.

Example:

```
ifndef       SMALL_MEMORY
stack_start equ   3800h
stack_size  equ   800h
elsec
stack_start equ   4000h
stack_size  equ   1000h
endif
```

Related items:

DEFINE, EQU, IF, IFNDEF, SET

Limitation:

This pseudo-instruction can only be used in the structured preprocessor sap88. It cannot be accepted in the asm88 and will cause an error if used.

3.10 Output List Control Pseudo-Instructions

The output list control pseudo-instructions are used for that can be easily referred, and are as following 7 types:

LINENO
SUBTITLE
SKIP
NOSKIP
LIST
NOLIST
EJECT

Name:

LINENO ... Change of line number for assembly list file

Format:

LINENO <numeric expression>

Functions:

This instruction forcibly changes the line number for the assembly list file to the following line number set by the <numeric expression>. The line number can be changed up to 65535, and starts from 0 if it exceeds the upper limit.

Example:

```
lineno    99      ;line number begins from 100
```

Name:

SUBTITLE .... Subtitle setting to assembly list file

Format:

SUBTITLE <character string>

Functions:

The SUBTITLE instruction is used for outputting optional character string as subtitles onto the 4th line of the list output. After the first page, SUBTITLE appearing within the current page is used as the subtitle of the following page and continue to be used until a new SUBTITLE appears.
The character string should be enclosed by double quotation marks.

Example:

```
subtitle "asm88 Special function library"
```

Name:

SKIP .... Suppresses all initialization codes output that exceed 4 bytes to assembly list file

Format:

SKIP

Functions:

When this instruction appears, even when there is an initialization that exceeds a one line assembly list file, that is, a size greater than 5 bytes in each of the following instructions ASCII, DB, DL and DW, it will output a 1 line code only to the assembly list file and will suppress code outputs that do not fit on the assembly list file. The NOSKIP instruction serves to counter this function, however, SKIP is set in the default.

Example:

```
noskip
      db    1,2,3,4,5,6,7,8,9,0
            ;All the hexadecimal codes output to the
             assembly list file
skip
      ascii "1234567890"
            ;ASCII codes output to list file as one
             line only
```

Related item:

NOSKIP

Name:

NOSKIP .... Outputs all initialization codes to assembly list file

Format:

NOSKIP

Functions:

This instruction is used to reverse the function of the SKIP instruction (default) that suppresses output of codes exceeding 4 bytes to the assembly list file. When this instruction appears, thereafter, if initialization codes are set for each of the ASCII, DB, DL and DW instructions, all of these codes will be output onto the list.

Example:

```
noskip
       db      1,2,3,4,5,6,7,8,9,0
               ;All the hexadecimal codes output to
                the assembly list file
skip
       ascii   "1234567890"
               ;ASCII codes output to list file as
                one line only
```

Related item:

SKIP

Name:

NOLIST ... Prohibition of assembly list file output

Format:

NOLIST

Functions:

When this instruction appears, thereafter, the assembly list file output will be prohibited. In order to resume the assembly list file output, use the LIST instruction. Further the line number is updated if the assembly list file output has been prohibited by NOLIST.

Related item:

LIST

Name:

LIST.....Assembly list file output

Format:

LIST

Functions:

When this instruction appears, thereafter, the assembly list file will be output. In the default, LIST is set.

Related item:

NOLIST

Name:

EJECT.....Form feed of assembly list file

Format:

EJECT

Functions:

When this instruction appears, the form feed with the page header is inserted to the assembly list file same as an auto form feed. This instruction itself is shown in the first line of the page after form feeding.

III REFERENCE

Part III describes the start-up format, all the start-up flags that can be used, error messages and so on as reference for each software tool.

Contents

FORMAT OF EXPLANATION

The explanation for each software tool has been arranged by the items shown below.

PROGRAM NAME

Shows the program name.

SUMMARY

Functions of the software tool are explained.

INPUT/OUTPUT FILES

Shows the execution flow and input/output files.

START-UP FORMAT

Shows the start-up command format of the software tool. This format includes the main component elements of the command line; the name of tool itself and all the flags that can be received in the tool. The command cannot be started up if you input invalid flags and/or arguments and forget the necessary arguments.

Flags are listed in [ ] by a delimiter "-" and the names. In principle, the flags are listed in alphabetical order. Flags that are composed of values alone, are listed behind all other flags. In the case of flags that accompany some values, the type of concerned value as well is shown by one of the below codes (assigned immediately following the flag name).

Code	Types of value
*	Character string
#	Integer (word size)
##	Integer (long word size)
?	Single character

The hash mark # shows word size (2-byte) integers. Double hash marks ## show long word size (4-byte) integers. When integers begin with 0x or 0X they may be interpreted as hexadecimal numbers. When they begin with O as octal numbers and in other cases as decimal numbers, they can optionally be preceded by either plus + or minus - signs. A caret "^" immediately follows the value code, of formats of the type where there are two or more assignments per flag such that the values are stacked.
For example, the asm88 utility format is as follows:

```
asm88 -[all c l o* q RAM# ROM#
          sig# suf* x] [drive:]<files>↵
```

We know that the asm88 receives the following 10 different sorts of flags.
That means, a word size integer value is assigned to the flags -RAM, -ROM and -sig. The flags -all, -c, -l, -q and -x do not have values. Character strings are assigned to -o and -suf.
Be careful of flags which normally have a hyphen placed immediately in front, appearing without one. (Provided there is no particular specification and a hyphen is assumed.)
When specifying the flag individually, RAM# in the list shown above should be assigned as -RAM#.
Furthermore, flags without values can continuously be specified by placing a "-" (hyphen) only for the head of the flags to be specified, for example, -clq.
The location and meaning of a non-flag argument is indicated by a word within < and > (<files> in the above example). Each meta-concept shows 0 or 1 or more arguments on the command line. When inputting command lines, type all the command line where meta-concepts appear in their position on the concerned line. In the case of the asm88, input one or more file names in the position shown by <files>. Meta-concepts in brackets are optional specifications. It is all right if they appear, and they may appear more than once.

FLAGS

Functions of all flags are listed. In some cases, supplementary explanations follow them depending on the situation.

ERROR MESSAGES

A list of error messages displayed during execution.

RETURN VALUE

When execution has been completed, each tool returns either of two values, "success" or "failure". This item describes the conditions under which either of the two are returned by the tool. Generally, the return value of "success" indicates that the tool executed all the necessary file processing.
This return value is used to evaluate an execution result of the tool when executing batch processing.

EXAMPLE

Here is an example using the software tool.

NOTE

Here notes for use are described.

1 Structured Preprocessor <sap88>

PROGRAM NAME

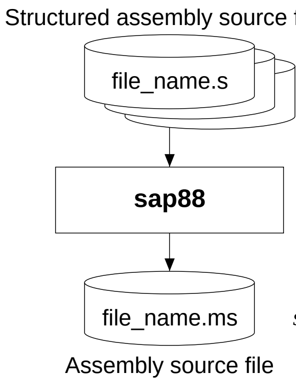
sap88.exe

SUMMARY

The structured preprocessor sap88 adds the macro functions to the cross assembler asm88.
The sap88 expands the macro and structured control statements included in the specified E0C88 assembly source file into a format that can be assembled by the asm88, and outputs it. At this time, the sap88 also executes the processing for including of the modularized E0C88 assembly source files and conditional assembly.
When file name has not been specified, the sap88 reads from the standard input (console).

INPUT/OUTPUT FILE

• Execution flow



sap88 execution flow

• Input file

Structured assembly source file: file_name.s
This is a structured assembly source file which is created by an editor such as EDLIN.

• Output file

Assembly source file: file_name.ms
This is the output file in which the macros in the structured assembly source file are expanded into the E0C88 instructions that can be assembled by the asm88. This file becomes an input file of the asm88. The output file extension should be made as ".ms".

START-UP FORMAT

```
sap88 -[d*^ l* o* q] [drive:] <file>↵
```

flags:
Character strings enclosed with [ ] mean flags. Explanations for each flag are discussed later.

drive:
In case the input file is not in current drive, input the drive name in front of the input file name. It can be omitted if the input file is in current drive.

file:
Specify the file name to be input to the sap88. This file name can be input using either capital letters or small letters. When <file> has not been specified, the sap88 reads from the standard input.

Note: The extension for the structured assembly source file should be made as ".s".

FLAGS

The sap88 can accept the following flags. The flags should be input with small letters.

Function	Flag	Explanation
Character-string macro definition	`-d*^`	A character-string macro is defined prior to reading in an input file. "*" has the following format: <character-string macro name> = <substitution character string> If the substitution character string is not defined and only the <character-string macro name> is defined, only the character-string macro will be defined and the substitution character string will become a blank character string. The character-string macros using the -d flag can be defined up to a maximum of 20.
Front character string specification	`-l*`	The front character string of a label name that is created at the time of the expansion of the structured control statement is designated. It is "L" in default.
Creating output file	`-o*`	An output file name is turned to *. The default status is standard output.
Suppression of start-up message	`-q`	Does not output any message related to processing of the structured preprocessor.

ERROR MESSAGES

Error message	Description
unexpected EOF in ~	The file is terminated in the middle of ~.
can't include ~	~ cannot be included.
illegal ~	~ is incorrect.
illegal define	"define" statement is incorrect.
illegal expression at ~	~ in the expression is incorrect.
illegal undef	"undef" statement is incorrect.

RETURN VALUE

The sap88 returns "success" if there is no syntax error in the input file. If there is a syntax error, "failure" is returned even if the contents of the input file are correct.

EXAMPLE

Expands the structured assembly source file "sample.s" to the assembly source file "sample.ms".

```
C>sap88 -o sample.ms sample.s↵
```

NOTE

If there is no syntax error in a macro statement, the sap88 expands it normally even though it contains illegal operands such as wrong register names. This error will be detected by the assembler asm88.

2 Cross Assembler <asm88>

PROGRAM NAME

asm88.exe

SUMMARY

The cross assembler asm88 converts an assembly source file to machine language by assembling the assembly source file in which the macros are expanded by the structured preprocessor sap88. The asm88 is a high speed assembler whose functions have been simplified to increase speed, and all the added functions, such as macro and conditional assembly, are supplemented with another utility (sap88).
The asm88 deals with the relocatable assembly for modular development.
In the relocatable assembly, the relocatable object file to link up with the other modules using the linker link88 is created.
In addition, the asm88 can directly input an assembly source file and in such case, the source program can be described in free format as the following format.

Label: Mnemonic Operand ;Comment

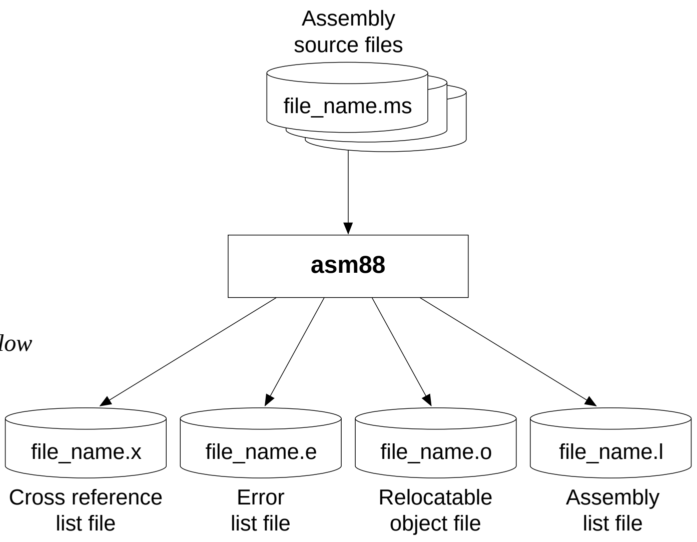
In the above format, ":" indicates the end of the label and ";" indicates the beginning of the comment.
It is possible to format freely by using these separators.

The asm88 also outputs three types of lists for the programmer, an assembly list, an error list and a cross-reference list. The assembly list is composed of a line number, address and a machine code corresponding to each source statement. The line number is output as a decimal number and the address and machine code as a hexadecimal number. When errors occur during assembly, an error list file is created that is composed of a file name, the line number that generated the error, the error level and an English error message.
Also in the assembly list file, a mark "*" is placed at the line number in which an error has been generated. It has also been designed such that the relationship between the definitions and the references of the symbols within the files can be easily understood by a cross-reference list. Since these are created as individual files, file management has also been simplified. Processing can continue even when an error occurs, provided it is not a fatal error.

INPUT/OUTPUT FILES

• Execution flow

The asm88 inputs assembly source files and outputs relocatable object files, an assembly list file, a cross reference list file and an error list file after assembly.



asm88 execution flow

• Input file

Assembly source file: file_name.ms
This is an assembly source file created by the sap88. In the default of the asm88, ".ms" is set as the input file extension. Although the extension can be changed by specifying an option, do not change the default setting if unnecessary.

• Output files

1. ***Relocatable object file: file_name.o***
   This is the file output from the asm88 after converting the assembly source file to the relocatable E0C88 machine language by the relocatable assembly. This file becomes an input file for the linker link88.
2. ***Assembly list file: file_name.l***
   This is the file in which the machine language converted by assembly and the address are output as a list corresponding to each source statement. The addresses are output as relative addresses that the head of the CODE section or the DATA section in the file assume as "000000H". The creating of this file can be prohibited by a start-up flag.
3. ***Cross reference list file: file_name.x***
   This is a list of addresses in which a symbol has been defined and referred. Creating this file can be prohibited by a start-up flag.
4. ***Error list file: file_name.e***
   This is a list of errors that have been generated during assembly.

START-UP FORMAT

```
asm88 -[all c l o* q RAM# ROM# sig# suf* x] [drive:] <files>↵
```

flags:
Character strings enclosed with [ ] mean flags. Explanations for each flag are discussed later.

drive:
In case the input file is not in current drive, input the drive name in front of the input file name. It can be omitted if the input file is in current drive.

files:
Specify the file name to be input to the asm88. This file name can be input using either capital letters or small letters, and specifying two or more source files is possible. An error will occur when <files> are not specified.

Note: *Up to eight characters are available for the source file name. Furthermore, the extension ".ms" must be input.*

FLAGS

The asm88 can accept the following flags.
-ROM# and -RAM# should be input using capital letters and the others should be input using small letters.

Function	Flag	Explanation
All symbols output	`-all`	Outputs all symbols including local symbols to a symbol table. In default, only global symbols and undefined symbols are output.
Differentiation between capital and small letters within source program	`-c`	Differentiates capital and small letters within the input source. Since capital and small letters are not differentiated in default, ABC and abc are handled as the same symbol. When this flag is specified, the CPU instructions and the register names must be described using small letters.
Prohibition of assembly list generation	`-l`	Prohibits the creation of an assembly list file. In default, an assembly list file with the extension ".l" is created.
Creating output file	`-o*`	Creates output files with the name "*". In default, the output file name is the same as the input file and the extension becomes ".o" when the input file extension is ".ms". When the input file extension is other than ".ms", the default output file name becomes "xeq". Example: When creating "out.o" from "sample.ms", specify as below. `asm88 -o out.o sample.ms↵`

Function	Flag	Explanation
Suppression of start-up message	`-q`	Does not output any messages related to the assembly processing.
RAM capacity setting	`-RAM#`	Sets the RAM capacity in byte units with #. When the total size of the DATA section exceeds the value set by this flag, an error is output. Example: When the internal RAM capacity is set in 2K (2048 bytes), specify as below. `asm88 -RAM 2048 sample.ms↵`
ROM capacity setting	`-ROM#`	Sets the ROM capacity in byte units with #. When the total size of the CODE section exceeds the value set by this flag, an error is output. Example: When the internal ROM capacity is set in 16K (16384 bytes), specify as below. `asm88 -ROM 16384 sample.ms↵`
Setting character numbers of symbols	`-sig#`	Character numbers of symbols that are significant can be set with a # value. In default the # is set to 15 characters.
Change of input file extension	`-suf*`	Changes the extension of the input file to * (a separator "." is not included). The default is ".ms". Example: When the extension of an input source file (sample.ms) is changed to ".bs", specify as below. `asm88 -suf bs sample.bs↵`
Prohibition of cross reference list file creation	`-x`	Prohibits the creation of a cross reference list file. In default, a cross reference list file with the extension ".x" is created.

When one or more <files> without the -o flag are specified and the file name extension of the input file name is the suffix of the default file name, the asm88 outputs the object files with the same name as the input files and the extention ".o".

```
asm88 file1.ms file2.ms files3.ms
```

By inputting the above, the three object files file1.o, file2.o and file3.o are automatically created. Be aware that the -o flag will not function, when multiple files have been specified for <files>.

ERROR MESSAGE

• Fatal errors

Error message	Description
can't create <file>	<file> cannot be created.
can't open <file>	<file> cannot be opened.
can't read tmp file	Temporary file cannot be read.
can't write tmp file	Temporary file cannot be written.
namelist full	Name list table is full.
no i/p file	There is no input file specification.
insufficient memory	There is not enough memory.
can't seek on vmem file	Seeking of virtual memory file has failed.
can't seek to end of vmem file	Cannot reach the end of virtual memory file.
no swoppable page	There is no swap space.
read error on vmem file	Reading of virtual memory file has failed.
write error on vmem file	Writing to virtual memory file has failed.

• Severe errors

Error message	Description
<numeric label> already defined	The numeric label has been defined previously.
<identifier> wrong type	An illegal identifier has appeared.
<token> expected	A token is needed.
' missing	A quotation mark is missing.
attempted division by zero	Attempt has been made to divide by zero.
attempt to redefine <identifier>	Attempt has been made to redefine an identifier.
constant expected	A constant expression is required.
end expected	There is no end instruction.
encountered too early end of line	The line has terminated in the middle.
field overflow	The field to be secured has overflowed.
invalid branch address	An external defined symbol is used for the operand of the short branch instruction.
invalid byte relocation	The byte relocation is invalid.
invalid character	Three is an illegal character.
invalid flag	The flag is invalid.
invalid operand	The operand is invalid.
invalid relocation item	The relocation item is invalid.
invalid register	The register is invalid.
invalid register pair	The register combination is invalid.
invalid symbol define	The symbol definition is invalid.
invalid word relocation	The word relocation is invalid.
new origin incompatible with current psect	There is an absolute origin within the relocatable section (relocatable mode).
non terminated string	The termination of a string cannot be located.
<identifier> not defined	Undefined identifier has appeared.
missing numeric expression	A numeric expression is missing.
cars or jrs out of range	Branch destination by cars or jrs is out of range.
carl or jrl out of range	Branch destination by carl or jrl is out of range.
operand expected	There is no operand.
psect name required	A section name must be specified.
phase error <identifier>	The label address is different between pass 1 and pass 2.
CODE or DATA missing	There is no section setting pseudo-instruction.
ROM capacity overflow	ROM capacity has overflowed.
RAM capacity overflow	RAM capacity has overflowed.
relocation error in expression	A relocation error has appeared within the expression.
<identifier> reserved word	<identifier> is a reserved word.
syntax error <token> expected	Syntax error due to insufficient token(s)
syntax error <token> unexpected	Syntax error due to excess token(s)
syntax error - invalid identifier <identifier>	Syntax error due to an illegal identifier
syntax error <token> invalid in expression	Syntax error due to an illegal token
system error < > <token>	System error due to an illegal token
unsupported instruction	Unsupported instruction has appeared.
unsupported operand	Unsupported operand has appeared.

• Warning errors

Error message	Description
directive is ignored in relocatable mode	The pseudo-instruction is skipped because it is in the relocatable mode.
possibly missing relocatability	Relocatability may lose.
constant overflow	Seven or more digits has been defined for the name.
expected operator	There is no operator (BOC, LOC, POD, LOD).

RETURN VALUE

When there is no syntax error within the input file nor pass 2 error, and all the processing is successfully completed, the asm88 returns "success".

EXAMPLE

Performs relocatable assembly of the file "sample.ms" to simultaneously obtain the list file "sample.l".

```
      C>asm88 sample.ms↵
```

3 Linker <link88>

PROGRAM NAME

link88.exe

SUMMARY

The link88 links multi-section relocatable object files for the E0C88 and creates an absolute object file. The absolute object file is used to create a program data HEX file that is used for debugging with the ICE88 or EVA88 by inputting to the binary/HEX converter hex88. It will also be used to create absolute symbol information (rel88) after linking the relocatable assembled file.

The basic functions of the link88 process are as follows.

1) The global flag controls the overall link88 process.
2) It defines the new CODE section and DATA section by the addition of a flag and a file.
3) It relocates sections, rearranging them in optional locations of the physical memory and permits them to be mutually "stacked" (chaining) in appropriate storage boundaries.
4) Each object file input affects the current CODE section and DATA section.
5) The final output starts with the header, thereafter (in the named order) all CODE sections, all DATA sections, symbolic table and the relocation stream for all CODE sections and all DATA sections. The respective component elements for these sorts of outputs are controlled through use of the appropriate global flag which will be described later.
6) Since all the sections are continuous in the linker output, the binary/HEX converter hex88 must be used for writing the section into the appropriate physical location, in order to execute it in a special location within the memory.

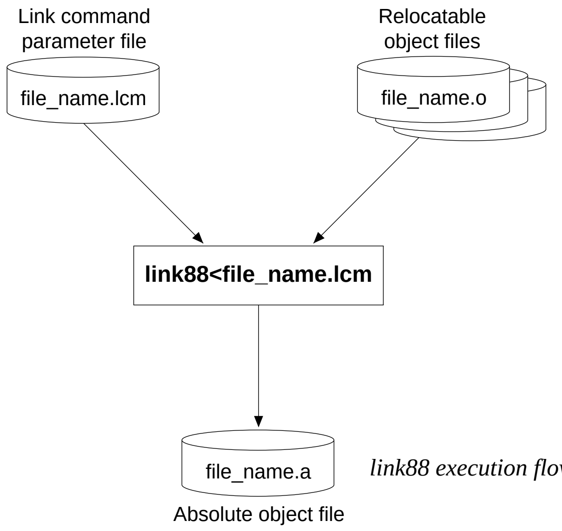
The E0C88 has a 24-bit wide address space (maximum 16M bytes). It splits that address space into a 32K-byte bank (code section) or a 64K-byte page (data section) by controlling the most significant 8-bit by registers such as the code bank register (CB) and the expanded page registers (EP, XP and YP) in an effort to expand the access performance within that range. It is possible to access an optional bank or page from an optional bank or page by rewriting the content of the register, thus permitting easy management of such things as large programs and data bases. However, since the register will not be automatically renewed, even if the bank and the register are crossed, a load module image permitting the 16M-byte address space to be described linearly cannot be created.

The E0C88 adopts a multi-segment system for linking relocatable objects, in order to create load images to be laid out in the optional physical addresses of the address spaces managed by it.

This is a technique in which "All the spaces are split into optional sections of 64K-byte (page) or 32K-byte (bank) units and the address information necessary for the memory layout determines all the address information in accordance with the assignment to each segment unit."
In this technique, since the creation of continuous data objects whose size exceeds 64K bytes (page) and 32K bytes (bank) for one section is not permitted, a limitation is imposed whereby the total size for the CODE sections included in the modules of assembly units cannot exceed 32K bytes and the total size for the DATA section cannot exceed 64K bytes. This restriction reflects the address restriction of the CPU itself and even if a diagnosis of a data overflow generated during assembly were overlooked, it is set up such that it would be rediagnosed during linking.
However, it outputs an error when the size exceeds 64K bytes in default, but does not output when the size exceeds 32K bytes. Consequently, a flag must be specified for judgment when the size exceeds 32K bytes.

INPUT/OUTPUT FILES

• Execution flow



link88 execution flow

• Input files

1. ***Relocatable object file: file_name.o***
   This is a relocatable file in machine language that is output through relocatable assembly with the cross assembler asm88.
2. ***Link command parameter file: file_name.lcm***
   This is a link command parameter file that is directly described by the user.

• Output file

Absolute object file: file_name.a
This is a multi-section object file created by the link88.

Note: Multi-section object file is an absolute object image whose format is composed of a global header, a section descriptor, objects within all CODE sections, objects within all DATA sections, objects within all DEBUG sections, objects within all ZPAG section, a symbolic table, a debug symbolic table, and all relocation information.

START-UP FORMAT

`link88 -[c cd +dead max## o* q]` **<sections>**

<sections> includes one or more following contents.

`-[+code +data m## p##] [drive:]`↵

flags:
Character string enclosed with [ ] mean flags. Flags within the first [ ] are global flags and flags within the [ ] included in <sections> are local flags.

drive:
In case of the relocatable object files or the libraries are not in current drive, input the drive name in front of these file names. It can be omitted if these files are in current drive.

Note: The extension for the relocatable object files should be made as ".o".

FLAGS

The link88 can accept the following flags. The flags should be input with small letters.

• Global flags

Function	Flag	Explanation
Distinction between capital and small letters within symbols	`-c`	Distinguishes capital and small letters used for symbols within the relocatable object file. In default, they are not distinguished, therefore ABC and abc are handled as the same symbol.
Deletion of DATA code part	`-cd`	Does not output the code part for the DATA section. -cd is used to create modules that define only symbol values for such purposes as specification of the addresses for the common library.
Listing of undefined symbols	`+dead`	Outputs a list of dead wood symbols on the CRT, that is, symbols that have been defined, but are not referred as absolute.
Setting of maximum section size	`-max##`	Sets the maximum section size at ## bytes. The default value is FFFFFFH (16M bytes). This value is used when sections are linked. When it exceeds this value, an error will occur.
Setting of output file name	`-o*`	Writes the output module on the file *. The default output file name is xeq.
Skip start-up message	`-q`	Does not output any message related to link processing.

When the arguments on the command line are not transferred to the link88, the list of flags and files that become arguments of the link88 are transferred from standard input. When a "-" (hyphen) first appears in the argument list of the command line, a standard input is incorporated into the argument list in place of the "-". The occurrences of "-" following thereafter are disregarded.
The specified <files> are linked in that order.

• Local flags

Flags for sections

Function	Flag	Explanation
Beginning CODE section	`+code`	Begins a new CODE section, then processes the local flag for that section.
Beginning DATA section	`+data`	Begins a new DATA section, then processes the local flag for that section.

A new section of a specified format is not actually created, when the final section of that format has a zero size. However, a new local flag is processed and overwrites the preceding value. These two flags must immediately precede the local flag set to appropriately process the flags and to decide to what flag is to be applied.

Flags used only together with +code or +data

Function	Flag	Explanation
Setting of individual section size	`-m##`	Sets the maximum size of the individual segment as ## bytes. The default size is 8000H (CODE section) or 10000H (DATA section). An error will occur if the section size exceeds this setting value.
Physical address setting	`-p##`	Sets the physical address of the beginning of the section as ##.

ERROR MESSAGES

Error message	Description
bad file format: 'FILE NAME'	Format of the input file 'FILE NAME' is incorrect.
bad relocation item	There is long integer type relocation information.
bad symbol number: 'NUMBER'	'NUMBER' is detected as illegal symbol code.
can't create 'FILE NAME'	The file 'FILE NAME' cannot be created.
can't create tmp file	Temporary file cannot be created.
can't open: 'FILE NAME'	The input file 'FILE NAME' cannot be opened.
can't read binary header: 'FILE NAME'	Header of the file 'FILE NAME' cannot be read.
can't read file header: 'FILE NAME'	First two bytes of the file 'FILE NAME' cannot be read.
can't read symbol table: 'FILE NAME'	Symbol table cannot be read from the file 'FILE NAME'.
can't read tmp file	Temporary file cannot be read.
can't write output file	Cannot write into output file.
can't write tmp file	Cannot write into temporary file.
field overflow	Branch destination by cars or jrs is out of range.
inquiry phase error: 'SYMBOL NAME'	Symbol value of the 'SYMBOL NAME' is different between pass 1 and pass 2.
link: early EOF in pass2	Unexpected EOF is detected during pass 2 processing.
multiply defined 'SYMBOL NAME'	'SYMBOL NAME' is multiply defined.
no object files	No input object files exist.
no relocation bits: 'FILE NAME'	The relocation information corresponding to the file 'FILE NAME' is suppressed.
'SECTION NAME' overflow	The section size in the 'SECTION NAME' exceeds the upper limit value.
phase error: 'SYMBOL NAME'	Symbol value of the 'SYMBOL NAME' is different between pass 1 and pass 2.
previous reference blocked: 'SYMBOL NAME' range error	The information related relocation bit width is unmatched.
read error in pass2	Read error is generated during pass 2 processing.
undefined 'SYMBOL NAME'	'SYMBOL NAME' has not been defined.

RETURN VALUE

When an error message is not output to the standard output, in other words, no undefined symbol remains and all reads and writes have succeeded, the link88 returns "success". If not, it returns "failure".

EXAMPLE

Links the sample.o by the link88 via standard input.

```
A>link88↵
-o c88xxx.a +code -p0x100 +data -p0x8000↵
sample.o↵
^Z↵
A>
A>link88 < sample.lcm↵
```

4 Symbol Information Generator <rel88>

PROGRAM NAME

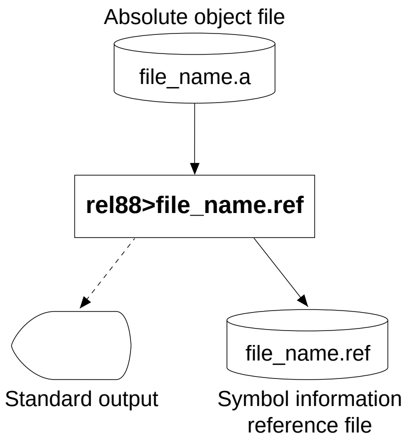
rel88.exe

SUMMARY

The rel88 checks the multi-section relocatable objects. The files that become the object of such checks are relocatable object files output by the cross assembler asm88 and absolute object files output by the link88. The rel88 can be used to check the size and configuration of relocatable object files and to output symbol information in absolute object files output from the link88.

INPUT/OUTPUT FILES

• Execution flow



rel88 execution flow

• Input file

Absolute object file: file_name.a
Inputs an absolute object file created by the link88.

• Output file

Standard output or
Symbol information reference file: file_name.ref
The rel88 outputs a symbol information reference file that is allocated in the physical address from the absolute object file.

START-UP FORMAT

```
rel88 -[a +dec d g +in +sec v] [drive:] <files>↵
```

flags:
Character strings enclosed with [ ] mean flags. Explanations for each flag are discussed later.

drive:
In case an input file is not in current drive, input the drive name in front of the input file name. It can be omitted if an input file is in current drive.

files:
Specify the file name to be input into the rel88. This file name can be input using either capital or small letters and specifying two or more files is possible. An error will occur when <files> is not specified.

FLAGS

The rel88 can accept the following flags. The flags should be input with small letters.

Function	Flag	Explanation
Sorting of symbol names	`-a`	Sorts outputs in alphabetical order of the symbol names.
Decimal output	`+dec`	Outputs symbol values and segment sizes in decimal numbers. The default is a hexadecimal number.
Output of defined symbols	`-d`	Outputs all defined symbols within each file, one per line. The symbol value, the "relocation code" showing to what the value is related and the symbol name are entered on each line. Values are output in the number of digits needed to indicate the integers in the E0C88. The meanings of the relocation codes in the outputs are as follows. • C indicates CODE relativity • D indicates DATA relativity • A indicates absolute (not relocatable) • ? indicates rel88 cannot recognize it. Small letters are used to indicate local symbols. Capital letters are used for global symbols.
Output only global symbols	`-g`	Outputs global symbols only.
Standard input	`+in`	Takes <files> from standard input and adds them to command line. Redirecting is also possible and is valid when many files are specified.
Physical address and size of multi-section	`+sec`	Outputs the physical address and size of each section of multi-segment output files.
Sorting by symbol values	`-v`	Sorts the inside of section by symbol values. The aforementioned -d flag is tacitly specified. Symbols that have the same value are sorted in alphabetic order. Absolute (non-relocatable) symbols are displayed first and are followed by CODE relative symbols and DATA relative symbols.

<files> are zero, or one or more files and they must have a multi-section format. When two or more files are specified, the name of each file or module precedes the information that is output pertaining to it. Each name is followed by a colon and a new-line. When there is no <files> specification, or when a "-" appears on the command line, xeq is used as an input file.

ERROR MESSAGE

Error message	Description
can't read binary header	Reading of the object header excluding magic number and configuration byte has failed.
can't read header	Reading of the first two bytes of the object header (magic number and configuration byte) has failed.
can't read symbol table	Reading of the symbolic table in the object has failed.

RETURN VALUE

When a diagnostic message has not been created (in other words, when all the reads have succeeded and all the file formats are valid), rel88 returns "success".

EXAMPLE

Obtains a list of all the symbols within the module in alphabetic order in hexadecimal numbers.

```
C>rel88 -a alloc.o↵
0x0074C _alloc
0x0000D _exit
0x01feC _free
0x00beC _nalloc
0x0000D _sbreak
0x0000D _write
```

NOTE

When no symbol is in the object or local symbols only exist, rel88 outputs a "no memory" message. However, the local symbols are registered in the symbolic table by setting the -all flag of the asm88 (all symbols output). If you wish to refer to all symbols, set the -all flag of the asm88.

5 Symbolic Table File Generator <sym88>

PROGRAM NAME

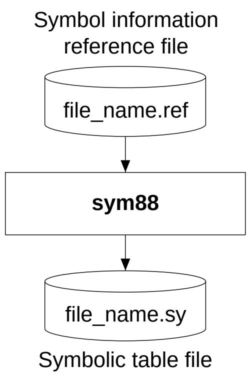
sym88.exe

SUMMARY

The symbolic table file generator sym88 converts a symbolic information file (file_name.ref) generated in file redirect with the symbol information generating utility rel88 to a symbolic table file (file_name.sy) that can be referenced in the ICE88. Loading the symbolic table file and the corresponding relocatable assembly program file in the ICE88 makes symbolic debugging possible.

INPUT/OUTPUT FILE

• Execution flow



sym88 execution flow

• Input file

Symbol information reference file: file_name.ref
Inputs a symbol information reference file created by the rel88.

• Output file

Symbolic table file: file_name.sy
The sym88 converts a symbol information file into a format that can be loaded to the ICE88 and outputs a symbolic table file.

START-UP FORMAT

`sym88` **<file>**↵

file:

Specify the symbol information file (.ref) to be input to the sym88.
This file name can be input using either capital letters or small letters.
An error will occur when <file> is not specified.

ERROR MESSAGE

Error message	Description
No Input File	Input file ".ref" has not been specified.

RETURN VALUE

The sym88 returns "success" if there is no error in the input file and an output file is created. If there is an error in the input file or internal created file, "failure" is returned.

EXAMPLE

Converts the symbol information reference file sample.ref into the symbolic table file sample.sy.

```
    A:/>rel88 sample.ref↵
```

NOTES

1. Drives and directories for input files can not be specified in the startup command of the sym88. Therefore, be sure to start up the sym88 after setting the directory of the input file as the current directory.
2. The sym88 does not check the format of the input file. Therefore, the symbol information file to be input to the sym88 must only be generated using the symbol information generating utility rel88 with the flags shown below.
   ```
   A:/>rel88 -v +sec sample.a>sample.ref
   ```

6 Binary/HEX Converter <hex88>

PROGRAM NAME

hex88.exe

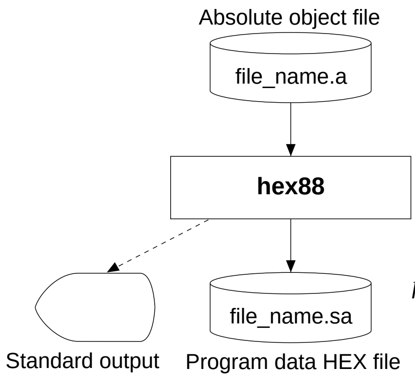
SUMMARY

The hex88 converts an absolute object file created by the link88 into a hexadecimal data conversion format (program data HEX file). This system adopted Motorola S record format. An absolute object file is read from the <ifile>. When an <ifile> is not assigned, or when an assigned file name is a "-" (hyphen), file xeq is read.
Further, S2 format in Motorola S record (can convert up to 3-byte address) is used since the E0C88 has a maximum 16M-byte address space (000000–FFFFFFH).

INPUT/OUTPUT FILES

• Execution flow

The hex88 is a tool to convert an absolute object file output from the linker (link88) into a program data HEX file in hexadecimal format. The execution flow is shown below.



hex88 execution flow

• Input file

Absolute object file: file_name.a
File to be input into the hex88 is an absolute object file output from linker.

• Output file

Standard output
or Program data HEX file: file_name.sa
The hex88 converts an absolute object file to an ASCII file that can be input to the unused area filling utility fil88XXX.

START-UP FORMAT

```
hex88 -[o*] [drive:] <ifile>↵
```

flag:
Character string enclosed with [ ] means flag. Explanations for the flag is discussed later.

drive:
In case an absolute object file is not in current drive, input the drive name in front of the file name. It can be omitted if an input file is in current drive.

ifile:
Specify the file name input to the hex88. This file name can be input using either capital or small letters. When an <ifile> is not assigned, or when an assigned file name is a "-" (hyphen), file xeq is read.

Note: The extension for the absolute object file should be made as ".a".

FLAG

The hex88 can accept the following flag. The flag should be input with small letters.

Function	Flag	Explanation
Output file specification	`-o*`	Writes the output module for the file *. The default is standard output. (hex88 fixed setting flag)

ERROR MESSAGE

Error message	Description
bad file format	Input file format is incorrect.
can't read <input file>	Reading of the <input file> has failed.
can't write <output file>	Writing to the <output file> has failed.

RETURN VALUE

If an error message is not printed, in other words if all the records have meanings, and all the reading and writing is successful, the hex88 returns "success". Otherwise, the hex88 returns "failure".

EXAMPLE

Converts the absolute object file sample.a into the program data HEX file in the Motorola S2 format.

```
A>hex88 -o sample.sa sample.a↵
```

Appendix A List of sap88 Pseudo-Instructions

SOURCE FILE INSERTION PSEUDO-INSTRUCTION

Pseudo-instruction	*Function and format*
INCLUDE	Another file insertion `INCLUDE` <file name>

MACRO RELATED PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*
MACRO~ENDM	Macro definition <macro name> `MACRO` [<parameter> [, <parameter>] *] <statement string> `[EXITM]` <statement string> [<macro name>] `ENDM`
DEFINE	Character-string macro definition `DEFINE` <character-string macro name> [<substitute character-string>]
LOCAL	Definition of local label `LOCAL` [<local label name> [,<local label name>] *]
PURGE	Macro deletion `PURGE` [<macro name>]
UNDEF	Deletion of a character string macro `UNDEF` <character-string macro name>
IRP~ENDR	Repetition by character strings `IRP` <parameter>, <argument> [, <argument>] * <statement string> `ENDR`
IRPC~ENDR	Repetition by characters `IRPC` <parameter>, <argument character string> <statement string> `ENDR`
REPT~ENDR	Repetition by the specified number of times `REPT` <operation expression> <statement string> `ENDR`

CONDITIONAL ASSEMBLY PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*

IFC~ENDIF	Conditional assembly by conditional expression
	`IFC` <conditional expression>
	<statement string> [
	`ELESEC`
	<statement string>]
	`ENDIF`

IFDEF~ENDIF	Conditional assembly by the name either defined or undefined
	`IFDEF` <name>
	<statement string> [
	`ELSEC`
	<statement string>]
	`ENDIF`

IFNDEF~ENDIF	Conditional assembly by the name either undefined or defined
	`IFNDEF` <name>
	<statement string> [
	`ELSEC`
	<statement string>]
	`ENDIF`

Appendix B List of asm88 Pseudo-Instructions

SECTION SETTING PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*
CODE	Definition of CODE section `CODE`
DATA	Definition of DATA section `DATA`

DATA DEFINITION PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*

DB	Reserve/constant setting of the byte unit data area
	`DB` <expression> {,<expression>}*
	`DB` <expression> (<numeric expression>) {,<expression> (<numeric expression>)}*
	`DB` [<numeric expression>] {,[<numeric expression>]}*

DW	Reserve/constant setting of the word (2-byte) unit data area
	`DW` <expression> {,<expression>}*
	`DW` <expression> (<numeric expression>) {,<expression> (<numeric expression>)}*
	`DW` [<numeric expression>] {,[<numeric expression>]}*

DL	Reserve/constant setting of the long word (4-byte) unit data area
	`DL` <expression> {,<expression>}*
	`DL` <expression> (<numeric expression>) {,<expression> (<numeric expression>)}*
	`DL` [<numeric expression>] {,[<numeric expression>]}*

ASCII	ASCII text storing in memory `ASCII` character expression {, character expression}* character expression = character string | character constant | byte constant
PARITY	Setting/resetting of parity bit `PARITY` <operand>

SYMBOL DEFINITION PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*
EQU	Name value setting <name> `EQU` <expression>
SET	Name value setting <name> `SET` <expression>

LOCATION COUNTER CONTROL PSEUDO-INSTRUCTION

Pseudo-instruction	*Function and format*
ORG	Changing of location counter value `ORG` <expression>

EXTERNAL DEFINITION AND EXTERNAL REFERENCE PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*
EXTERNAL	Symbol external definition declaration `EXTERNAL` <symbol> {,<symbol>}*
PUBLIC	Global declaration of symbol `PUBLIC` <symbol> {,<symbol>}*

OUTPUT LIST CONTROL PSEUDO-INSTRUCTIONS

Pseudo-instruction	*Function and format*
LINENO	Change of line number for assembly list file `LINENO` <numeric expression>
SUBTITLE	Subtitle setting to assembly list file `SUBTITLE` <string>
SKIP	Suppresses all initialization codes output that exceed 4 bytes to assembly list file `SKIP`
NOSKIP	Outputs all initialization codes to assembly list file `NOSKIP`
LIST	Assembly list file output `LIST`
NOLIST	Prohibition of assembly list file output `NOLIST`
EJECT	Form feed of assembly list file `EJECT`

ASSEMBLY TERMINATION PSEUDO-INSTRUCTION

Pseudo-instruction	*Function and format*
END	Assembly stop `END` {<label>}

Appendix C Example for Mnemonic Notation

The examples for mnemonic notation in each addressing mode are shown in the below.

Addressing	Constant	Name name equ 50h	Label (default) label: address 00ffh	Default definition
#nn 0 to 255	eg.) ld a,#0ffh	eg.) ld a,#name	eg.) ld a,#label	-----
#mmnn 0 to 65535	eg.) ld ba,#1000h	eg.) ld ba,#name	eg.) ld ba,#label	-----
[br:ll] 0 to 255	eg.) ld b,[br:0ffh]	eg.) ld b,[br:name]	eg.) ld b,[br:label]	[br:low lod label]
[hhll] 0 to 65535	eg.) ld 1,[1000h]	eg.) ld l,[name]	eg.) ld l,[label]	[lod label]
[ix+dd] [iy+dd] [sp+dd] -128 to 127	eg.) ld [ix+10h],a	eg.) ld [ix+name],a	-----	-----
#hh 0 to 255	eg.) ld br,#0ffh	eg.) ld br,#name	eg.) ld br,#label	high lod label
#pp 0 to 255	eg.) ld ep,#05h	eg.) ld ep,#name	eg.) ld ep,#label	pod label
#bb 0 to 255	eg.) ld nb,#05h	eg.) ld nb,#name	eg.) ld nb,#label	boc label
rr -128 to 127	eg.) jrs 10h	eg.) jrs name	eg.) jrs label	loc label
[kk] 0 to 255	eg.) jp [10h]	eg.) jp [name]	eg.) jp [label]	[low lod label]
qqrr -32768 to 32767	eg.) jrl 1000h	eg.) jrl name	eg.) jrl label	loc label

- Meaning of the above mentioned default definitions are as follows:
  For example, when "jrl label" has been described, the cross assembler asm88 judges as "jrl loc label".

  ```
      jrl  label  →  jrl  loc label
  ```

  The program sequence is long jumped to the logical address converted from the physical address.

- An error occurs when the operand exceeding the above mentioned addressing range has been specified, or when it is judged to exceed it.

- In programming, pay attention to the following points when using the short branch or long branch instruction.

`jrs(l)  10H .....`	Jumps to the address at a distance of (10+1)H from current address
`jrs(l)  $+10H ...`	Jumps to the address at a distance of 10H from current address

Except for the above, notations described in the "E0C88 Core CPU Manual" can be used as is.

EPSON **International Sales Operations**

AMERICA

EPSON ELECTRONICS AMERICA, INC.

- HEADQUARTERS -
1960 E. Grand Avenue
El Segundo, CA 90245, U.S.A.
Phone: +1-310-955-5300 Fax: +1-310-955-5400

- SALES OFFICES -

West
150 River Oaks Parkway
San Jose, CA 95134, U.S.A.
Phone: +1-408-922-0200 Fax: +1-408-922-0238

Central
101 Virginia Street, Suite 290
Crystal Lake, IL 60014, U.S.A.
Phone: +1-815-455-7630 Fax: +1-815-455-7633

Northeast
301 Edgewater Place, Suite 120
Wakefield, MA 01880, U.S.A.
Phone: +1-781-246-3600 Fax: +1-781-246-5443

Southeast
3010 Royal Blvd. South, Suite 170
Alpharetta, GA 30005, U.S.A.
Phone: +1-877-EEA-0020 Fax: +1-770-777-2637

EUROPE

EPSON EUROPE ELECTRONICS GmbH

- HEADQUARTERS -
Riesstrasse 15
80992 Munich, GERMANY
Phone: +49-(0)89-14005-0 Fax: +49-(0)89-14005-110

- GERMANY -
SALES OFFICE
Altstadtstrasse 176
51379 Leverkusen, GERMANY
Phone: +49-(0)2171-5045-0 Fax: +49-(0)2171-5045-10

- UNITED KINGDOM -
UK BRANCH OFFICE
Unit 2.4, Doncastle House, Doncastle Road
Bracknell, Berkshire RG12 8PE, ENGLAND
Phone: +44-(0)1344-381700 Fax: +44-(0)1344-381701

- FRANCE -
FRENCH BRANCH OFFICE
1 Avenue de l' Atlantique, LP 915 Les Conquerants
Z.A. de Courtaboeuf 2, F-91976 Les Ulis Cedex, FRANCE
Phone: +33-(0)1-64862350 Fax: +33-(0)1-64862355

ASIA

- CHINA -
EPSON (CHINA) CO., LTD.
28F, Beijing Silver Tower 2# North RD DongSanHuan
ChaoYang District, Beijing, CHINA
Phone: 64106655 Fax: 64107319

SHANGHAI BRANCH
4F, Bldg., 27, No. 69, Gui Jing Road
Caohejing, Shanghai, CHINA
Phone: 21-6485-5552 Fax: 21-6485-0775

- HONG KONG, CHINA -
EPSON HONG KONG LTD.
20/F., Harbour Centre, 25 Harbour Road
Wanchai, HONG KONG
Phone: +852-2585-4600 Fax: +852-2827-4346
Telex: 65542 EPSCO HX

- TAIWAN -
EPSON TAIWAN TECHNOLOGY & TRADING LTD.
10F, No. 287, Nanking East Road, Sec. 3
Taipei, TAIWAN
Phone: 02-2717-7360 Fax: 02-2712-9164
Telex: 24444 EPSONTB

HSINCHU OFFICE
13F-3, No. 295, Kuang-Fu Road, Sec. 2
HsinChu 300, TAIWAN
Phone: 03-573-9900 Fax: 03-573-9169

- SINGAPORE -
EPSON SINGAPORE PTE., LTD.
No. 1 Temasek Avenue, #36-00
Millenia Tower, SINGAPORE 039192
Phone: +65-337-7911 Fax: +65-334-2716

- KOREA -
SEIKO EPSON CORPORATION KOREA OFFICE
50F, KLI 63 Bldg., 60 Yoido-dong
Youngdeungpo-Ku, Seoul, 150-763, KOREA
Phone: 02-784-6027 Fax: 02-767-3677

- JAPAN -
SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

Electronic Device Marketing Department
IC Marketing & Engineering Group
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5816 Fax: +81-(0)42-587-5624

ED International Marketing Department Europe & U.S.A.
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5812 Fax: +81-(0)42-587-5564

ED International Marketing Department Asia
421-8, Hino, Hino-shi, Tokyo 191-8501, JAPAN
Phone: +81-(0)42-587-5814 Fax: +81-(0)42-587-5110

ENERGY
SAVING
EPSON

In pursuit of **“Saving” Technology**, Epson electronic devices.
Our lineup of semiconductors, liquid crystal displays and quartz devices
assists in creating the products of our customers’ dreams.
Epson IS energy savings.

EPSON

SEIKO EPSON CORPORATION
ELECTRONIC DEVICES MARKETING DIVISION

■ EPSON Electronic Devices Website
http://www.epson.co.jp/device/

First issue JULY 1993, Printed JUNE 1999 in Japan Ⓜ A